

Finite-State Morphology:
Xerox Tools and Techniques

Pre-Publication Review Copy

Do Not Quote, Copy or Redistribute

Kenneth R. Beesley and Lauri Karttunen

May 27, 2002

Preface

This book will teach you how to use **Xerox** finite-state tools and techniques to build practical and efficient computer systems that perform morphological analysis and generation. The tools include **lexc**, a high-level language for specifying lexicons, **twolc**, a high-level language for specifying the rewrite rules used in phonology and morphology, and **xfst**, an interface that provides a regular-expression compiler and direct access to the XEROX FINITE-STATE CALCULUS, the toolbox of algorithms for building and manipulating finite-state networks. These tools have been used at the **Xerox Research Centre Europe (XRCE)**, the **Palo Alto Research Center (PARC)** and elsewhere to build many practical linguistic applications including tokenizers, morphological analyzer/generators, part-of-speech taggers and even syntactic “chunkers” and shallow parsers.

The CD-ROM included with this book contains executable versions of **xfst**, **lexc** and **twolc** compiled for the Solaris, Linux (both Intel and PowerPC), Windows (both NT and 2000), and Macintosh OS X operating systems. See the attached non-commercial software license agreement for details.

Linux is a trademark of Linus Torvalds. *Windows NT* and *Windows 2000* are trademarks of Microsoft Corporation. *Intel* is a trademark of Intel Corporation. *PowerPC* is a trademark of IBM Corporation. *OS X* is a trademark of Apple Corporation.

Webpage

For the latest information about this book and the finite-state software, point your Internet browser to the URL

<http://www.fsmbook.com/>

The site will eventually include

- Errata.
- Clarifications.
- New exercises, chapters, and other auxiliary material.
- Frequently Asked Questions (FAQ).
- Commercial software licensing information.
- Downloadable software updates and examples.

In addition the webpage will include information about how to subscribe to the email distribution lists dedicated to Xerox finite-state software. These lists serve as channels for official announcements and as forums for questions and tips among users.

Acknowledgements

Many people have contributed in one way or another to this book. We owe a general debt to pioneers in the field of finite-state linguistic theory, including C. Douglas Johnson (Johnson, 1972), now Professor Emeritus of Linguistics at the University of California at Santa Barbara. Ronald Kaplan and Martin Kay rediscovered the key insights of Johnson (Kaplan and Kay, 1981; Kaplan and Kay, 1994) and developed the first set of practical and robust algorithms, originally written in INTERLISP, for finite-state computing at the **Xerox Palo Alto Research Center (PARC)**. Kimmo Koskenniemi's influential Two-Level Morphology (Koskenniemi, 1983), based on work from **PARC** and popularized mainly by Lauri Karttunen (Karttunen, 1983), was an ingenious but limited implementation of finite-state morphology that worked without an automatic rule compiler or the algorithms for manipulating finite-state networks. Kaplan, Karttunen, Koskenniemi and other colleagues and students participated in the development of the first finite-state rule compilers (Koskenniemi, 1986; Karttunen et al., 1987) at CSLI at Stanford. Karttunen ported the algorithms into Common Lisp and, with Todd Yampol, ported them again into C. Karttunen and Kenneth Beesley wrote **twolc** (Karttunen and Beesley, 1992), the two-level rule compiler, in C; and Karttunen and Yampol wrote **lexc** (Karttunen, 1993), the lexicon compiler. Yampol also wrote an early interface to the finite-state algorithms called **ifsm**, which was eventually replaced by **fsc**, written by Pasi Tapanainen. The newest interface, **xfst**, and the library of finite-state algorithms are currently maintained and expanded by Lauri Karttunen, André Kempe, and Tamás Gaál at the **Xerox Research Centre Europe (XRCE)** in Grenoble, France.

We also wish to thank our students and colleagues for their feedback during finite-state training and development. Some, including Mike Maxwell of the Linguistic Data Consortium, have gone out of their way to send thoughtful problem reports and suggestions that have led to better code and documentation.

The Authors

Lauri Karttunen

Lauri Karttunen is a Research Fellow at PARC Inc. in Palo Alto, California, USA. He received his Ph.D. in Linguistics from Indiana University in 1969 with a dissertation on discourse referents and pronoun/antecedent relations (Bach-Peters sentences). As a Linguistics professor at the University of Texas at Austin, 1968-1983, he worked mostly on formal semantics. His papers from that period are about presuppositions, conventional implicatures, and the semantics of questions.

During his last years at the University of Texas, Karttunen became increasingly interested in computational linguistics, and his 1983 KIMMO system was an early and highly influential implementation of Koskenniemi's Two-Level Morphology. At the Stanford Research Institute (SRI) Artificial Intelligence Laboratory from 1984 to 1987, his primary interest was PATR-II, a unification-based grammar formalism. At **Xerox** from 1987 to 2002, and now at PARC Inc., Karttunen has been an active developer of finite-state technology and applications.

Kenneth R. Beesley

Kenneth R. Beesley is a Principal Scientist at the **Xerox Research Centre Europe** in Grenoble, France. He received a D.Phil. in Epistemics (now renamed Cognitive Science) at the University of Edinburgh in 1983, with a dissertation on English adjectives in Montague Grammar. He spent six years at Alpnet (né ALPS), working on computer-assisted translation and glossing, and three years at Microlytics, a **Xerox** spinoff company, before joining **Xerox Corporation** in 1993. Since becoming exposed to finite-state morphology in 1988, through Lauri Karttunen, he has worked on finite-state compilers and numerous applications, including morphological analyzers for Arabic, Aymara, Malay, Spanish, Portuguese, Dutch, and Italian.

Introduction

0.1 Target Audience

This book teaches linguists how to use the **Xerox** finite-state tools and techniques to build useful and efficient programs that process text in NATURAL LANGUAGES such as French, English, Spanish, Yoruba, Navajo and Mongolian. Most of the presentation will center around morphological analysis and generation, but many other applications are possible.

The presentation is based on years of practical development and training of computational linguists at the **Xerox Palo Alto Research Center (PARC)** and the **Xerox Research Centre Europe (XRCE)**. It is aimed squarely at our trainees, who are typically sophisticated native speakers of the language they wish to work on and have at least a basic background in computing and formal linguistics. However, few trainees come to us with an adequate understanding of the formal properties of finite-state networks, and even our professional colleagues often need to have their eyes opened to the practical possibilities of computing with finite-state networks. Beyond learning the syntax and idioms of the various **Xerox** finite-state tools, the bigger challenge is to appreciate the possibilities and develop the finite-state mindset that is so different from traditional algorithmic or procedural computing.

Most publications in finite-state theory are addressed to a small professional community, including expert teams at **Xerox**, Mitsubishi, Paris VII University, Groningen University, and ITT Research. Long experience has shown that the terse technical presentation appropriate for that audience is too often a closed book to the working linguist who is recruited to sit down and build a real system, for example, a finite-state morphological analyzer for Romanian. This book is therefore a popularization, facing all the dangers of that medium, aimed not quite at the proverbial man on the street but at least at the working linguist. We try to follow the rules for good popularization, including the use of wordier definitions, plentiful practical examples and tutorial exercises, while at the same time staying honest and accurate. Suitable mathematical rigor must not and ultimately cannot be avoided, but we try to ease into it gradually.

Our training sessions have repeatedly shown that certain key concepts, like composition and the semantics of **twolc** rule operators, are surprisingly difficult to grasp and yet must be understood viscerally if a trainee is to catch on and be

able to understand, write and especially debug rules. We therefore do not eschew repetition, sometimes explaining the same concept in several ways, hoping that one will click.

While it is fashionable for programming books to claim that “no previous experience is required”, we cannot honestly do that here. We do assume an acquaintance with UNIX-like operating systems, the ability to use a text editor like emacs, xemacs or vi, and some kind of previous programming experience in a language like C, C++, Java, Perl, etc. Trainees without programming experience often have a difficult time in our courses.

We also assume that our students are trained in formal linguistics and are capable of looking at their language objectively and explaining in some kind of formal terms what is going on. This book explains how to formalize linguistic facts in the various **Xerox** finite-state notations, but it cannot tell you what the facts are or train you to be a formal linguist. The tools cannot do the linguistics for you.

Having said that, we try to assume no acquaintance with finite-state networks or formal-language theory. We provide a whole chapter entitled “A Gentle Introduction” to finite-state networks, wherein we begin at the beginning, even explaining what we mean by “finite”, “state”, and “network”; and laying out in intuitive fashion what finite-state networks can do and why they are Good Things. Once the general concepts are established, we progress to a more formal introduction; and we present the subsequent material in a gradual progression of readings and lessons, complete with practical exercises. Students are strongly urged to do the exercises.

0.2 Finite-State Tools

Finite-state computing involves specifying various finite-state networks and combining them together mathematically into applications such as tokenizers and morphological analyzers. Traditional grammar components such as lexicons, morphotactic rules, morphotactic filters, phonological and orthographical alternation rules and even some syntax rules are all implemented in this book as finite-state networks. **Xerox** has created an integrated set of software tools to facilitate such linguistic development:

- **xfst** is an interactive interface providing access to the basic algorithms of the Finite-State Calculus, providing maximum flexibility in defining and manipulating finite-state networks. **xfst** also provides a compiler for an extended metalanguage of regular expressions, which includes a powerful rule formalism known as replace rules.
- **lexc** is a high-level declarative language used to specify natural-language lexicons. The syntax of the language is designed to facilitate the definition of morphotactic structure, the treatment of gross irregularities, and the addition of the tens of thousands of baseforms typically encountered in a natural

language. **lexc** source files are produced with a text editor such as emacs, and the result of the compilation is a finite-state network.

- **twolc** is a high-level declarative language designed for specifying, in classic two-level format, the alternation rules required in phonological and orthographical descriptions. As with **lexc**, **twolc** source files are written using a common text editor and are compiled into finite-state networks.

Although **Xerox** developers have now largely abandoned **twolc** rules in favor of the newer Replace Rules of **xfst**, we continue to supply and document **twolc** for those who support legacy systems and for those who might simply prefer writing rules in the two-level style.

0.3 Applications

The mathematical properties of finite-state networks have been well understood for a long time, but it was once generally believed that finite-state grammars were too weak in descriptive power to model interesting phenomena in natural language.

More recently, although finite-state power and natural languages have not changed, the descriptive possibilities of finite-state grammars have been reexamined more positively, and the availability of practical tools like **xfst**, **lexc**, and **twolc** has made possible the development of

- Finite-state morphological analyzer/generators (often called LEXICAL TRANSDUCERS at **Xerox**) for English, French, German, Spanish, Portuguese, Dutch, Italian, Arabic and other languages;
- Finite-state tokenizers that divide an input string into tokens, i.e. words and various multi-word strings, for further morphological or syntactic processing;
- Finite-state part-of-speech disambiguators or “taggers” that examine tokens, which are often ambiguous, in their syntactic context and disambiguate them, assigning a single TAG such as *NOUN* or *VERB* to each token; and
- Finite-state shallow syntactic parsers, such as nominal-phrase identifiers that recognize syntactic patterns in a tagged text and bracket them.

Morphological analysis is the basic enabling application for further kinds of natural-language processing, including part-of-speech tagging, parsing, translation and other high-level applications. This book concentrates on morphological analysis while explaining the tools and techniques that can also be used to build larger systems.

The two central problems in morphology are

1. WORD FORMATION, also called MORPHOTACTICS or, in other traditions, MORPHOSYNTAX: Words are composed of smaller units of meaning called MORPHEMES. The morphemes that make up a word are constrained to appear in certain combinations and orders: *piti-less-ness* and *un-guard-ed-ly* are valid words of English, but **piti-ness-less* and **un-elephant-ed-ly* are not.
2. PHONOLOGICAL and ORTHOGRAPHICAL ALTERNATION: The spelling (or sound) shape of a morpheme often depends on the environment. In English we note the following alternations among many, and similar phenomena appear in almost all languages.
 - *pity* is realized as *piti* in the context of a following *less*
 - *fly* is realized as *flie* in the context of a following *s*
 - *die* is realized as *dy*, and *swim* as *swimm*, in the context of a following *ing*

The **Xerox** work in computational morphology is based on the fundamental insight that both problems, morphotactics and alternation, can be solved with the help of finite-state networks:

1. The legal combinations of morphemes (MORPHOTACTICS) can be encoded as a finite-state network;
2. The rules that determine the form of each morpheme (ALTERNATION) can be implemented as finite-state transducers; and
3. The lexicon network and the rule transducers can be composed together into a single network, a LEXICAL TRANSDUCER, that incorporates all the morphological information about the language including the lexicon of morphemes, derivation, inflection, alternation, infixation, interdigitation, compounding, etc.

Lexical transducers have many advantages. As well as being mathematically beautiful, bidirectional and highly efficient, these finite-state applications can have wide lexical coverage, take up little memory space, and be robust, commercially-viable products. In 1996 the **Xerox** Spanish Lexical Transducer, for example, contained over 46,000 baseforms, analyzed and generated over 3,400,000 inflected wordforms, and yet occupied only 3349 kbytes of memory, or about 1 byte per inflected wordform. For commercial applications, it can also be further compressed, and yet run directly in the compressed form, using **Xerox** compression algorithms and runtime code. The same language-independent runtime code is used for all languages.

Large lexical transducers and taggers now exist for English, French, German, Spanish, Portuguese, Dutch, Italian and Japanese, and the extension of finite-state

techniques to process new languages has become almost routine at **Xerox** and among our research and business clients. For example, significant work has already been done on the morphological analysis of Basque, Turkish, Arabic, Finnish, Swedish, Norwegian, Danish, and several Eastern European languages.

0.4 Requirements

0.4.1 Software

Xerox Finite-State Tools

This book is a hands-on tutorial in the use of **xfst**, **lexc** and **twolc**, so the student should have these tools installed and running from the beginning. The enclosed CD-ROM contains executable version of these programs, compiled for the Solaris and Linux (both Intel and PowerPC) operating systems. See the CD-ROM for information on non-commercial licensing and installation.

For any questions about commercial licensing and any other uses beyond those permitted by the attached license agreement, contact **Xerox** at the following address:

Xerox Research Centre Europe
ATTN: Licensing of Finite-State Programming Languages
6, chemin de Maupertuis
38240 MEYLAN
France

Tel. from inside France: 04 76 61 50 50
Tel. international +33 4 76 61 50 50

Fax from inside France: 04 76 61 50 99
Fax international +33 4 76 61 50 99

<http://www.xrce.xerox.com/>
xerox-fs@xrce.xerox.com

Text Editor

The source files for **xfst**, **lexc** and **twolc** are best created using commonly available text editors such as emacs, xemacs, or vi. If you use a word processor such as Microsoft Word, you must be careful to save the buffers as text-only files.

The **Xerox** finite-state tools have been written to accommodate UNICODE input, which would have distinct advantages in many applications. However, the difficulties of UNICODE text-editing, display and printing make this option effectively unavailable at this time.

0.4.2 Hardware

The finite-state tools are available in object form, compiled for SUNOS or Solaris (SUN workstations), for Microsoft, and for LINUX. Although modest experiments can be performed on modest machines, large-scale linguistic development can often require serious computer crunch and large amounts of RAM. Although the final results of most morphological analyzers, for example, are under five megabytes in size, and while the **Xerox** finite-state algorithms are the fastest and most memory-efficient available, intermediate stages of various operations can often explode in size before minimization operations can be invoked. While 24 or 36 MBytes of RAM might be considered a minimum for full-scale work on most languages, **XRCE** maintains SUN workstations with up to 2 GBytes of RAM to handle particularly intensive finite-state computations.

0.5 Other Documentation

This book replaces previous documentation, some of which is terse, lacking in examples, and increasingly out of date. Where appropriate, large sections of the following documents have been recycled

Karttunen, Lauri. *Finite-State Lexicon Compiler*, ISTL-NLTT-1993-04-02. Palo Alto: Xerox Corporation.

Karttunen, Lauri and Beesley, Kenneth R. *Two-Level Rule Compiler*, ISTL-92-2. Palo Alto: Xerox Corporation.

Online documentation is also available at

<http://www.xrce.xerox.com/research/mltt/fst/>

0.6 Email Distribution List

[Complete this section with information about an email distribution list for those interested in Xerox Finite-State Tools.]

Contents

0.1	Target Audience	vii
0.2	Finite-State Tools	viii
0.3	Applications	ix
0.4	Requirements	xi
0.5	Other Documentation	xii
0.6	Email Distribution List	xii
 I Networks and Regular Expressions		1
 1 A Gentle Introduction		3
1.1	The Beginning	4
1.2	Some Unavoidable Terminology	4
1.3	A Few Intuitive Examples	11
1.4	Sharing Structure	19
1.5	Some Background in Set Theory	20
1.6	Composition and Rule Application	33
1.7	Lexicon and Rules	34
1.8	Where We are Heading	38
1.9	Finite-State Networks are Good Things	39
1.10	Exercises	40
 2 A Systematic Introduction		45
2.1	Where We Are	45
2.2	Basic Concepts	46
2.3	Simple Regular Expressions	47
2.4	Complex Regular Expressions	64
2.5	Properties of Finite-State Networks	75
2.6	Exercises	78
 3 The xfst Interface		81
3.1	Introduction	84
3.2	Compiling Regular Expressions	87
3.3	Interacting with the Operating System	112

3.4	Incremental Computation of Networks	125
3.5	Rule-Like Notations	134
3.6	Examining Networks	186
3.7	Miscellaneous Operations on Networks	194
3.8	Advanced xfst	200
3.9	Operator Precedence	206
3.10	Conclusion	206
II	Finite-State Compilers	207
4	The Finite-State Lexicon Compiler: lexc	209
4.1	Introduction	212
4.2	Defining Simple Automata	213
4.3	Defining Lexical Transducers	238
4.4	The lexc Interface	251
4.5	Useful lexc Strategies	258
4.6	Integration and Testing	278
4.7	lexc Summary and Usage Notes	298
5	The twolc Language	307
5.1	Introduction to twolc	308
5.2	Basic twolc Syntax	321
5.3	Full twolc Syntax	336
5.4	The Art and Craft of Writing twolc Grammars	346
5.5	Debugging twolc Rules	352
5.6	Final Reflections on Two-Level Rules	369
III	Finite-State Systems	373
6	Planning and Managing Finite-State Projects	375
6.1	Infrastructure	377
6.2	Linguistic Planning	380
6.3	Composition is Our Friend	390
6.4	Priority Union for Handling Irregular Forms	397
6.5	Conclusions	407
7	Testing and Debugging	409
7.1	The Challenge of Testing and Debugging	410
7.2	Testing on Real Corpora	410
7.3	Checking the Alphabet	418
7.4	Testing with Subtraction	427

8	Flag Diacritics	439
8.1	Features in Finite-State Systems	440
8.2	What Are Flag Diacritics?	441
8.3	Using Flag Diacritics	443
8.4	Flag Diacritics and Finite-State Algorithms	457
8.5	Examples	466
9	Non-Concatenative Morphotactics	477
9.1	Introduction	478
9.2	Formal Morphotactic Description	481
9.3	Practical Non-Concatenative Examples	487
9.4	Usage Notes for compile-replace	518
9.5	Debugging Tips for compile-replace	523
9.6	The Formal Power of Morphotactics	525
9.7	Conclusion	526
10	Finite-State Linguistic Applications	527
10.1	Introduction	528
10.2	The tokenize Utility	529
10.3	The lookup Utility	539
10.4	Using xfst in Batch Mode	546
10.5	Transducers for Morphological Analysis	547
10.6	Spelling	557
10.7	Beyond Tokenization and Morphological Analysis	559
10.8	Application Summary	561
11	Computational Complexity	563
11.1	Complexity	563
11.2	Finite-State Solutions to Constraint Problems	565
IV	Appendices	579
A	Graphing Quiz	581
B	Suggested Analysis Tags	585
B.1	The Challenge of Tag Choice	586
B.2	Tag-Name Spelling	586
B.3	Principles for Tag-Name Choice	588
B.4	Suggested Tags to Choose From	589
B.5	Miscellaneous Problematic Leftovers	599

C	Makefiles	601
C.1	Introduction	601
C.2	Example: A Simple Morphological Analyzer	603
C.3	Example: Separating Source, Intermediate, and Binary Files	609
C.4	twolc and Makefiles	611
C.5	quit-on-fail	612
C.6	Conclusion	612
D	Solutions to Exercises	613
D.1	Graphing Quiz	614
D.2	The Better Cola Machine Diagram	621
D.3	The Better Cola Machine Network	621
D.4	Brazilian Portuguese Pronunciation	624
D.5	The Monish Language	631
D.6	Tag Moving	635
D.7	Esperanto Nouns	637
D.8	Esperanto Adjectives	638
D.9	Esperanto Nouns and Adjectives	639
D.10	Esperanto Nouns and Adjectives with Upper-Side Tags	640
D.11	Esperanto Nouns, Adjectives and Verbs	641
D.12	Einstein II Problem	644

Part I

**Networks and Regular
Expressions**

Chapter 1

A Gentle Introduction

Contents

1.1	The Beginning	4
1.2	Some Unavoidable Terminology	4
1.2.1	State	5
1.2.2	Finite	7
1.2.3	Networks	8
1.3	A Few Intuitive Examples	11
1.3.1	Finite-State Languages and Natural Languages	11
1.3.2	Symbol Matching and Analysis	12
1.3.3	Getting More Back from Analysis: Transducers	14
1.3.4	Generation	17
1.3.5	Further Finite-State Applications	18
1.4	Sharing Structure	19
1.5	Some Background in Set Theory	20
1.5.1	Sets	21
	What is a Set?	21
	The Empty Set	21
	Infinite Sets	22
	Universal Set	22
	Ordered Set	22
1.5.2	Relations	22
	Infinite Relations	24
	Identity Relation	25
1.5.3	Some Basic Set Operations	25
	Union	25
	Intersection	27
	Subtraction	28

Concatenation	29
Composition	30
Projection	33
1.6 Composition and Rule Application	33
1.7 Lexicon and Rules	34
1.8 Where We are Heading	38
1.9 Finite-State Networks are Good Things	39
1.10 Exercises	40
1.10.1 The Better Cola Machine	40
1.10.2 The Softdrink Machine	41
1.10.3 Relations and Relatives	43
Father-in-Law	43
Cousin	43
1.10.4 Lowercase/Uppercase Composition	43

This book shows how to use the **Xerox** finite-state programming languages **xfst**, **lexc** and **twolc** to create finite-state networks that perform morphological analysis and related tasks.

1.1 The Beginning

Let's begin at the beginning: What are FINITE-STATE NETWORKS and why should anyone care about them?

The following gentle introduction will attempt to paint the big picture, avoiding technical vocabulary and mathematical definitions, but conveying the core concepts necessary to make sense of all the formalism that must follow. We'll try to provide a vision of where we're headed, showing how finite-state networks, created using various **Xerox** tools, are combined together into practical working applications, and in particular into morphological analyzers. Finally, we will try to convey the general notion that finite-state networks are Good Things, useful for doing many kinds of linguistic processing. Compared to the alternatives, applications based on finite-state networks are usually smaller, faster, more elegant, and easier to maintain and modify.

Analogies and explanations in this gentle introduction are not to be taken too far or too seriously; some will certainly not stand up to rigorous scrutiny. Experienced computational linguists and computer scientists can safely skip this section and move on to the more formal introduction in the next chapter.

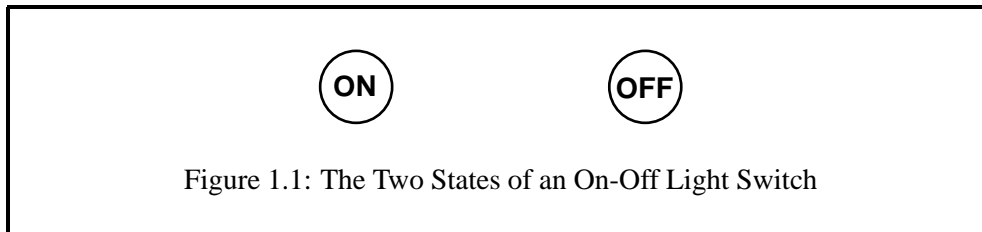
1.2 Some Unavoidable Terminology

First of all, what is meant by the terms FINITE, STATE and NETWORK? Let's start with the term STATE, which has everyday meanings that are closely related to the formal one we will need.

1.2.1 State

Substances, people, and physical machines are often said to be in one state or another. For example, H_2O is a substance that can exist in a frozen state (ice), a liquid state (water), or a gaseous state (water vapor). Under the right conditions, H_2O will change from one state to another. Similarly, a person, as far as emotions are concerned, may at any given time be in a happy state, a depressed state, an excited state, a bored state, an amorous state, etc.; and like H_2O , people usually do a lot of changing from one state to another until they reach a final dead state. Finally, many physical machines in the real world can be described in terms of the states they can assume and the way that they change from one state to another.

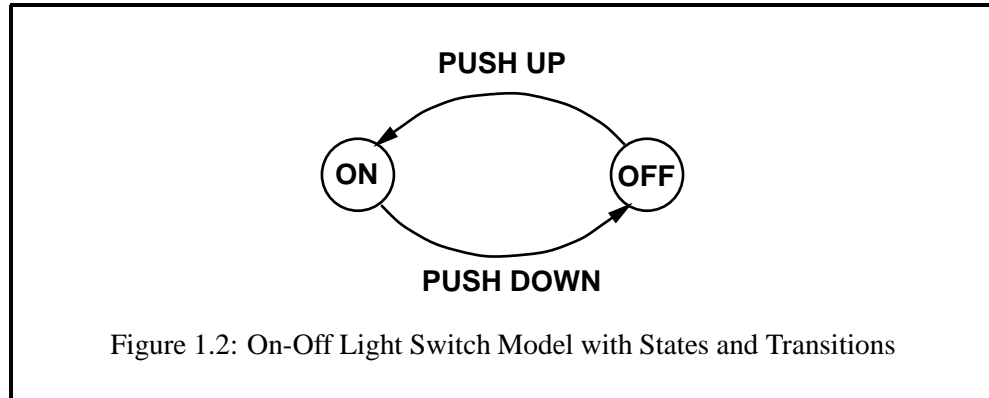
While it is perhaps most common to talk about machine states when discussing complex machines like computers, let's start with some simpler mechanical machines that are easier to grasp. Consider first a common light switch, which is always in one of two states, which we will label **ON** and **OFF**. By convention, we will represent or MODEL the states of this and other machines with circles as in Figure 1.1.



A common light switch changes from the **ON** state to the **OFF** state, and vice versa, when we humans physically manipulate it from one state to the other. Let us assume that the **OFF** state corresponds to the switch lever being down and that the **ON** state corresponds to the switch lever being up. The things that we can do to a light switch are pretty limited: when the switch is down (**OFF** state) we can push it up; and when the switch is up (**ON** state) we can push it down. The possible TRANSITIONS from one state to another are modeled, by convention, with arrow-line ARCS leading from one state to another. Each arc is labeled with the action or INPUT that causes that particular change of state.

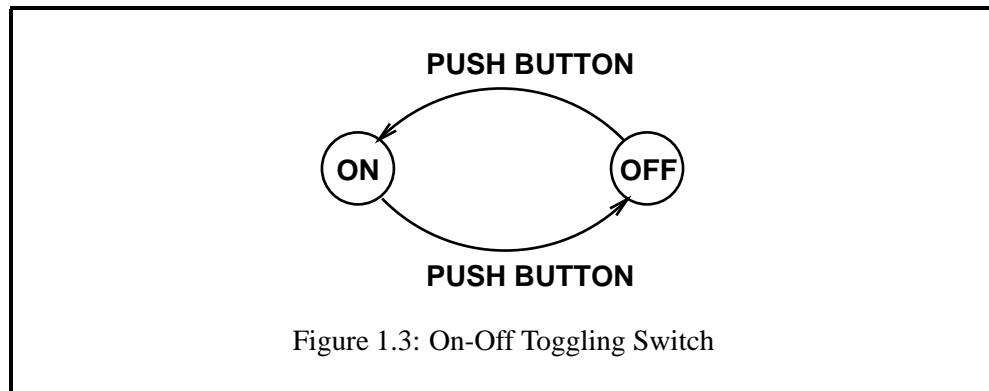
Note that our little graphic model of the light-switch machine in Figure 1.2 ignores details of physical material, shape, electrical contacts and wiring that can vary greatly and don't usually concern us. What our model does capture is the state behavior of the light switch, showing all the possible states it can assume, all the possible inputs, and how the inputs cause transitions from one state to another. Other inputs, such as burning the switch with a blowtorch or pounding it with a hammer, are conceivable but illegal—they would result in the jamming or even the complete destruction of the machine. Such illegal inputs are simply absent from the model.

Note also that particular inputs may be legal only when the machine is in par-



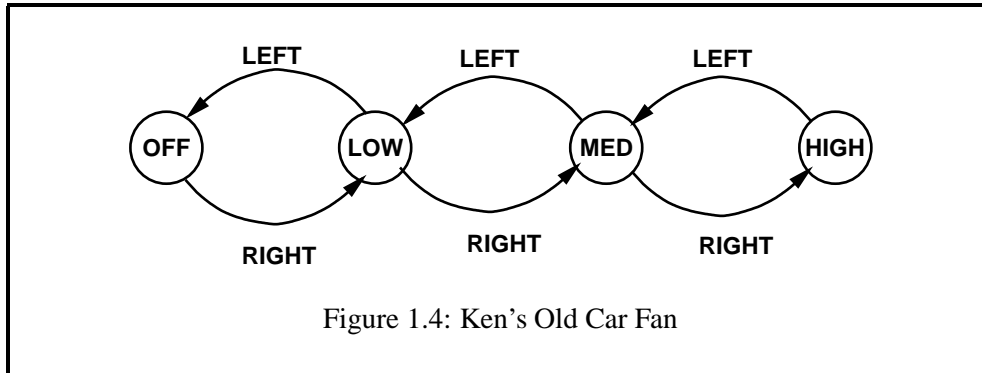
ticular states. Thus when the switch is in the **ON** state, Pushing Down is legal; but if the switch is already in the **OFF** state, Pushing Down is illegal as it would break the machine. Illegal inputs and transitions are simply left out of the model.

Now let's model another kind of on-off light switch that has a single button to push rather than a lever to manipulate. Pushing the single button toggles the switch from **ON** to **OFF** and also from **OFF** to **ON**. Such a toggling switch has but a single legal input, pushing the button, as shown in Figure 1.3. Notice here that the effect of an input to a machine can differ radically depending on the current state of the machine.



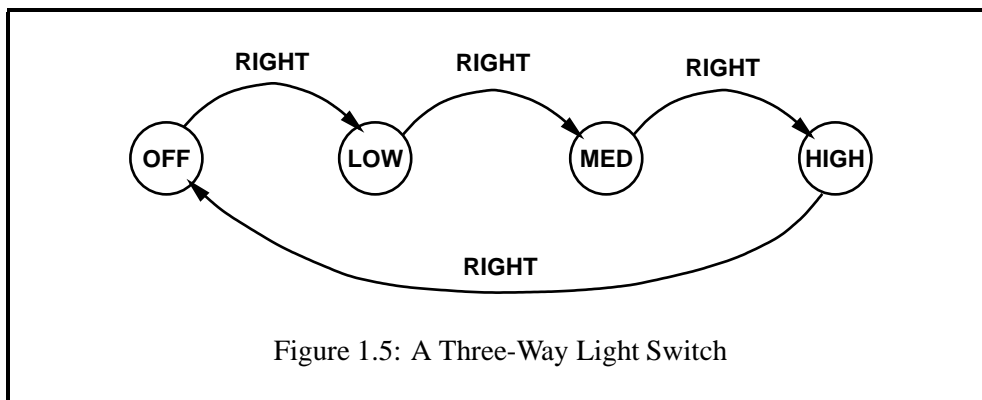
Before we push these examples too far, let's move on to a slightly more complex physical machine and model its behavior in a similar way. The fan in Ken's old car had exactly four settings or states in which it could be placed, **OFF**, **LOW**, **MED** (medium), and **HIGH**. One could change the state by turning a dial Left (counterclockwise) or Right (clockwise). Figure 1.4 is a state model of this particular fan machine.

The way the fan control was built, it allowed no direct transition from **OFF** to **MED** or **HIGH**, from **LOW** to **HIGH**, from **MED** to **OFF**, or from **HIGH** to **LOW** or **OFF**; one had to move the machine, even if rather quickly, through the intermediate states. From the **OFF** state, turning the dial Left again would break



the machine; the result of turning the dial Right from the **HIGH** state would be similarly unfortunate. Such illegal inputs are simply not shown in the model.

We can easily imagine, and find, similar machines with dials that turn completely around in either direction, or perhaps only in one direction, allowing transitions between the **OFF** and **HIGH** states. So-called *3-way* lamps in the USA have four states, **OFF**, **LOW**, **MED**, **HIGH**, controlled by a dial that turns all the way around, but only to the right, and these machines are modeled in Figure 1.5. Comparing this machine with the one in Figure 1.4, we see that two machine can have exactly the same states but different inputs and transitions.



1.2.2 Finite

Another inescapable technical term within finite-state networks is **FINITE**, which refers to the number of states and can be satisfactorily defined, for present purposes, as simply being “not infinite”.

So far we have looked at some very simple physical machines and have modeled them in terms of possible states and transitions from one state to another. The examples have been restricted to those with a finite number of states. Consider one more kind of light control, the dimmer or rheostat, that allows an infinite number of gradations (states) between fully **OFF** and fully **ON**; such a machine is not

finite-state, and we cannot model a light dimmer with a finite number of circles and arcs. Similarly, human beings display mixtures and infinite gradations of emotional states that preclude a finite-state modeling.

1.2.3 Networks

The final term to examine is NETWORK, which will need to be reexamined with more rigor in the following chapter. For now, let us accept that networks are graph-like structures of nodes linked together with directed arc-transitions as shown in Figures 1.4 and 1.5.

Other terms for finite-state network commonly used in more formal discussion are FINITE-STATE MACHINE and FINITE-STATE AUTOMATON (plural AUTOMATA). Other more complicated networks are called TRANSDUCERS. The word automaton is also commonly used to describe robots, especially those that seem to move about and act under their own will. The finite-state networks that we will be defining and manipulating may not be as cute as little metal rodents negotiating a maze, but we will see that they are indeed abstract machines that can perform some interesting linguistic tasks for us.

As a stepping stone to the linguistic examples, let us examine and model one last mechanical machine called the Cola Machine. It is based on the familiar coin-operated soft-drink machines, and we will limit our modeling to that part of the machine that accepts our coin inputs and decides when we have entered enough money to deserve a drink. We will not try to model the refrigerator, the drink-selection system, the drink-delivery system, or anything else. To further facilitate the example, we specify that our cola machine has the following characteristics:

1. Drinks cost 25 cents in US currency.
2. The only coins accepted by the machine are
 - The quarter (abbreviated **Q**) which is worth 25 cents,
 - The dime (abbreviated **D**) which is worth 10 cents, and
 - The nickel (abbreviated **N**) which is worth 5 cents.
3. The machine accepts any combination of these coins, in any order, that add up to 25 cents.
4. The machine requires exact change.

This coin-accepting machine is in fact a finite-state machine, and our task is to model it as a network of states and transitions. If we walk up this machine, before we put any money into it, it will be in a START STATE that we can label helpfully as **0** (zero). In the **0** state, it will steadfastly refuse to deliver a drink, and our job is to enter appropriate coins that change the state of the machine until it is in a special

FINAL or ACCEPTING STATE, which we will label as **25**, that allows a drink to be selected. By convention, we will use a double circle to mark such a final state.

The first obvious way to change the machine from the start state to the final state is to add a quarter as in Figure 1.6. We can then select our preferred beverage, the coin-accepting machine will reset to state zero (magically for now), and it will be ready for the next thirsty customer.

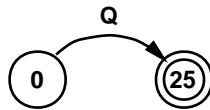


Figure 1.6: The Cola Machine Accepts a Quarter

There are of course other ways to get a drink, by adding various combinations and permutations of nickels and dimes. If we start by adding a nickel (**N**), the resulting new state of the machine will be closer to the final goal, but the machine still won't give us a drink; let's label the new non-final state **5** as in Figure 1.7. If we continue to add four more nickels, we will reach three more non-final states and, at last, the final state. The complete model of our simple Cola Machine is shown in Figure 1.8.

Adding dimes will, in this particular machine, cause a jump of two states at a time. For example, adding a nickel (**N**), a dime (**D**) and another dime (**D**), in that order, will move the machine from the **0** state to the **5** state to the **15** state and finally to the **25** state.

Now that we have our complete coin-accepting machine modeled, we list all the possible sequences of coins that it will accept, where acceptance means reaching

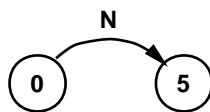
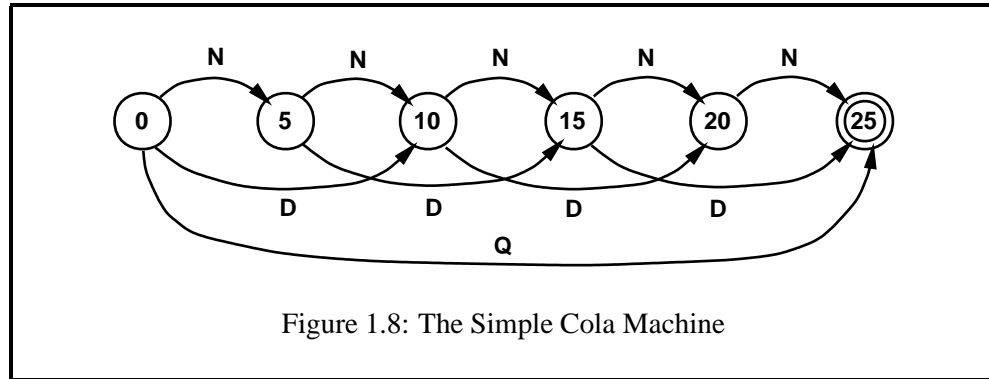


Figure 1.7: Inputting a Nickel to the Cola Machine



the final state and allowing us to get a drink.

Q
DDN
DND
NDD
DNNN
NDNN
NNDN
NNND
NNNNN

These nine sequences of coins are the only valid inputs, to this particular cola machine, if you want to get a drink. Any other sequence of coins, including too few coins, or too many coins (in our simple machine that requires exact change) simply won't work. Other kinds of coins, like pennies (worth one cent) or any kind of foreign coins including Mexican pesos or Canadian quarters, are simply illegal inputs and will cause the mechanism to jam.

Here is where we can make the transition from our mechanical machines to the linguistic ones that we will learn about and build in this book:

- Think of the inputs to the machine not as coins (quarters, dimes and nickels) but as letter SYMBOLS like **Q**, **D** and **N**.
- The set of valid symbols that the machine will accept is its ALPHABET.
- The sequences of symbols that the machine will accept are WORDS.
- The entire set of words that the machine accepts or recognizes is its LANGUAGE.

In this case, the nine words listed above would constitute the entire Cola Machine Language.

This technical use of the word language, to denote just a collection of symbol strings (words), is somewhat odd, but we are stuck with it in formal language theory. In what follows, we will see that the task of the computational linguist is usually to make the formal language accepted by our finite-state network match as closely as possible to a natural language, such as French and Spanish. Our task, in short, is to model natural languages.

1.3 A Few Intuitive Examples

1.3.1 Finite-State Languages and Natural Languages

All of our technical terms, especially the odd usage of ALPHABET and LANGUAGE, will be made more precise as we progress. What we will do now is present a few small but fairly typical finite-state networks that linguists might build, and we will suggest ways that they could be genuinely useful in various kinds of natural-language processing.

Figure 1.9 shows a very small finite-state network that accepts the single word “canto”, which happens to look like a word in Spanish. We can also say that the language of this machine consists of the single word “canto”. If we think of the real Spanish language, rather perversely, as just the set of all its possible written words, then our new network models a very small subset of Spanish. The alphabet of the machine consists of just five symbols: **c**, **a**, **n**, **t** and **o**. The machine itself consists of a start state (if not overtly labeled, the start state is the leftmost one in our diagrams), a final state (marked with two circles), and non-final states in between linked by arcs labeled **c**, **a**, **n**, **t** and **o**, in that order.

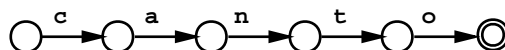


Figure 1.9: One-Word Language

If we walk up to our new machine, like we walked up to the Cola Machine with our coins, and if we enter the symbols **c**, **a**, **n**, **t** and **o**, in that order, the machine will transition through a series of states, ending up in the final state, and the word will be accepted. We don’t get a drink as a reward this time, but the finite-state machine will in essence tell us “I accept this string”, which means “This string is in my language”. If we enter any other string, such as “libro”, it will be rejected.

Now let’s imagine a slightly larger machine, shown in Figure 1.10, that accepts a language consisting of three strings, “canto”, “tigre” and “mesa”. Again, all of these words happen to look like words of Spanish. Again, each valid word corresponds to the symbols on a path from the start state (by convention the leftmost state in the diagram) to a final state (indicated with a double circle). This

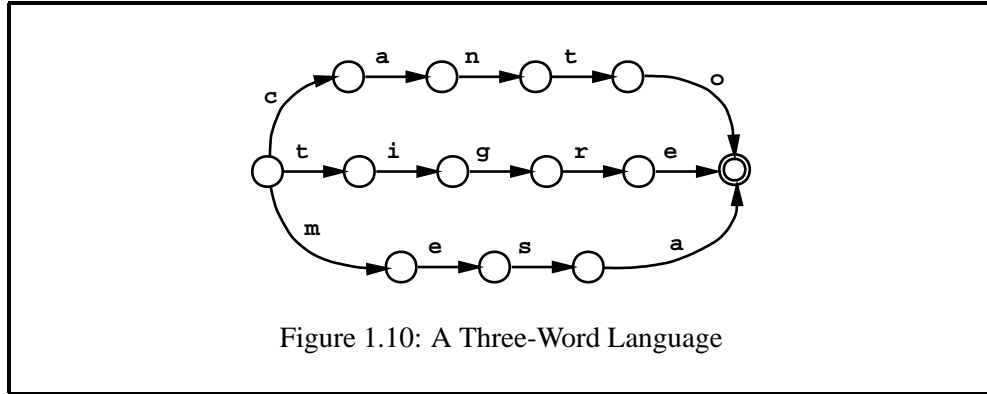


Figure 1.10: A Three-Word Language

new machine recognizes a language with a bigger alphabet and vocabulary, but it is not different in type or usage from the first. If you enter the symbols **m**, **e**, **s** and **a**, in that order, the word “mesa” will be accepted, as will any other string of symbols representing a word in the language of the network. Enter any word not in the language, such as “panes”, and it will be rejected.

Now imagine the same network expanded to include three million words, all of them happening to correspond to words of the real Spanish language. At this point, we have a potentially valuable basis for several natural-language-processing systems, the most obvious being a spelling checker. Simple spell-checking involves somehow looking up each word from a document to see if it is a valid word, and flagging all the words that are not found. Given that we have a large finite-state network that accepts 3,000,000 Spanish-like words, all we have to do is to enter each word from the document and flag all the words that the network rejects. The quality of the spell-checker will depend largely on the coverage and accuracy of the network, i.e. the degree to which the formal language that it accepts corresponds to the real Spanish language. **Xerox** linguists routinely build and manipulate networks of this size, modeling real natural languages like Spanish and French as closely as humanly possible.

1.3.2 Symbol Matching and Analysis

It’s time to introduce some slightly different metaphors for the word-entering and accepting process, those of ANALYSIS (also called LOOKUP) and SYMBOL MATCHING. When we enter a word (as a string of symbols) into a network, to see if it is contained in the language of the network, we often talk of LOOKING UP the word. From this point of view, the network is a kind of dictionary. The analysis (lookup) will be successful if and only if the word is in the language of the network. Such analysis, viewed as a process, involves matching the symbols of the input word against the symbols on the arcs of a path through the network. Let’s return to our 3-word language and analyze (look up) the string “mesa” as shown in Figure 1.11.

The analysis algorithm starts at the start state of the network and at the begin-

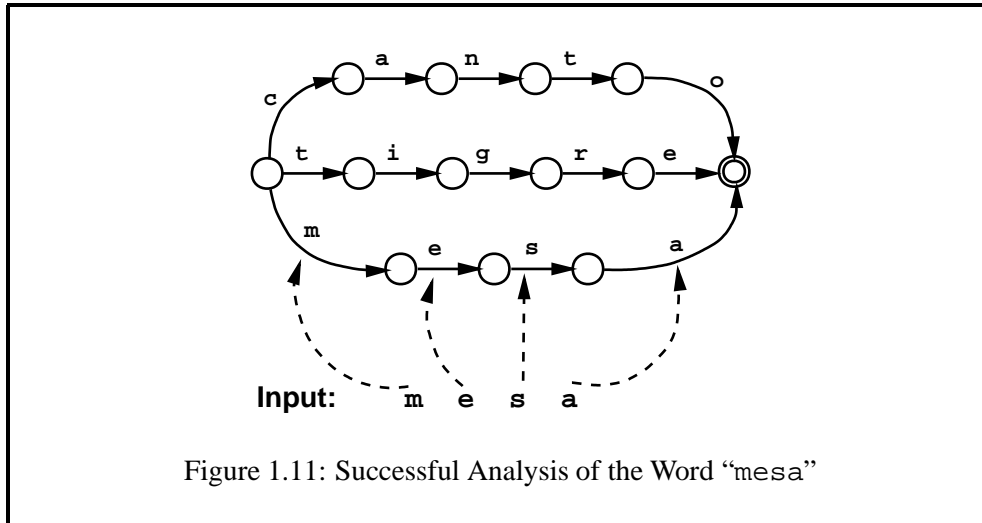


Figure 1.11: Successful Analysis of the Word “mesa”

ning of the word to be looked up. The first symbol of the input word is **m**, so the algorithm looks for an arc labeled **m** that leaves the start state. There is such an arc, so the network machine moves or TRANSITIONS to the new state at the end of the **m** arc, and the **m** in the input string is CONSUMED. Then, from the new state, the algorithm looks for an arc labeled **e**, the next symbol in the input string; finding it, the network transitions to the new state at the end of the **e** arc and consumes the **e** in the input string. Progressing similarly, this analysis will succeed, reaching a final state and, simultaneously, consuming all the input symbols. The word “mesa” is in the language of this network, and any attempts to look up “canto” and “tigre” will also succeed. Analysis of other strings will fail for various reasons, e.g.

- Analysis of “libro” will fail immediately because there is no **l** arc leading from the start state.
- Analysis of “tigra” will fail, getting as far as “tigr” before failing to find a transition to match the input symbol **a**.
- Analysis of “cant” will fail, even though all the input is consumed, because the network will be left in a non-final state.
- Analysis of “mesas” will fail, even though the network reaches a final state, because the final **s** of the input is unconsumed (left over).

It is important to understand that the analysis (lookup) algorithm knows nothing about Spanish or any other language; it just matches input symbols against a network and either succeeds or fails. So in fact the analysis algorithm is language-independent, and the very same algorithm is used to perform lookup with networks that model German, French, English, Portuguese, Arabic, etc.

1.3.3 Getting More Back from Analysis: Transducers

So far the analysis of words in a network has simply yielded one of two responses, either an ACCEPT, indicating that the word is in the language of the network, or a REJECT, indicating that the word is not in the language. While this can be valuable, as for instance in spell-checking, networks are capable of storing and returning much more interesting information. The first step in understanding this capability is to imagine our three-word network with a *pair* of labels on each arc rather than just a single label. Such a two-level network is shown in Figure 1.12.

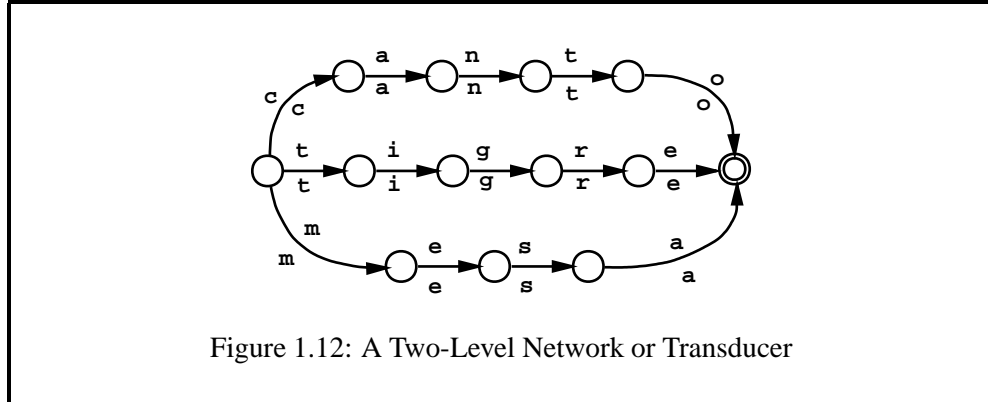


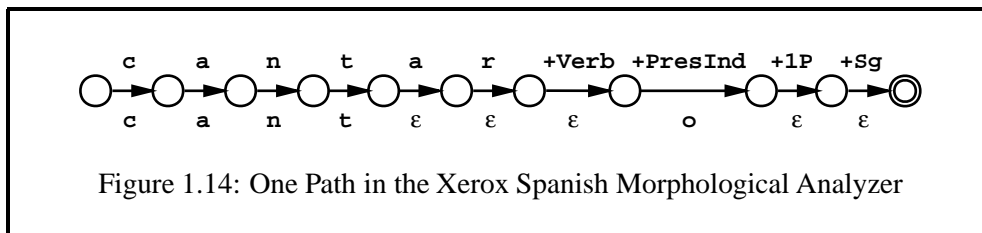
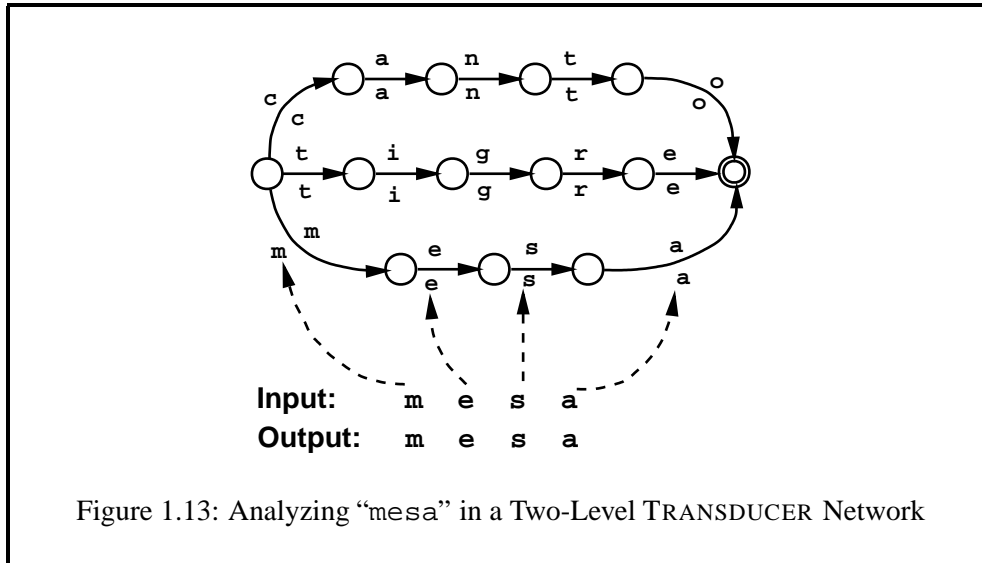
Figure 1.12: A Two-Level Network or Transducer

The analysis process is now modified slightly:

- Start at the start state.
- Match the input symbols of a string against the *lower-side* symbols on the arcs, consuming input symbols and finding a path to a final state.
- If successful, return the string of *upper-side* symbols on the path as the result.
- If the analysis is not successful, return nothing.

If we analyze the word “*mesa*”, the successful result will now be an output string “*mesa*” as shown in Figure 1.13. The dotted arrows in the figure indicate where input symbols are matched along the lower side of a path through the network from the start state to the final state. The output is the string of symbols on the upper side of this successful path. Granted, this output is not especially interesting, being exactly the same as the input, but in fact the upper-side symbols in a network do not need to be the same as the lower-side symbols.

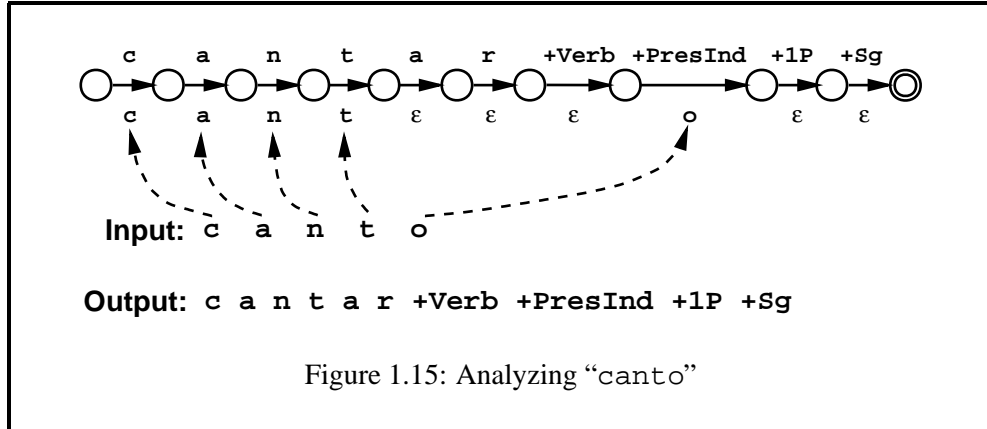
The **Xerox** Spanish Morphological Analyzer network includes over 3,000,000 paths, from the start state to a final state, that look like the path in Figure 1.14. When you analyze the word “*canto*”, as in Figure 1.15, one of the solutions returned is the string “*cantar+Verb+PresInd+1P+Sg*”, which is intended to be read as follows:



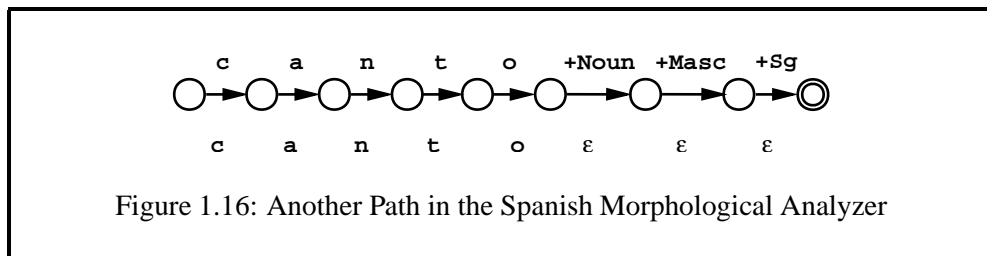
1. The traditional baseform is *cantar* (“to sing”).
2. The word is a verb.
3. The verb is conjugated in the present-indicative (Spanish combines the tense and mood into a single inflection paradigm).
4. First person.
5. Singular.

Thus the process of analysis identifies “canto” as the Spanish equivalent of the English “I sing”, and yet it’s all done via the simple language-independent process of matching symbols from the input string against paths of symbols in the network. The actual analysis step is trivial, language independent and very fast when performed by computers. The hard part is defining and building the network itself.

Notice the new MULTICHA RACTER SYMBOLS labeling some of the arcs in Figure 1.14: +Verb, +PresInd, +1P and +Sg are in fact single symbols, with multicharacter print names, that were chosen and defined by the linguists who built the system. The order of these symbol TAGS, and the choice of the infinitive as the baseform, were also determined by the linguists. Another special symbol in the



network of Figure 1.14 is the EPSILON symbol (ϵ), which fills in the gaps when the upper-level string of symbols and the lower-level string of symbols are not of the same length. During analysis, EPSILON ARCS on the bottom side are traversed freely without consuming any input symbols. You will also see the symbol **0** (zero) used for the epsilon in **Xerox** notations and compilers because the epsilon is not available on standard ASCII keyboards.



Another of the three million paths through the Spanish network looks like Figure 1.16. When you look up “canto”, the analysis algorithm detects the multiple possibilities and automatically BACKTRACKS to find and return a second solution as well: “canto+Noun+Masc+Sg”. Again, this solution, often called a LEXICAL STRING, is just another string of symbols, where +Noun and +Masc are more multicharacter-symbol tags defined by the linguists who created the system. The Spanish noun *canto* means “song”.

Two-sided networks, like the Spanish Morphological Analyzer, are called LEXICAL TRANSDUCERS. In more common discourse, a transducer is a device that converts energy from one form to another; thus a microphone is a transducer that converts physical vibrations of air into analogous electrical signals. Our finite-state transducers convert one string of symbols into another string or strings of symbols. When the transducer is constructed so that the lower-side language consists of valid written words in Spanish, and the upper-side language consists of strings showing baseforms and tags, and when the strings are properly related, the result is an extremely valuable system that returns the morphological analyses of each word that

is analyzed.

1.3.4 Generation

The opposite of analysis is GENERATION, and in fact we use the exact same Spanish network, applying it *backwards*, to generate surface strings from lexical strings. Assume that we want to generate the first-person plural, present-indicative form of the Spanish verb *cantar*. It happens that the Spanish network also contains the path shown in Figure 1.17.

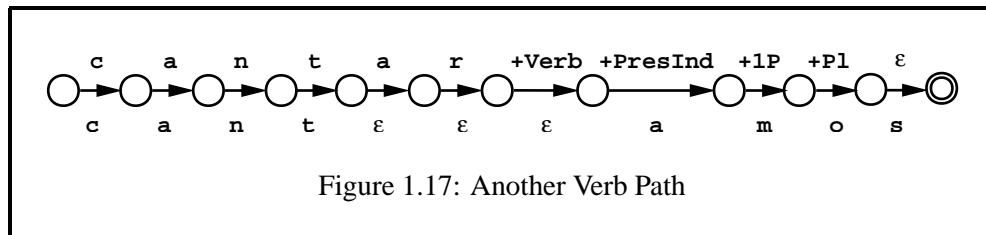


Figure 1.17: Another Verb Path

The process of generation (sometimes called LOOKDOWN) is just the inverse of analysis (also called LOOKUP). Let us assume that the input string, the string we want to put through the generation process, is “cantar+Verb+PresInd+1P+Pl”, i.e. the first-person plural, present-indicative form of the verb *cantar*.

- Start at the start state and at the beginning of the input string.
- Match the input symbols of the input string one by one with the *upper-side* symbols on the arcs, consuming input symbols and finding a path from the start state to a final state.
- If successful, return the string of *lower-side* symbols on the path as the result.
- If generation is not successful, return nothing.

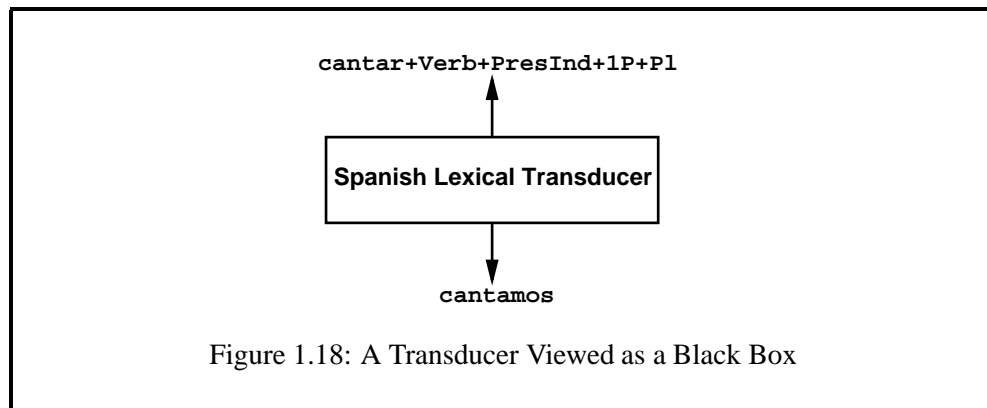
The output of the generation in this case will be the surface string “cantamos” (“we sing”). The epsilon symbols in the network represent the empty string and are ignored in the output. One can, of course, turn back around and enter “cantamos” for analysis and get back the corresponding lexical string. Finite-state transducers are inherently bidirectional.

In a more formal sense, the set of lexical strings produced during analysis, or accepted during generation, constitutes the lexical language (or upper-side language) of the transducer; similarly, the set of surface strings constitutes the surface language (or lower-side language). The transducer MAPS between strings of the upper-side language and strings of the lower-side language.

1.3.5 Further Finite-State Applications

A morphological analyzer/generator, like the Spanish Lexical Transducer discussed above, is often the first goal when a linguist applies finite-state tools and techniques to a new language. A lexical transducer does morphological analysis or generation, depending on which way it is applied to the input, and it is often a vital component of larger systems, such as syntactic parsers, or a crucial starting point for making derivative systems like spelling checkers or part-of-speech taggers.

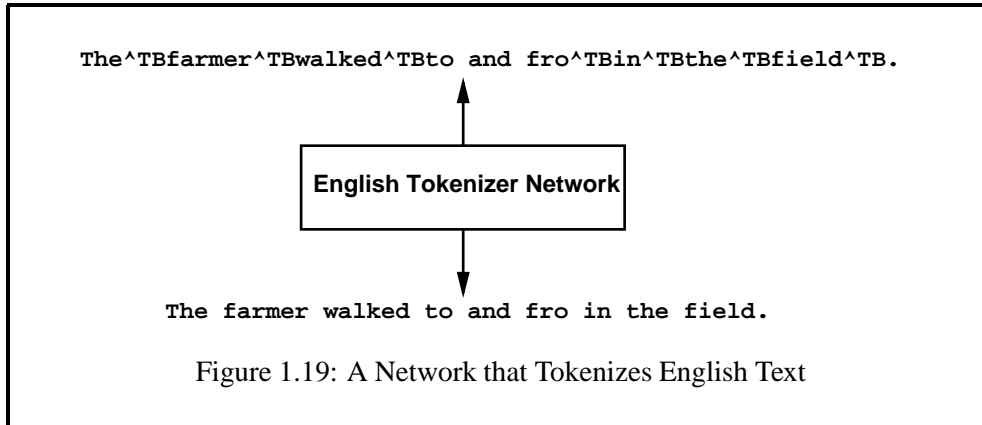
Only the smallest example networks can be diagrammed, so we typically view a network as a black-box component, as in Figure 1.18, keeping in mind that it consists entirely of state-and-arc paths as in the examples shown above, though there may well be millions of such paths, or even an infinite number of them, in a full-scale network. However large the lexical transducer may become, analysis (lookup) and generation (lookdown) are performed by the same language-independent symbol-matching techniques.



Besides morphological analysis and generation, finite-state networks can do many other jobs for us, including TOKENIZATION, which is the dividing up of a running text into individual terms or TOKENS. Tokens usually, but not always, correspond to our everyday notion of words, but there are complicating exceptions, including contractions like *couldn't*, joined words like *cannot*, and indivisible multiword tokens like *to and fro* that often require special attention.

It is possible and often quite useful to define finite-state networks that tokenize strings of written natural language. Such an English tokenizer might be defined so that the lower-side language consists of strings that look like English text. When such surface strings are looked up, the output string is the input string plus defined multicharacter symbols, e.g. **^TB**, for “Token Boundary”, inserted between tokens as shown in Figure 1.19. By convention, feature-like multicharacter symbols in **Xerox** finite-state systems are usually spelled with an initial circumflex (^) character.

Other common applications of finite-state techniques include part-of-speech GUESSERS, which are typically defined to guess categories of words based on

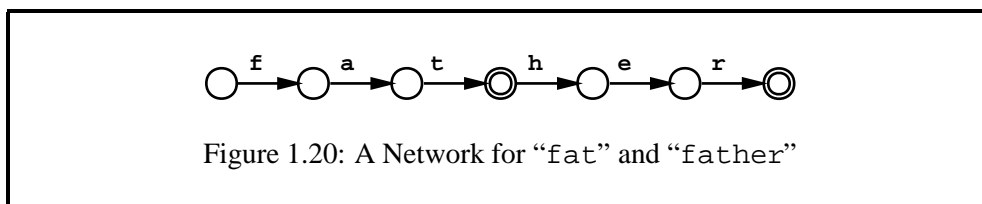


characteristic affixes, and systems that perform phonological/orthographical ALTERNATIONS, based on rule types that have been used by linguists for centuries. More advanced and experimental applications include shallow syntactic parsing or “chunking”, certain kinds of counting, perfect hashing, etc. Suffice it to say that while systems of finite-state power cannot do everything necessary in linguistics, they can do a great deal; and more applications are being found all the time.

1.4 Sharing Structure

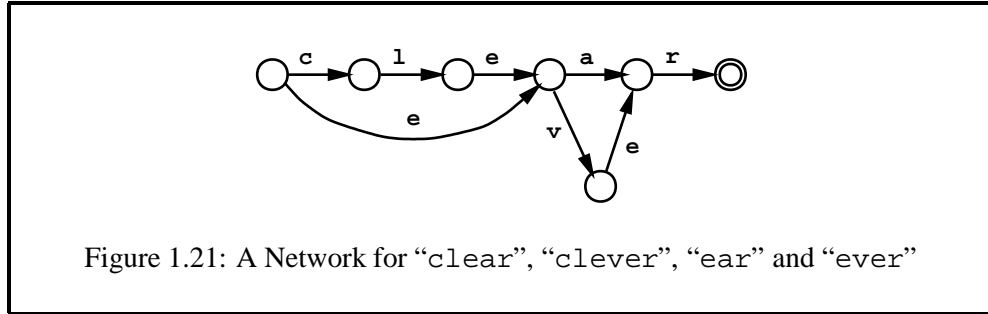
All of the networks we have seen in the previous section have a single initial state and a single final state. Every word or pair of words in these examples has its own unique path sharing only the common initial and final state.

In fact a network can have any number of final states, and an arc may be part of several distinct paths. Figure 1.20 shows a simple example. This network recognizes two words, “fat” and “father”. Because the words begin with the same three letters, the first three arcs of the network can be shared. The path for “father” goes through the final state terminating “fat” and continues to the second final state.



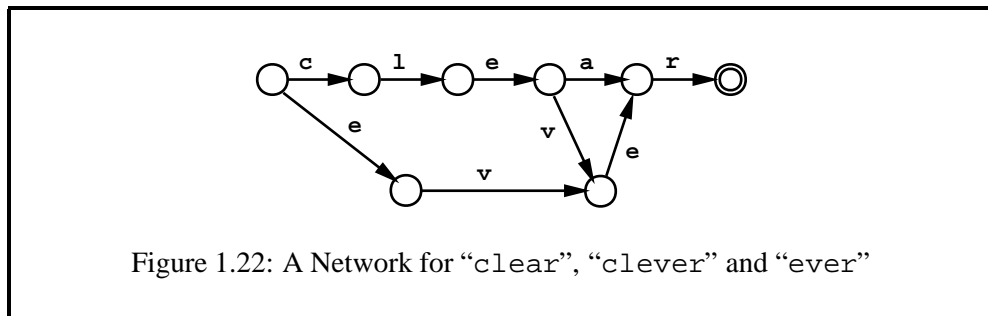
In an optimally configured network, all words that begin or end the same way as some other word share the common beginnings and tails. Figure 1.21 shows an optimally encoded network that accepts the four words “clear”, “clever”, “ear” and “ever”. Because all the words end in “r”, they can all share the last arc leading to the final state. The paths for “clear” and “clever” go together

up to the point where the words diverge and join again for the common ending. Only the first letter of “ear” and “ever” needs its own arc.



The network in Figure 1.21 is MINIMAL in the sense that it is impossible to encode the same four paths using fewer states and arcs. Minimality is an important property for many practical applications. For a typical natural language, a minimal network can encode in the space of a few hundred kilobytes a list of words that takes several megabytes of disk space in text format. The compression ratio is often better than zip or any other general compression can offer.¹ We will discuss minimality and other formal properties of networks in Section 2.5.

Because of structure sharing, removing paths from a network may actually increase its size. For example, if we remove “ear” from the network in 1.21, one new state and arc have to be added, as shown in Figure 1.22.



1.5 Some Background in Set Theory

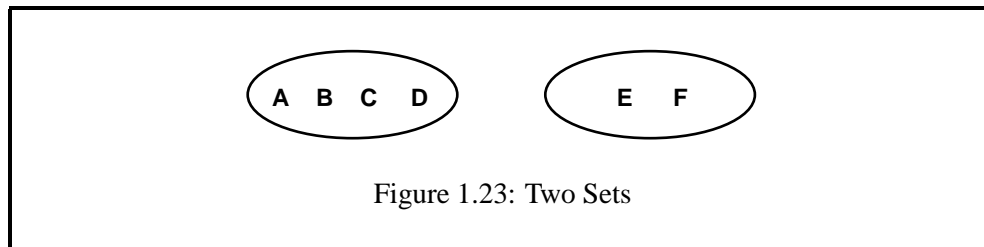
Formal language theory assumes some concepts and terminology from set theory, including membership, union, and intersection. In the same spirit as the rest of this chapter, we will introduce these concepts more intuitively than formally, always trying to illustrate them with linguistically oriented examples showing how sets and operations on them are represented by finite-state networks. Some of these concepts will be examined more formally in the next chapter.

¹For example, the 25K Unix word list /usr/dict/words takes up 206K bytes of disc space as a text file, 81K as a gzipped file, and just 75K as a finite-state network.

1.5.1 Sets

What is a Set?

A SET corresponds to the everyday notions of a collection or group. If you have objects A, B, C, D, E and F, they might be grouped into two sets as shown in Figure 1.23. We use ellipses, arbitrarily, to show set groupings of objects; they should not be confused with the circles that represent states in our network diagrams.



We say that the first set has four MEMBERS or ELEMENTS, A, B, C and D. The second set has two members, E and F. For sets that have a finite number of members, as in Figure 1.23, we can explicitly list or ENUMERATE the sets as $\{A, B, C, D\}$ and $\{E, F\}$. The order in which the elements of a set are listed has no significance. $\{A, B\}$ and $\{B, A\}$ denote the same set. There is only one instance of each element in a set, thus $\{A, A, B\}$ is a redundant listing of the set $\{A, B\}$.

In finite-state linguistics, we often talk about a network encoding or accepting a particular set of strings; and we call this set a LANGUAGE. We usually think of these strings as words, and we use the two terms string and word interchangeably. For example, Figure 1.22 shows a network whose language is a set of three strings. Each word accepted by a network is a MEMBER of the set of words that constitutes the language of the network. For a small language like this one, we can enumerate it easily as $\{\text{“clear”}, \text{“clever”}, \text{“ever”}\}$.

The Empty Set

Some sets are empty, like the set of all living Neandertals or the set of all transparent elephants. Empty sets have no members. The EMPTY LANGUAGE is the language that contains no strings of any kind. This notion of the empty language must be distinguished from the notion of the EMPTY STRING, a string of length zero. Thus if “dog” is a string of length 3, and “tiger” is a string of length 5, “” is a string of length 0 (zero), the empty string.

In finite-state terms the empty set is represented by a network that consists of one *non-final* state (see Figure 1.25).

A network consisting of a single *final* state with no arcs represents the language that contains just the empty string (see Figure 1.26). The empty-string language is not an empty set; it contains one element, which is the empty string.

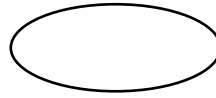


Figure 1.24: The (Empty) Set of All Living Neandertals

Figure 1.25: Network for the Empty Language $\{\}$

Infinite Sets

Many sets, such as the set of all integers, contain an infinite number of elements. Real human languages that allow productive compounding, like German, contain an infinite number of words, at least theoretically. Obviously, it is impossible to enumerate all the members of an infinite set, but as we shall see in the next chapter, we can easily specify many infinite languages, using the finite metalanguage of REGULAR EXPRESSIONS.

Some infinite languages can be represented by a very small finite-state network. For example, the language $\{\epsilon, "a", "aa", "aaa", \dots\}$ that contains all strings of zero or more **a**s is represented by the network in Figure 1.27.

Universal Set

The infinite set that contains all strings of any length, including the empty string, is called the UNIVERSAL LANGUAGE. If the question mark (?) represents any symbol, then the network shown in Figure 1.28 accepts the universal language.

Ordered Set

An ordered set is a set whose members are ordered in some way. We will only be concerned with one very specific type of ordered set. An ORDERED PAIR is a set that has two members: a *first* and a *second* element. To distinguish an ordered pair from an ordinary set we list it in angle brackets. While $\{A, B\}$ and $\{B, A\}$ denote the same set, $\langle A, B \rangle$ and $\langle B, A \rangle$ are distinct ordered pairs. One is the *inverse* of the other.

1.5.2 Relations

A RELATION is a set whose members are ordered pairs. Words like *father*, *husband*, *wife*, *spouse*, etc. denote relations in this technical sense because they involve pairs



Figure 1.26: Network for the Empty-String Language {""}

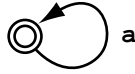


Figure 1.27: A Network for an Infinite Language

of individuals. For example, consider the British royal family tree in Figure 1.29.

A genealogical tree implicitly defines certain relations over a set of people. For example, the *father* relation in Figure 1.29 is the set $\{\langle \text{Philip}, \text{Charles} \rangle, \langle \text{John}, \text{Diana} \rangle, \langle \text{Charles}, \text{William} \rangle\}$. Here we chose the ordering that reflects the word order in the corresponding English sentences: *Philip* is the father of *Charles*, etc. By the same principle, the *wife* relation is the set $\{\langle \text{Elisabeth}, \text{Philip} \rangle, \langle \text{Frances}, \text{John} \rangle, \langle \text{Diana}, \text{Charles} \rangle\}$. Its inverse, $wife^{-1}$, is the *husband* relation: $\{\langle \text{Philip}, \text{Elisabeth} \rangle, \langle \text{John}, \text{Frances} \rangle, \langle \text{Charles}, \text{Diana} \rangle\}$.

In this book we focus on relations that contain pairs of strings, for example,

$$\{ \langle \text{'cantar+Verb+PresInd+1P+Sg'}, \text{'canto'} \rangle, \\ \langle \text{'cantar+Verb+PresInd+1P+Pl'}, \text{'cantamos'} \rangle, \\ \langle \text{'canto+Noun+Masc+Sg'}, \text{'canto'} \rangle \\ \}.$$

The relation may involve some phonological change, the expression of some morphological inflection or derivation, one string being the analysis or translation of the other, etc. A string-to-string relation is a mapping between two languages. One language is the set of strings that appear as the first member of some pair; the other is the set of second member strings. We refer to them as the UPPER LANGUAGE and the LOWER LANGUAGE, respectively. In the case at hand, the upper language is $\{\text{'cantar+Verb+PresInd+1P+Sg'}, \text{'cantar+Verb+PresInd+1P+Pl'}, \text{'canto+Noun+Masc+Sg'}\}$ and the lower one is $\{\text{'canto'}, \text{'cantamos'}\}$.

As we have already seen in Figures 1.15 and 1.16, string-to-string relations can be represented as a finite-state transducer in which every pair in the relation corresponds to a path through a network from the start state to a final state. Each path enumerates the symbols of the two strings that constitute the pair. Each arc is labeled with an upper symbol and a lower symbol, which may be identical.



Figure 1.28: A Network for the Universal Language

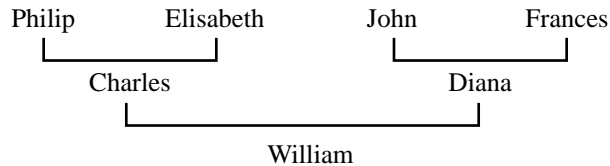


Figure 1.29: A Royal Family Tree

Infinite Relations

A relation may contain an infinite number of pairs. One example of such a relation is the mapping of all lower-case strings to the corresponding upper-case strings. This relation contains an infinity of pairs such as $\langle \text{"dog"}, \text{"DOG"} \rangle$, $\langle \text{"abc"}, \text{"ABC"} \rangle$, $\langle \text{"xyzzzy"}, \text{"XYZZY"} \rangle$, and so on. The upper language is the infinite language of lower-case strings, the lower language contains all the upper-case strings, and the relation itself is a mapping that preserves the word. We denote an ordered pair $\langle a, A \rangle$ of symbols as $a : A$, where a is the upper-side symbol and A is the lower-side symbol. The network for this infinite relation of upper-case and lower-case strings is like the ones in Figures 1.27 and 1.28 except that the arc has the label $a:A$, and there is a similar looping arc for each uppercase:lowercase pair in the alphabet. To save space, we may draw such networks with just one multiply labeled arc, as in Figure 1.30, but in reality each label has its own arc. Using this network, the analysis and generation methods defined in the previous section will convert any string from upper case to lower case or vice versa.

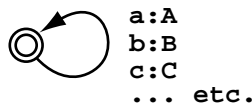


Figure 1.30: The Lowercase/Uppercase Transducer. Each lowercase:UPPERCASE pair of symbols in fact has its own arc.

The path that relates “xyzzzy” to “XYZZY” cycles many times through the

single state of the transducer. Figure 1.31 shows it in a linearized form. The lowercase/uppercase relation may be thought of as the representation of a simple orthographic rule. In fact we view all kinds of string-changing rules in this way, that is, as infinite string-to-string relations. The networks that represent them are generally much more complex than the little transducer shown in Figure 1.30.

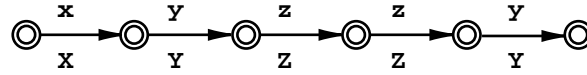


Figure 1.31: A Linearized Path in the Lowercase/Uppercase Transducer

Identity Relation

One technically important but otherwise uninteresting kind of relation is one in which the upper and the lower languages are the same and every string of the language is paired with itself. This is called the IDENTITY RELATION. For example, the network in Figure 1.12 represents the identity relation $\{ \langle \text{“canto”}, \text{“canto”} \rangle, \langle \text{“tigre”}, \text{“tigre”} \rangle, \langle \text{“mesa”}, \text{“mesa”} \rangle \}$ on the language $\{ \text{“canto”}, \text{“tigre”}, \text{“mesa”} \}$ of Figure 1.10.

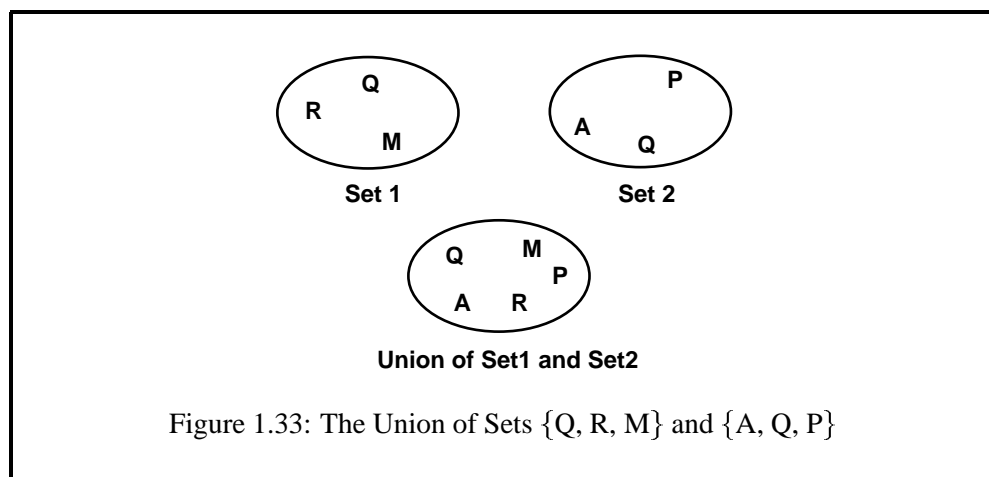
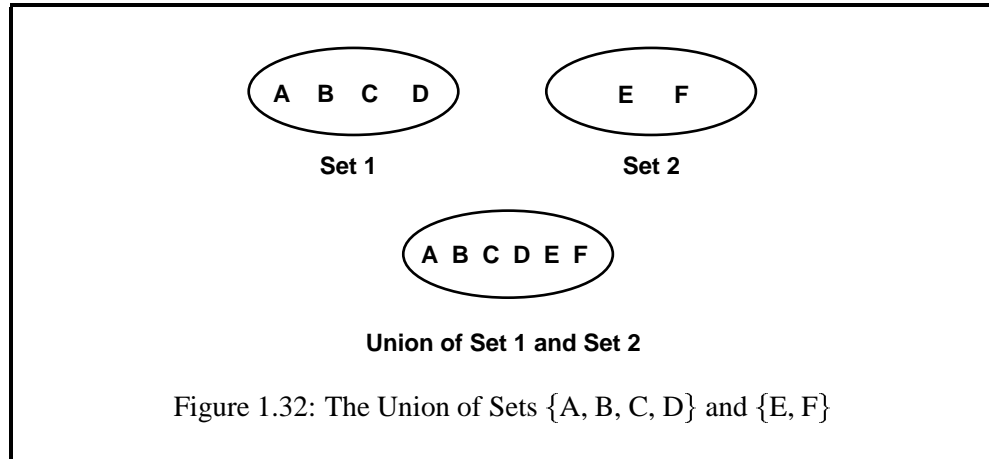
1.5.3 Some Basic Set Operations

Union

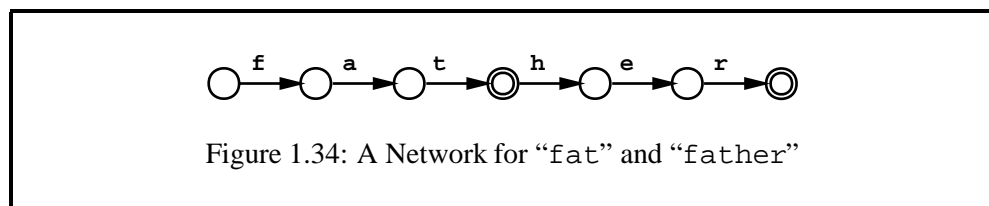
The union of any two sets S_1 and S_2 is another set that contains all the elements in S_1 and all the elements in S_2 . The union of the two sets introduced in Figure 1.23 is the set shown in Figure 1.32. The order of the sets in the union operation makes no difference: the union of S_1 and S_2 is the same set as the union of S_2 and S_1 , just as in addition $2+3$ is the same as $3+2$. Union and addition are COMMUTATIVE operations.

Now consider another pair of sets and their union, as shown in Figure 1.33. We have purposely scattered the members in the set ellipses to emphasize the fact that the set members are not ordered; the set $\{Q, R, M\}$ is the same as $\{Q, M, R\}$, $\{M, R, Q\}$, etc. Note also that the two original sets have three members each, but the union has five rather than six members. In this case, the Q member is common to both the original sets, so it appears only once in the union.

Some of the relations in Figure 1.29 are unions of other relations. For example the *spouse* relation $\{ \langle \text{Elisabeth}, \text{Philip} \rangle, \langle \text{Philip}, \text{Elisabeth} \rangle, \langle \text{Frances}, \text{John} \rangle, \langle \text{John}, \text{Frances} \rangle, \langle \text{Diana}, \text{Charles} \rangle, \langle \text{Charles}, \text{Diana} \rangle \}$ is obviously the union of the *wife* relation with its inverse, the *husband* relation. Similarly, *parent* is the union of the *father* and *mother* relations.



Languages, being sets of strings, can also be unioned. So if we start with the language represented in Figure 1.34, and another language as represented in Figure 1.35, the union of these two languages is the language accepted by the network in Figure 1.36. There is no significant ordering of the arcs leading from a state in a finite-state network, just as there is no significant ordering of members in sets. The network could be represented in many other equivalent ways.



Unioning of languages (or the unioning of networks that accept languages) is valid for both simple one-level languages and the more complex relations, encoded

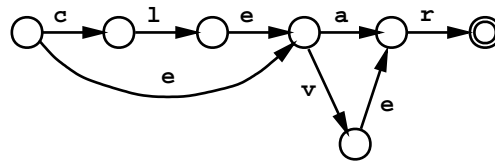


Figure 1.35: A Network for “clear”, “clever”, “ear” and “ever”

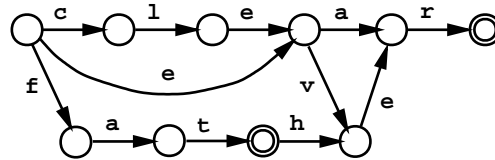


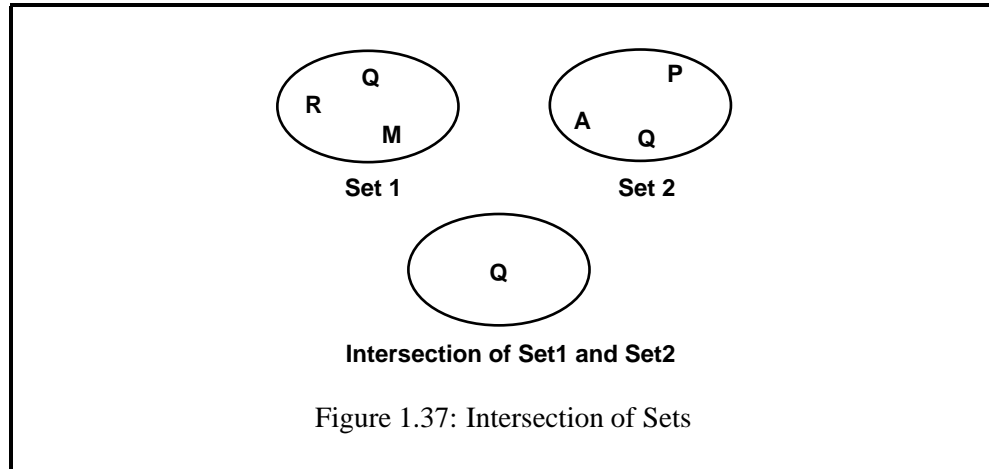
Figure 1.36: The union of { “fat”, “father” } with { “clear”, “clever”, “ear”, “ever” }

as two-level transducers. In finite-state linguistic development, this often makes it possible for one person to work on adjectives, another on nouns, and a third on verbs, producing three different networks that can later simply be unioned together into a single network.

Intersection

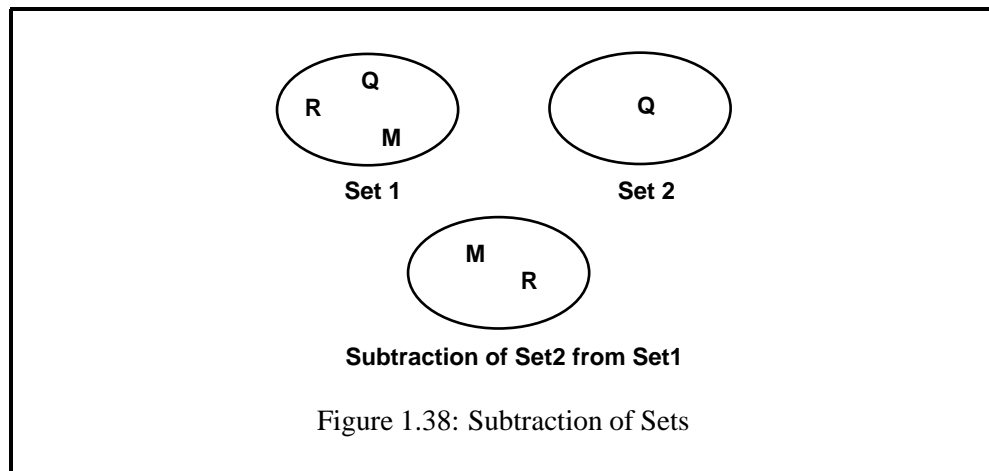
For any two sets S_1 and S_2 , the intersection is the set containing all the members that are common to both sets. The intersection of $\{Q, R, M\}$ and $\{Q, A, P\}$, as shown in Figure 1.37, is the set $\{Q\}$. The intersection of two sets that share no members, such as $\{A, B, Z\}$ and $\{M, N, C\}$, is the empty set. Like union, intersection is a commutative operation: the intersection of S_1 and S_2 is the same as the intersection of S_2 and S_1 .

Simple languages, and networks accepting simple languages, can also be intersected. Thus if one network accepts the simple language {“apple”, “banana”, “pear”}, and another network accepts the simple language {“kumquat”, “pear”, “grape”, “quince”, “banana”}, the intersection of these two simple networks accepts the language {“banana”, “pear”}. As we shall see in coming chapters, only simple one-level languages (modeled with one-level networks) can be intersected; relations (modeled with two-level transducers) can be intersected only in special cases.



Subtraction

The subtraction of sets is also easy to conceptualize and diagram. In the simplest case, shown in Figure 1.38, set $\{Q, M, R\}$ minus set $\{Q\}$ leaves set $\{M, R\}$.



Simple languages, and networks accepting simple languages, can also be subtracted. Thus if language L_1 is {"apple", "banana", "pear"}, and language L_2 is {"kumquat", "pear", "grape", "quince", "banana"}, the subtraction of L_2 from L_1 is {"apple"}. It is important to understand that subtraction, unlike intersection and union, is not a commutative operation: L_2 minus L_1 can be completely different from L_1 minus L_2 , just as in numeric subtraction $3-2$ is not the same as $2-3$. As we shall see in coming chapters, only simple languages (modeled with one-level networks) can be subtracted; relations (modeled with two-level transducers) can be subtracted only in special cases.

Concatenation

The operation of **CONCATENATION** is an easy one to grasp through linguistic examples; in the most intuitive case, consider the network that corresponds to the language that contains a single string “work”, which happens to look like a verb in English (Figure 1.39). Now imagine another language of three words, “s”, “ed” and “ing”, which happen to look like verbal suffixes of English. The network for this language will look like Figure 1.40. If we concatenate the verbal-suffix language on the end of the “work” language, we get the new language recognized by the network in Figure 1.41, which contains the three strings “works”, “worked” and “working”.

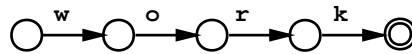


Figure 1.39: Network for the Language {“work”}

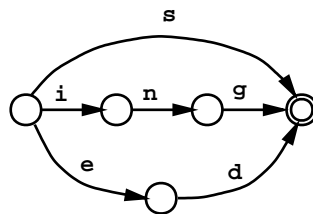


Figure 1.40: Network for the Language {“s”, “ed”, “ing”}

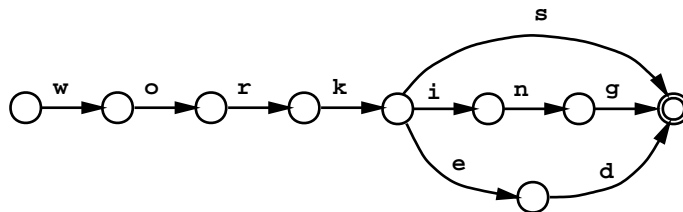


Figure 1.41: Language {“work”} Concatenated with {“s”, “ed”, “ing”}

We can also see the “work” string itself as a concatenation of **w**, **o**, **r** and **k**; and the “ed” string is a concatenation of **e** and **d**, and similarly for “ing”. In any

finite-state language, the strings are simply concatenations of symbols available from the alphabet of the language.

As the example would suggest, concatenation is the main mechanism for building complex words in a finite-state grammar, and **lexc** (Chapter 4) is the primary **Xerox** language to use for specifying the concatenations that construct entire natural-language words out of the parts known as MORPHEMES. The study and modeling of legal word formation is called MORPHOTACTICS or, in some traditions, MORPHOSYNTAX.

But there is another problem yet to be solved. The concatenation of verb base-forms like *work* with *s*, *ed* and *ing* is a productive morphotactic process in English, also applying to verbs like *talk*, *kill*, *kiss* and *print*. However, when we try to extend this simple concatenation of verbal endings “s”, “ed” and “ing” to other English verbs like *try*, *plot* and *wiggle*, we immediately find problems.

*trys	*tryed	trying
plots	*ploted	*ploting
wiggles	*wiggled	*wiggling

We use the prefixed asterisk (*) to mark those words that aren't spelled correctly. The correct forms, paired vertically with the incorrect ones, are shown below. Spaces are used to line up the two strings letter-by-letter as closely as possible.

try s	tryed	plot ed	plot ing	wiggled	wiggling
tries	tried	plotted	plotting	wiggled	wiggling

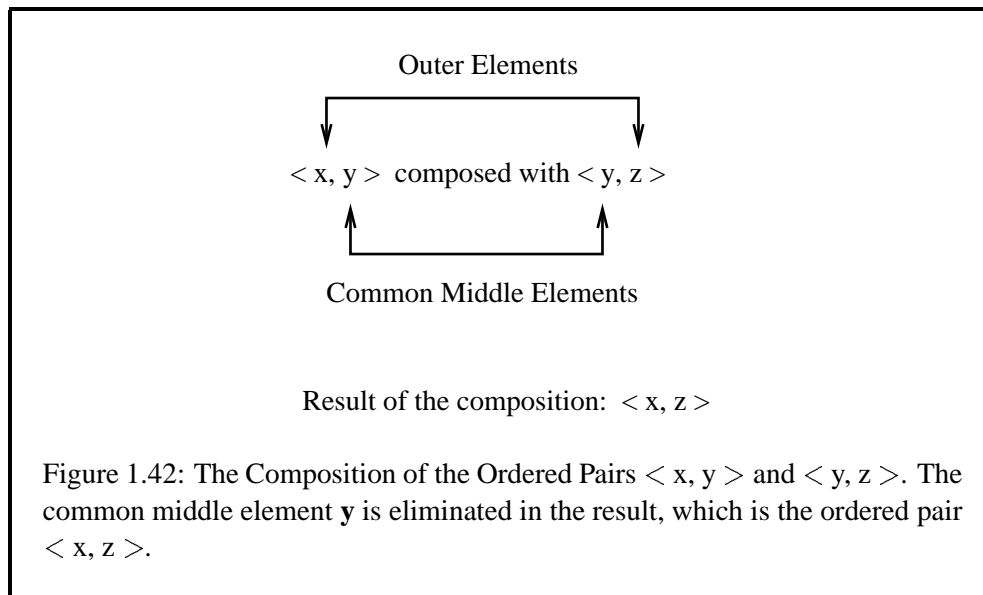
There are many ways to characterize the differences or ALTERNATIONS between the two LEVELS of strings, but we note in rough terms that sometimes letters need to be changed, such as a **y** becoming **i** in *tried*, or added, as in **ploting* becoming *plotting*, or deleted, as in **wiggling* becoming *wiggling*. These alternations from one level to the other can be described by finite-state RULES written in the form of **Xerox** replace rules or perhaps **twolc**, an alternative formalism.

Rules that describe phonological and orthographical alternations can be viewed as infinite string-to-string relations just like the simple lowercase/uppercase rule in Figure 1.30. They naturally tend to be more complicated to express, but there is no fundamental difference. To discuss this idea in more depth we first have to introduce the last operation in this section.

Composition

The operation of COMPOSITION is a little more complex than the ones we have seen so far. Composition is an operation on two relations, and the result is a new relation. If one relation contains the ordered pair $\langle x, y \rangle$ and the other relation contains the ordered pair $\langle y, z \rangle$, the relation resulting from composing $\langle x, y \rangle$

and $\langle y, z \rangle$, in that order, will contain the ordered pair $\langle x, z \rangle$. Composition brings together the “outside” components of the two pairs and eliminates the common one in the middle, as shown in Figure 1.42.

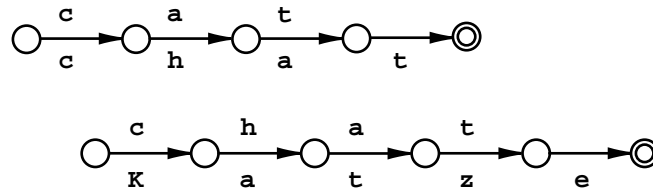
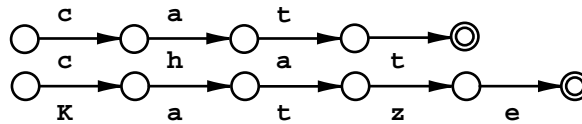


Looking back at the royal family tree in Figure 1.29, we see that some new relations can be defined in terms of the others by way of composition. For example the *father-in-law* relation, $\{\langle \text{Philip}, \text{Diana} \rangle, \langle \text{John}, \text{Charles} \rangle\}$, is obviously the composition of *father*, $\{\langle \text{Philip}, \text{Charles} \rangle, \langle \text{John}, \text{Diana} \rangle, \langle \text{Charles}, \text{William} \rangle\}$, and *spouse*, $\{\langle \text{Elisabeth}, \text{Philip} \rangle, \langle \text{Philip}, \text{Elisabeth} \rangle, \langle \text{Frances}, \text{John} \rangle, \langle \text{John}, \text{Frances} \rangle, \langle \text{Diana}, \text{Charles} \rangle, \langle \text{Charles}, \text{Diana} \rangle\}$. The *grandfather* relation, $\{\langle \text{Philip}, \text{William} \rangle, \langle \text{John}, \text{William} \rangle\}$, is the composition of *father* and *parent*.

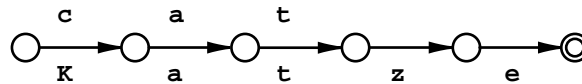
To illustrate composition with a very simple string example, consider the relation $\{\langle \text{“cat”}, \text{“chat”} \rangle\}$ and the relation $\{\langle \text{“chat”}, \text{“Katze”} \rangle\}$. The result of the composition is $\{\langle \text{“cat”}, \text{“Katze”} \rangle\}$, where “chat” and “Katze” happen to be the French and German words for “cat”, respectively.

Without going into too many technical details, it is useful to have a general idea of how composition is carried out when such string-to-string relations are represented by finite-state networks. The transducers representing the two relations are shown in Figure 1.43. Unlabeled arcs in these diagrams should be understood to be epsilon arcs.

We can think of composition of transducers as a two-step procedure. First the paths of the two networks that have a matching string in the middle are merged, as shown in Figure 1.44. The method of finding the matching path is essentially the same as described in Sections 1.3.3 and 1.3.4 on Analysis and Generation except that here we are tracking a string in two networks simultaneously. We then eliminate the string “chat” in the middle, giving us a transducer that directly maps

Figure 1.43: The \langle “cat”, “chat” \rangle and \langle “chat”, “Katze” \rangle TransducersFigure 1.44: Merging \langle “cat”, “chat” \rangle with \langle “chat”, “Katze” \rangle

“cat” to “Katze”, and vice versa, as shown in Figure 1.45. If we were to compose in the opposite order, that is $\{\langle$ “chat”, “Katze” \rangle with $\{\langle$ “cat”, “chat” \rangle , the result would be the empty relation because there is no matching string in the middle.

Figure 1.45: The Composition of \langle “cat”, “chat” \rangle with \langle “chat”, “Katze” \rangle . The common string “chat” in the middle disappears in the composition.

Let us move on to a less trivial example of network composition, a case in which one of the relations is infinite. This time we will compose the $\{\langle$ “cat”, “chat” $\rangle\}$ network with the lowercase/uppercase transducer in Figure 1.30. For convenience we draw the network again in Figure 1.46, making more of its numerous looping arcs explicit. The first step in the composition algorithm gives us the structure in Figure 1.47. As in Figure 1.31, we linearize the looping \langle “chat”, “CHAT” \rangle path. The second step eliminates the middle-level “chat” string, giving us the network in Figure 1.48.

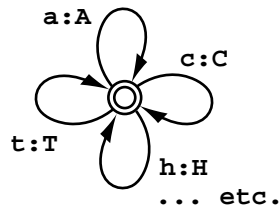


Figure 1.46: The Lowercase/Uppercase Transducer

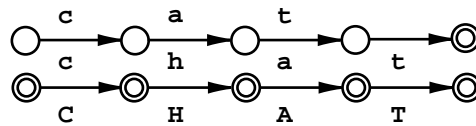


Figure 1.47: Merging <“cat”, “chat”> with the Lowercase/Uppercase Transducer

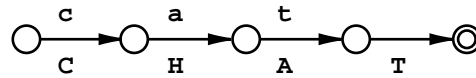


Figure 1.48: Composition of <“cat”, “chat”> with the lowercase/uppercase network in Figure 1.46

Projection

It is often useful to extract one side from a given relation. This simple operation on relations is called **PROJECTION**. From the lowercase/uppercase relation represented in Figure 1.46, we can derive a network for all lower case strings or for all upper case strings by just systematically relabeling the arcs in the desired way. The **UPPER** projection is obtained by replacing $a:A$ by a , $c:C$ by c , and so on. The **LOWER** projection is obtained by replacing each pair label with the lower (= second) member of the pair.

1.6 Composition and Rule Application

If we think of the lowercase/uppercase transducer as representing an orthographical rule, then the composition of the two networks in Figure 1.48 is in effect the

APPLICATION of this rule on the lower side of the $\langle \text{“cat”}, \text{“chat”} \rangle$ pair. When a string-changing rule is represented by a finite-state transducer, *composition* and *rule application* become essentially the same notion. There are only a couple of minor points of difference.

The first difference is that composition is technically defined as an operation on two relations, whereas we generally think of linguistic rules, such as the change of **y** to **i** in some context, as being applied to individual underlying strings like “**t**rys”. But it is easy to bridge the gap between the traditional intuition and the formalism. The application of the lowercase/uppercase rule to an individual string, such as “chat”, can be seen as the composition of the corresponding identity relation $\{ \langle \text{“chat”}, \text{“chat”} \rangle \}$ with the lowercase/uppercase relation, which gives us the result in Figure 1.49.

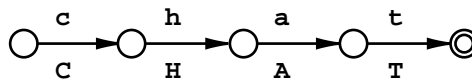


Figure 1.49: An application of the upper-casing rule

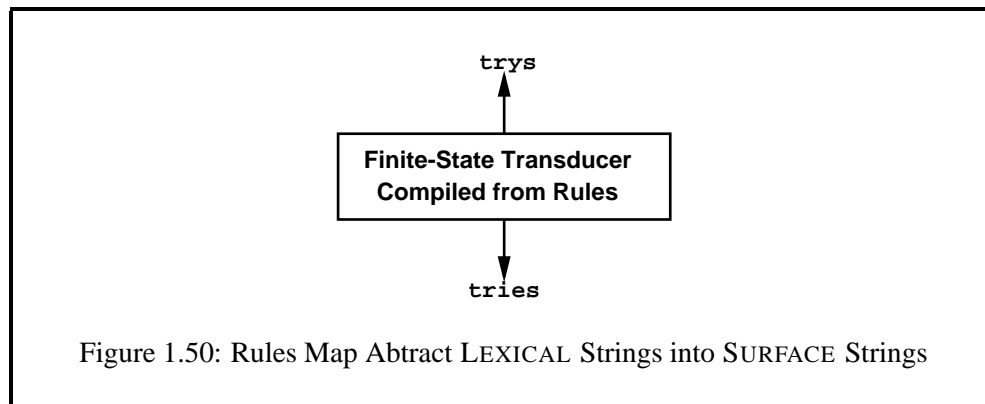
We must of course be aware of the distinction between a *string*, the *language* consisting of the string, and the *identity relation* on that language; but in fact we can (and do) use the same network to represent all three notions at the same time.

The second unimportant difference is that linguists tend to think of rule application as a process that yields a string as its output, and not as something that yields an input/output relation as in Figure 1.49. But it is of course a trivial matter to obtain the output language from the input/output relation by projection, i.e. by extracting the lower-side language {“CHAT”}.

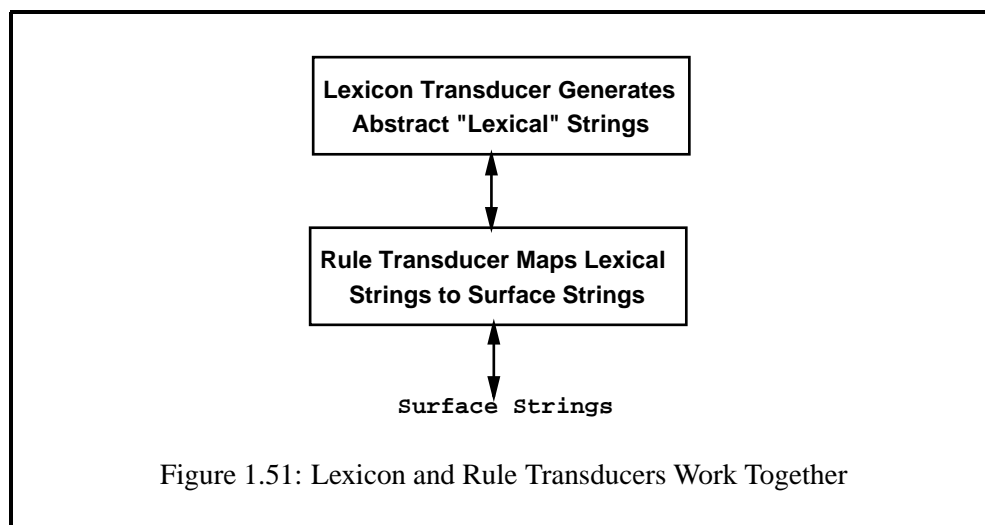
Once we think of rule application as composition, we immediately see that a rule can be applied to several words, or even a whole language of words, at the same time if the words are represented as a finite-state network. Two rule transducers can also be composed with one another to yield a single transducer that gives the same result as the successive application of the original rules, just as our $\{ \langle \text{“cat”}, \text{“Katze”} \rangle \}$ transducer translates directly from English “cat” to German “Katze”, eliminating the intermediate French translation “chat”.

1.7 Lexicon and Rules

This is not yet the time to go into the details of real linguistic rules. The transducers that represent them are often vastly more complex than the one-state network in our previous example. For the time being, let us ignore the internal details and think of a transducer that, for example, maps the incorrect string “**t**rys” to the correct string “**t**ries” as shown in the black box in Figure 1.50.

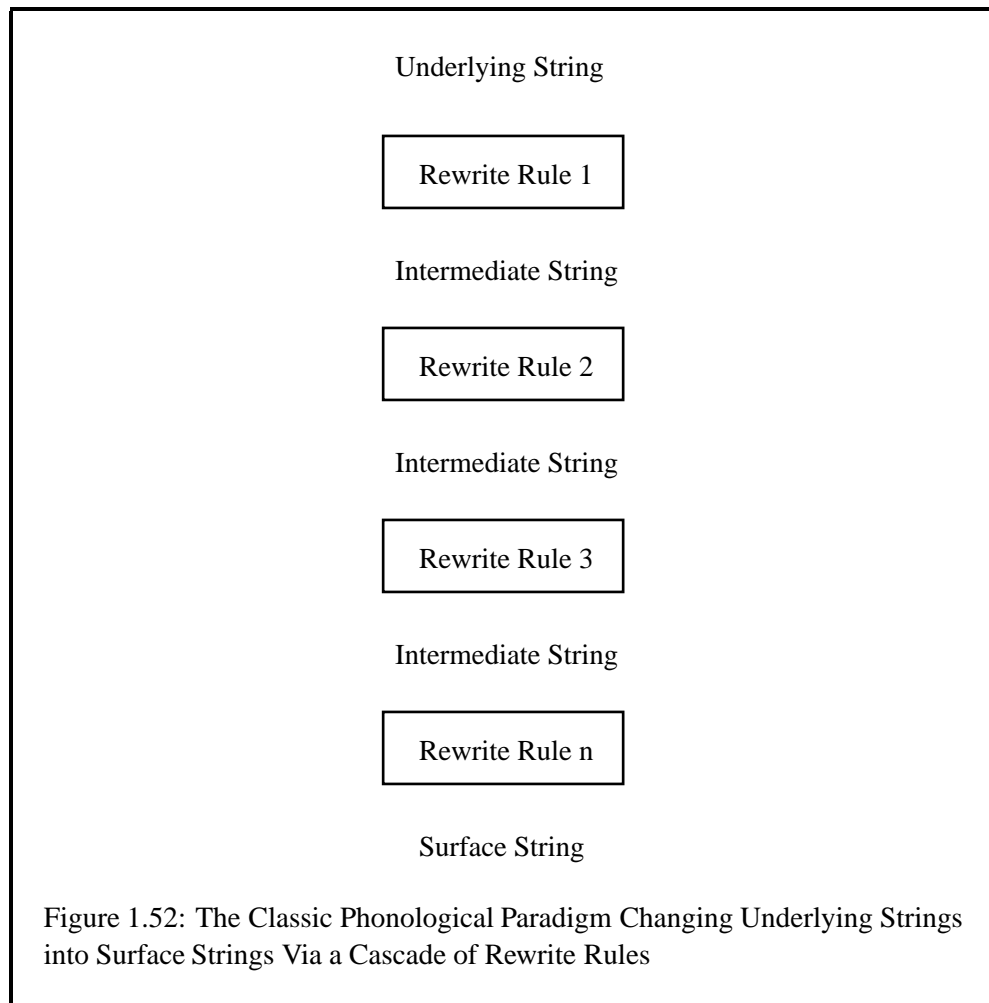


Typically, the lexicon is a finite-state transducer, defined with a **lexc** program, that generates morphotactically well-formed but still rather abstract strings. Such strings are sometimes called “underlying”, morphophonemic or, in the finite-state tradition, LEXICAL strings. Rules then map the abstract lexical strings into properly spelled SURFACE strings. This scheme supposes at least two separately defined finite-state networks as in Figure 1.51.

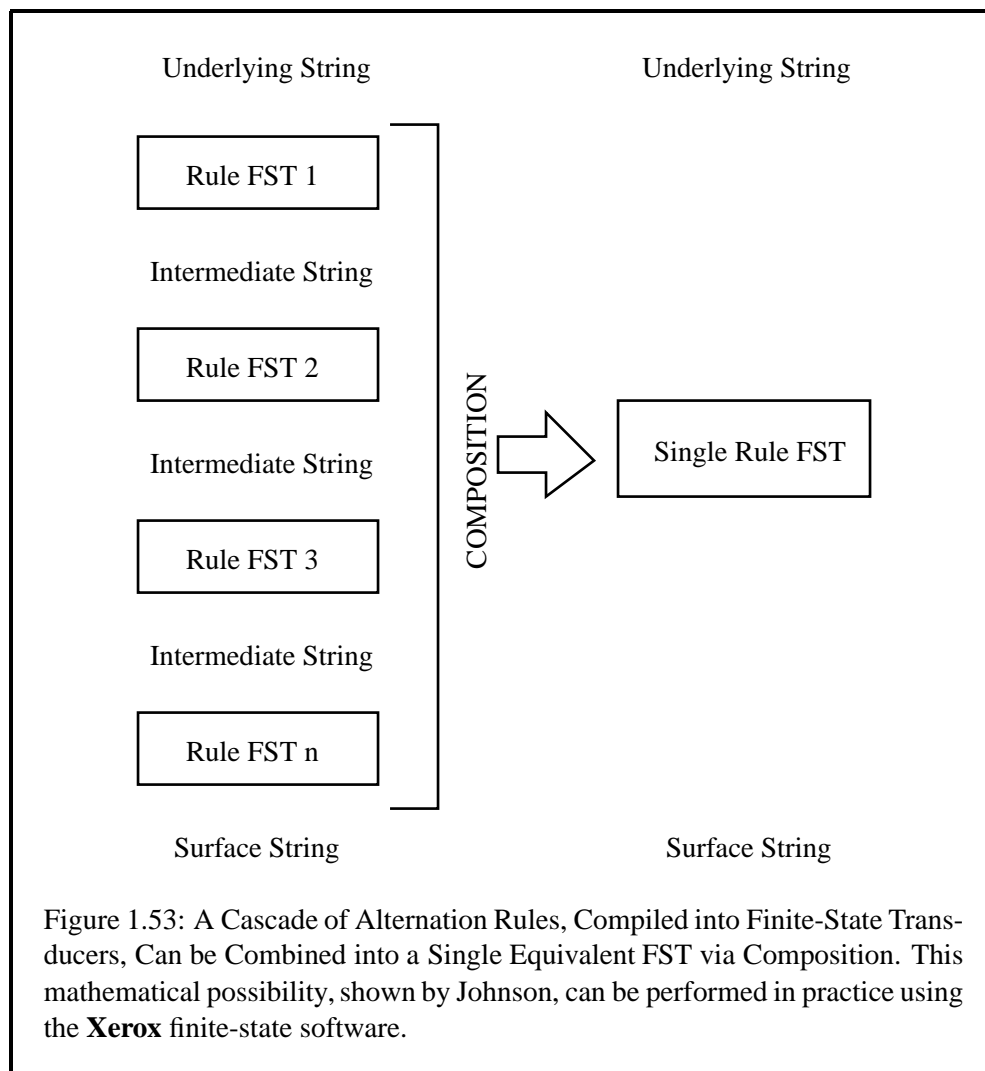


From the perspective of a linguist, this idea appears too simple. The mapping between lexical strings and surface strings can be very complex. It is certainly not possible to describe dozens of morphological alternations by a single rule. Orthographical and phonological rules are typically about the realization of just one symbol, for example, *y/ie* alternation in English, or about a particular class of symbols such as word-final devoicing in German. The linguistic description of the phonological and morphological alternations of natural languages typically requires dozens of rules. And there are other possible complications: some rules may have exceptions, and some rules may have priority over other rules.

For centuries, going back to the famous Sanskrit grammarian Panini who lived around 500 BC, linguists have described phonological alternations and historical sound changes in terms of UNDERLYING strings and REWRITE RULES that are applied in a given order with the output of one rule “feeding” the next (see Figure 1.52). The output of a rewrite rule is an INTERMEDIATE string, with the output of the final rule being a SURFACE string. Most modern phonologists model the underlying, intermediate and surface strings as sequences of phonemes, which in turn are bundles of features; the alternation rules can match and modify individual features within phonemes. In computational linguistics, we usually deal with orthographical symbols rather than phonetic feature bundles, but the principle is the same.

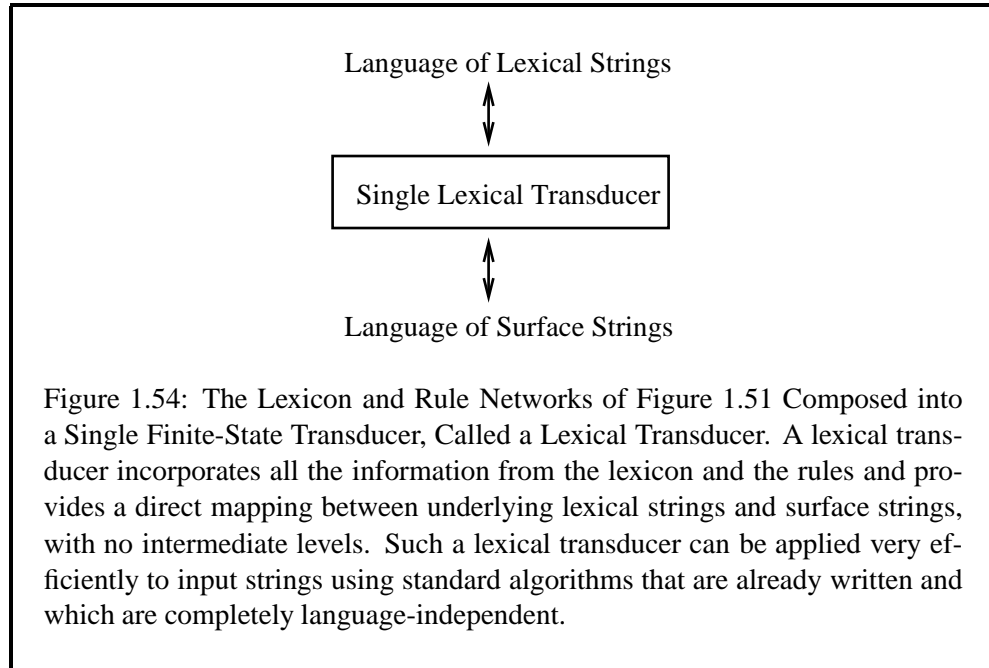


It was C. Douglas Johnson (Johnson, 1972) who apparently first realized that the phonological rewrite rules used by linguists could theoretically be modeled as finite-state transducers (FSTs). Furthermore, individual rule transducers can be ar-



ranged in a cascade in which the output of one transducer serves as the input for the next one. Johnson also observed that for any cascade of transducers that performs a particular mapping, there exists in theory a single transducer that performs the same mapping in a single step. This single transducer can be computed by successively composing the transducers with one another until just one remains. In other words, even if the mapping between lexical strings and surface strings is very complex and cannot be described by just one rule, we can in principle always produce a single rule transducer that does the equivalent mapping, as Figure 1.53 illustrates.

In finite-state state systems we can in fact go one step further. Finite-state rules apply to entire lexicons, which are also encoded as finite-state transducers, so the separate lexicon network and rule network shown in Figure 1.51 can be



composed together into a single all-inclusive network, a LEXICAL TRANSDUCER, as shown in Figure 1.54. Such a lexicon transducer incorporates all the lexicon and rule information in a single data structure, mapping directly between a language of underlying or “lexical” strings and a language of surface strings.

Another way arrive at a single lexical transducer is to use the two-level rule formalism invented by Kimmo Koskenniemi (Koskenniemi, 1983). Like classical rewrite rules, two-level rules can also be modeled as finite-state transducers (Karttunen et al., 1987), and a system of two-level rules can be composed with the lexicon to produce a single transducer (Karttunen et al., 1992).

1.8 Where We are Heading

The **Xerox** compilers **lexc** and **twolc**, and the interface **xfst**, are the tools that linguists use to create finite-state networks, including the lexical transducers that do morphological analysis and generation. You will learn how to use **lexc** to build finite-state lexicons, adding baseforms and describing the morphotactic possibilities for each word. In fact **lexc** is just one of several ways to specify finite-state transducers, but it is especially designed to facilitate the work of the lexicographer. You will also learn to use replace rules to formalize the alternation rules traditionally used in phonology. You may also learn the formalism of **twolc** rules, Koskenniemi’s two-level alternative to replace rules for which **Xerox** also offers a compiler. And you will learn to use the full arsenal of the Finite-State Calculus, available through the **xfst** interface, to combine, customize, test and constrain your

systems to fit your needs.

Because all the lexicons and rules defined by the linguist are compiled into finite-state networks, they can (respecting some formal mathematical restrictions) be combined together using any operations that are valid for finite-state networks, including union, concatenation, subtraction, intersection, and composition. Lexicon networks are almost always composed together with rule networks that map the somewhat abstract lexical strings into correctly spelled surface strings. For some natural languages, it is possible and convenient to divide up the work, doing nouns, verbs, and adjectives separately; the resulting sublanguage networks can then simply be unioned together into a single lexical transducer when they are finished.

And why doesn't everyone do computational linguistics this way? Because although the mathematical properties of finite-state networks have been well understood for a long time, robust and efficient computer algorithms and compilers to build and manipulate finite-state networks proved difficult to write and have been available only since the mid 1980s; they are still being refined. The **Xerox** Finite-State Calculus and compilers make practical what was only a theoretical possibility a few years ago.

1.9 Finite-State Networks are Good Things

Without getting into formal details of how finite-state languages differ from other kinds of formal languages, such as context-free and context-sensitive languages, we do want to emphasize here that computing with finite-state machines is attractive. First, the mathematical properties of finite-state networks are well understood, allowing us to manipulate and combine finite-state networks in ways that would be impossible using traditional algorithmic programs; there is a mathematical beauty to finite-state computing that translates into unparalleled flexibility. Second, finite-state networks are computationally efficient for tasks like natural-language morphological analysis, resulting in phenomenal processing speeds. Third, in most cases, finite-state networks can store a great deal of information in relatively little memory, and finite-state networks can be further compressed using commercial **Xerox** technology.

From a development point of view, finite-state programming is declarative; the linguist encodes facts about the language being modeled, not ad hoc algorithms. The algorithms that are required for the compilation of networks, and for applying them to do analysis and generation, are part of the toolkit and are completely language-independent. Multiple language modules therefore all share the same runtime "engine", and adding new language modules to an existing system involves just plugging new finite-state networks into the existing framework. Finite-state morphology is an excellent example of the principle of separating language-independent code from language-dependent data in natural-language processing systems.

1.10 Exercises

It takes some mind-tuning to become skilled in thinking about and computing with finite-state networks; it is a very different paradigm from the writing of procedures and algorithms. Throughout the book, we will present exercises to consolidate what has been learned, and these are the first. Be aware that superficially different solutions may be equivalent.

1.10.1 The Better Cola Machine

Starting from the example in Section 1.2.3, consider a slightly more sophisticated cola machine that returns change. As in the original Cola Machine, the valid inputs are nickels (**N**) worth 5 cents, dimes (**D**) worth 10 cents and quarters (**Q**) worth 25 cents; and it still costs 25 cents for a drink. As before, we won't try to model the selection or the delivery of drinks, but the new and better machine should now return appropriate change when you enter too much money. In particular:

- If the machine has reached the final state and the user continues to add coins, the extra coins will simply be returned to the user.
- If the user enters a partial sum, e.g. 2 dimes (equal to 20 cents), and then (s)he enters too much money (i.e. either another dime or a quarter), then the machine will move into the final state and return appropriate change.

Your task is to draw this better cola machine using states and labeled arcs.

Hints:

1. Start with the machine as shown in Figure 1.8. All the states and arcs in this machine are still valid for this exercise.
2. Arbitrarily, we will choose to think of entering coins as a kind of GENERATION, so we will match our inputs against the upper-side symbols of the network.
3. We will need to model the new machine as a two-level transducer, with input symbols on top of the arcs and output symbols on the bottom of the arcs. Use epsilon symbols where necessary so that there is always one symbol (or epsilon) above, and one symbol (or epsilon) below, each arc. Symbols indicating our change (a kind of output) should appear on the bottom of arcs.
4. From the final state, there should be three arcs that loop back to the final state:
 - One labeled with **N** on top and **N** on the bottom
 - One labeled with **D** on top and **D** on the bottom
 - One labeled with **Q** on top and **Q** on the bottom

This models what happens when too much money is entered: the extra coins are simply returned.

5. You will need to create some new intermediate states in the network diagram.

Trace the behavior of your machine through the following scenarios:

- If you enter exactly 25 cents, you should reach the final state as in the original cola machine, with no change returned.
- If you enter a quarter and then a nickel, you should reach the final state and get back a nickel.
- If you enter too little money, you should not reach the final state (and you will therefore go thirsty).
- If you enter three dimes (“DDD”) you should reach the final state and get back a nickel (N).
- If you enter three nickels and then a quarter, the machine should reach the final state and return 15 cents in some appropriate combination of coins (the choice of coins to be returned is up to you).
- Imagine and handle all other possible scenarios to make sure that your machine is robust and honest.

One solution to this exercise is shown on page 621.

1.10.2 The Softdrink Machine

The next exercise is to draw an even more sophisticated Softdrink Machine. This Softdrink Machine gives correct change, allows you to choose and receive your favorite drink, and even allows you to abort your purchase (and get your money back) by pressing the Coin Return button.

1. Start with a copy of The Better Cola Machine. All the states and arcs remain valid for this exercise.
2. In addition to the old inputs, **N**, **D** and **Q**, the Softdrink machine also has the following legal inputs in its alphabet:
 - **C**, which is what you input when you press the **Cola** button
 - **O**, which is what you input when you press the **Orange** button
 - **L**, which is what you input when you press the **Lemon-Lime** button
 - **R**, which is what you input when you press the **Coin Return** button

We arbitrarily choose to view the machine as a generator, so all the input symbols, including input coins and **C**, **O**, **L** and **R**, should appear on the top of arcs. Output symbols, representing drinks and returned coins, appear on the lower side of arcs, roughly matching our experience of where outputs appear on real soft-drink machines. I.e. we enter coins in a slot toward the top, and the drinks and any change fall out toward the bottom.

3. In addition to the old outputs, **N**, **D** and **Q** (representing change from overpayment), the new Softdrink Machine also has the following possible outputs:
 - **COLA**, a multicharacter symbol representing a can of cola
 - **ORANGE**, a multicharacter symbol representing a can of orange soda
 - **LEMONLIME**, a multicharacter symbol representing a can of lemon-lime soda

The output symbols **COLA**, **ORANGE** and **LEMONLIME** should appear only on the bottom of appropriate arcs.

4. From the final state, entering **C** should cause **COLA** to be returned, and the machine should move back into the Start state (and similarly for **O** and **L**). In other words, there should be an arc leading from the final state to the start state, with **C** on top, and **COLA** on the bottom.
5. From a non-final state, entering **C**, **O** or **L** should have no effect. The machine should accept the input, loop back to the same state, and output nothing.
6. From any state, entering **R** should cause any money already entered to be returned, and the machine should transition back to the start state. When giving change and when performing a coin return, don't worry about returning the same coins; just return some combination of coins adding up to the amount already entered.

Anticipate and handle all possible input scenarios so that your machine is honest and robust. Test it, by tracing the path through the states and arcs, for at least the following cases:

- You enter “DNNNC”. The machine should return “COLA”.
- You enter “DDNQNO”. The machine should return “QNORANGE”.
- You enter “DDDL”. The machine should return “NLEMONLIME”.
- You enter “DDQC”. The machine should return 20 cents (in some appropriate combination of coins) and a “COLA”.

- You enter “NNLDO”. The machine should return “ORANGE”.
- You enter “NDR”. The machine should return 15 cents, in some appropriate combination of coins.

1.10.3 Relations and Relatives

Father-in-Law

In Section 1.5.3 on Composition we suggested that the *father-in-law* relation is the composition of the *father* and *spouse* relations. Perhaps this is wrong. When Charles and Diana were married, Diana’s mother Frances and Diana’s father John, the 8th Earl of Spencer, had divorced. Frances had become Mrs. Raine McCorquodale. In the meantime she had also been married to Mr. Peter Kydd. While Mr. Kydd is clearly not Prince Charles’ father-in-law, perhaps Mr. McCorquodale is, by virtue of being the husband of Diana’s mother. In that case Charles has two fathers-in-law.

Using composition and possibly other set operations, give a new definition of *father-in-law* that gives us the relation $\{\langle \text{Philip}, \text{Diana} \rangle, \langle \text{John}, \text{Charles} \rangle, \langle \text{Raine}, \text{Charles} \rangle\}$, expanding the family tree as shown in Figure 1.29.

Cousin

The inverse of a relation R , denoted R^{-1} , consists of the same pairs as R but in the opposite order. Thus *husband* is the same relation as *wife*⁻¹; *child* is the inverse of *parent*, and vice versa.

Give a definition of *cousin* using inverse relations, composition, and possibly other set operations to ensure that no one is his own cousin or the cousin of his brother or sister.

Hint: You may find it convenient to start by defining *sibling* as the composition of *child* and *parent* minus the identity relation. The subtraction of the identity relation is required to satisfy the intuition that no one is his own sibling.

1.10.4 Lowercase/Uppercase Composition

The composition of the network representing $\{\langle \text{“Katze”}, \text{“Katze”} \rangle\}$ with the lowercase/uppercase transducer in Figure 1.46 yields the empty relation as the result. Why? Because the string “Katze” contains both uppercase and lowercase letters. Thus it is neither in the upper nor the lower language of the network.

What modification in the network in Figure 1.46 needs to be made if we want the outcome in Figure 1.55 when the “Katze” network comes first in the composition?

What result do we get if we put the new lowercase/uppercase transducer on top and compose it with a network representing $\{\langle \text{“KATZE”}, \text{“KATZE”} \rangle\}$. Why?

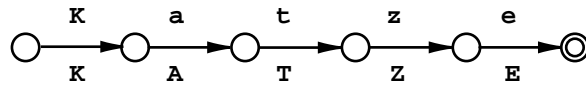


Figure 1.55: New upper-case rule applied to “Katze”

Chapter 2

A Systematic Introduction

Contents

2.1	Where We Are	45
2.2	Basic Concepts	46
2.3	Simple Regular Expressions	47
2.3.1	Definitions	48
2.3.2	Examples	53
2.3.3	Closure	56
2.3.4	The ANY Symbol	58
2.3.5	Operator Precedence	62
2.3.6	More About Symbols	62
	Special Symbols	63
	Multicharacter Symbols	63
	Tokenization of Input Strings into Symbols	64
2.4	Complex Regular Expressions	64
2.4.1	Restriction	65
2.4.2	Replacement	67
	Unconditional Replacement	67
	Conditional Replacement	70
	Directed Replacement	73
2.5	Properties of Finite-State Networks	75
2.6	Exercises	78
2.6.1	The Better Cola Machine	79

2.1 Where We Are

The previous chapter introduced in a gentle informal way some basic concepts of finite-state linguistics:

- A set of strings, a LANGUAGE, can be represented as a simple finite-state network that consists of states and arcs that are labeled by atomic symbols. Each string (= *word*) of the language corresponds to a path in this network.
- A set of pairs of strings, a RELATION, can be represented as a finite-state transducer. The arcs of the transducer are labeled by symbol pairs. Each path of the transducer represents a pair of strings in the relation. The first string belongs to the UPPER language, the second string belongs to the LOWER language of the relation.
- New languages and relations can be constructed by set operations such as UNION, CONCATENATION, and COMPOSITION. These operations can also be defined for finite-state networks to create a network that encodes the resulting language or relation. Some network operations such as INTERSECTION and SUBTRACTION can be defined only for languages, not for relations.

In this chapter we will examine in greater detail the relationship between languages and relations, their descriptions, and the finite-state automata that encode them. We will try to be precise and clear without being pedantic. We wish to avoid introducing too many formal definitions and irrelevant details. This is a systematic introduction rather than a formal one. We will not present proofs, theorems, or detailed algorithms but will include some references to where they can be found.

2.2 Basic Concepts

Finite-state networks can represent only a subset of all possible languages and relations; that is, only some languages are finite-state languages. One of the fundamental results of formal language theory (Kleene, 1956) is the demonstration that finite-state languages are precisely the set of languages that can be described by a REGULAR EXPRESSION.

Figure 2.1 gives a simple example that shows (1) a minimal regular expression, (2) the language it describes, and (3) a network that encodes the language. It illustrates how these notions are related to one another.

As Figure 2.1 indicates, a regular expression DENOTES a set of strings (i.e. a language) or a set of string pairs (i.e. a relation). It can be COMPILED INTO a finite-state network that compactly ENCODES the corresponding language or relation that may well be infinite.

The language of regular expressions includes the common set operators of Boolean logic and operators such as concatenation that are specific to strings. It follows from Kleene's theorem that for each of the regular-expression operators for finite-state languages there is a corresponding operation that applies to finite-state networks and produces a network for the resulting language. Any regular expression can in principle be compiled into a single finite-state network.

We can build a finite-state network for a complex language by first constructing a regular expression that describes the language in terms of set operations and by

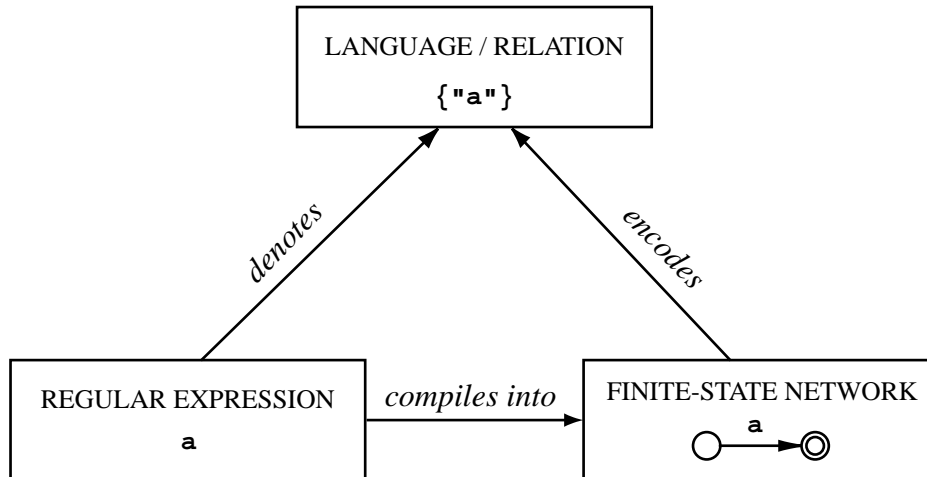


Figure 2.1: Regular Expression – Language – Network.

then compiling that regular expression into a network. This is in general easier than constructing a complex network directly, and in fact it is the only practical way for all but the most trivial infinite languages and relations.

We use the term FINITE-STATE NETWORK to cover both simple automata that encode a regular language and transducers that encode a regular relation. A network consists of STATES and ARCS. It has a single designated START STATE and any number (zero or more) of FINAL STATES. In our network diagrams states are represented by circles. The leftmost circle is the start state; final states are distinguished by a double circle. Each state is the origin of zero or more labeled arcs leading to some destination state. The arc labels are either simple symbols or symbol pairs depending on whether the network encodes a language or a relation between two languages. Each string or pair of strings is encoded as a path of arcs that leads from the start state to some final state. In the network of Figure 2.1 there is just one path; it encodes the string “a”. Unlike many introductory textbooks, we do not treat simple finite-state automata and transducers as different types of mathematical objects. Our presentation reflects rather closely the data structures in the actual **Xerox** implementation of finite-state networks. We hope it is as precise but more approachable than a rigorous formal introduction in terms of n -tuples of sets and functions (Roche and Schabes, 1997).

2.3 Simple Regular Expressions

We will start with a simple regular-expression language and expand it later in Section 2.4 with more constructions and operators. Even this initial description introduces many more types of regular expression operators than can be found in classical computer science literature (Hopcroft and Ullman, 1979). Because the expressions we use are meant to be typed, on the computer, our notation is slightly

different from what is used in most textbooks. No Greek symbols, just simple ASCII text.

We define the syntax and the semantics of the language in parallel. Many regular-expression operations can be applied both to regular languages and to regular relations. But some operators can be applied only to expressions whose denotation is semantically of the proper type, a language or a relation. In particular, complementation, subtraction, and intersection can be applied to all regular languages but only to a subset of regular relations. This issue will be discussed in detail in Section 2.3.3.

However, we are intentionally and systematically not making a distinction between a language and the identity relation on that language. The identity relation is the relation that maps every string to itself. If an operation such as concatenation is applicable to both languages and relations and if one operand denotes a language and the other a relation, we will automatically interpret the former as the identity relation on the language. Thus the result will be a relation as usual.

2.3.1 Definitions

We first introduce some atomic expressions and then construct more complex formulas with brackets, parentheses, braces, and regular-expression operators. In this section we define a family of basic operators: *optionality*, *iteration*, *complementation*, *concatenation*, *containment*, *ignoring*, *union*, *intersection*, *subtraction*, *crossproduct*, *projection*, *reverse*, *inverse*, and *composition*. More operators will be introduced in Section 2.4.

In the following definitions we use uppercase letters, A, B, etc., as variables over regular expressions. Lower case letters, a, b, etc., stand for symbols. We shall have more to say about symbols in Section 2.3.6.

Atomic Expressions

- The EPSILON symbol \emptyset denotes the empty-string language or the corresponding identity relation.
- The ANY symbol $?$ denotes the language of all single-symbol strings or the corresponding identity relation. The empty string is not included in $?$.
- Any single symbol, \mathbf{a} , denotes the language that consists of the corresponding string, here “a”, or the identity relation on that language.
- Any pair of symbols $\mathbf{a}:\mathbf{b}$ separated by a colon denotes the relation that consists of the corresponding pair of strings, $\{\langle\mathbf{a}, \mathbf{b}\rangle\}$. We refer to \mathbf{a} as the UPPER symbol of the pair $\mathbf{a}:\mathbf{b}$ and to \mathbf{b} as the LOWER symbol.

A pair of identical symbols, except for the pair $?:?$, is considered to be equivalent to the corresponding single symbol. For example, we do not distinguish between $a:a$ and a .

- The pair $?:?$ denotes the relation that maps any symbol to any symbol including itself. The identity relation, denoted by $?$, is a subrelation of $?:?$ that maps any symbol only to itself. Because $?$ does not cover the empty string, $?:?$ is an example of an EQUAL-LENGTH RELATION: a relation between strings that are of the same length. In this case, $\text{length} = 1$.
- The boundary symbol $. \# .$ designates the beginning of a string in the left context and the end of the string in the right context of a restriction or a replace expression. See Sections 2.4.1 and 2.4.2 for discussion. $. \# .$ has no special meaning elsewhere.

Brackets

- $[A]$ denotes the same language or relation as A .
- $[]$ denotes the empty string language, i.e. the language that consists of the empty string. $[]$ is thus equivalent to 0 .
- $[. .]$ has a special meaning in replace expressions. See Section 2.4.2 for discussion of dotted brackets.

In order to ensure that complex regular expressions have a unique syntax, we should in principle insist that *all* compound expressions constructed by means of an operator be enclosed in brackets. However, in keeping with the common practice, we do not insist on complete bracketing. As we shall see shortly, some syntactic ambiguities are semantically irrelevant. Bracketing can also be omitted when the intended interpretation can be derived from the rules of operator precedence presented in Section 2.3.5. However, we often add brackets just for clarity, to mark off the regular expression from the surrounding text.

Be careful to distinguish square brackets from round parentheses. Parentheses indicate optionality.

Optionality

- (A) denotes the union of the language or relation A with the empty string language. I.e. (A) is equivalent to $[A \mid 0]$.

Restriction: None. A can be any regular expression.

Iteration

- A^+ denotes the concatenation of A with itself one or more times. The $+$ operator is called *Kleene-plus* or *sigma-plus*.

- A^* denotes the union of A^+ with the empty string language. In other words, A^* denotes the concatenation of A with itself zero or more times. The $*$ operator is known as *Kleene-star* or *sigma-star*. $?^*$ denotes the UNIVERSAL LANGUAGE, the set of all strings of any length including zero. $[? : ?]^*$, equivalent to the notation $? : ?^*$, denotes the UNIVERSAL EQUAL-LENGTH RELATION, the mapping from any string to any string of the same length.

Restriction: None. A can be any regular expression.

Complementation

- $\sim A$ denotes the complement language of A , that is, the set of all strings that are not in the language A . The complementation operator \sim is also called *negation*. $\sim A$ is equivalent to $[?^* - A]$.
- $\setminus A$ denotes the term complement language, that is, the set of all single-symbol strings that are not in A . The \setminus operator is also called *term negation*. $\setminus A$ is equivalent to $[? - A]$.

Restriction: In both types of complementation, A must denote a language. The operation is not defined for relations. For more about this topic see Section 2.3.3.

Concatenation

- $[A B]$, equivalent to the notation $A B$, denotes the concatenation of the two languages or relations. The white space between regular expressions serves as the concatenation operator. Concatenation is an ASSOCIATIVE operation, that is, $[A [B C]]$ is equivalent to $[[A B] C]$. Because the order in which multiple concatenations are performed makes no difference for the result, we can ignore it and write simply $[A B C]$ omitting the inner brackets. This is an example where a syntactic ambiguity is semantically irrelevant. The same applies to all associative operations.
- $\{s\}$, where s is some string of alphanumeric symbols “abc...” denotes the concatenation of the corresponding single-character symbols $[a b c \dots]$. For example, $\{\text{word}\}$ is equivalent to $[w o r d]$.
- A^n denotes the n -ary concatenation of A with itself. n must be an integer. For example, a^3 is equivalent to $[a a a]$.
- $A^{<n}$ denotes less than n concatenations of A , including the empty string. For example, $a^{<3}$ denotes the language containing “” (the empty string), “a”, and “aa”.
- $A^{>n}$ denotes more than n concatenations of A . For example, $a^{>3}$ is equivalent to $[a a a a^+]$.

- $A^{\{i, k\}}$ denotes from i to k concatenations of A . For example, the language $a^{\{1, 3\}}$ contains the strings “a”, “aa”, and “aaa”.

Restriction: None. A and B can be any regular expressions.

Containment

- $\$A$ denotes the language or relation obtained by concatenating the universal language both as a prefix and as a suffix to A . For example, $\$[a\ b]$ denotes the set of strings such as “cabbage” that contain at least one instance of “ab” somewhere. $\$A$ is equivalent to $[?^* A ?^*]$.

Restriction: None. A can be any regular expression.

Ignoring

- $[A / B]$ denotes the language or relation obtained from A by splicing in B^* everywhere within the strings of A . For example, $[a\ b / x]$ denotes the set of strings such as “xxxxaxxxbx” that distort “ab” by arbitrary insertions of “x”. Intuitively, $[A / B]$ denotes the language or relation A ignoring arbitrary bursts of “noise” from B .
- $[A . / . B]$ denotes the language or relation obtained from A by splicing in B^* everywhere in the *inside* of the elements of A but not at the edges. For example, $[a\ b . / . x]$ contains strings such as “axxxb” but not “xab” or “axbx”.

Restriction: None. A and B can be any regular expressions.

Union

- $[A | B]$ denotes the union of the two languages or relations. The union operator $|$ is also called DISJUNCTION. The union operation is associative and also COMMUTATIVE; that is, $[A | B]$ and $[B | A]$ denote the same language or relation.

Restriction: None. A and B can be any regular expressions

Intersection

- $[A \& B]$ denotes the intersection of the two languages. The intersection operator $\&$ is also called CONJUNCTION. The operation is associative and commutative. Intersection can be expressed in terms of complementation and union. $[A \& B]$ and $\sim[\sim A | \sim B]$ are equivalent.

Restriction: A and B must denote languages or equal-length relations. Relations cannot, in general, be intersected. We explain the reason in Section 2.3.3.

Subtraction

- $[A - B]$ denotes the language of all the strings in A that are not members of B . $[A - B]$ is equivalent to $[A \ \& \ \sim B]$. Subtraction is neither associative nor commutative.

The two varieties of complementation introduced above could be defined in terms of subtraction because $\sim A$ is equivalent to $[?^* - A]$ and $\setminus A$ is equivalent to $[? - A]$.

Restriction: A and B should denote languages or equal-length relations. Relations cannot, in general, be subtracted.

Crossproduct

- $[A \ .x. \ B]$ denotes the relation that pairs every string of language A with every string of language B . Here A is called the *upper* language and B the *lower* language of the relation.

Because a pair such as $a : b$ denotes the relation between the corresponding strings, $[a \ .x. \ b]$ and $a : b$ are obviously equivalent expressions. The expression $[?^* \ .x. \ ?^*]$ denotes the *universal relation*, the mapping from any string to any string, including the empty string.

- $[[A] : [B]]$ denotes the relation that pairs every string of language A with every string of language B . Here A is called the *upper* language and B the *lower* language of the relation.

The $.x.$ and the $:$ are both general-purpose crossproduct operators, but they differ widely in precedence (see Section 2.3.5), with $:$ having very high precedence and $.x.$ very low precedence. In practice, it is most important to remember that $.x.$ has lower precedence than concatenation, so that $[c \ a \ t \ .x. \ c \ h \ a \ t]$ denotes the same relation as $[[c \ a \ t] \ .x. \ [c \ h \ a \ t]]$; in contrast, the $:$ has higher precedence than concatenation, such that $[c \ a \ t : \ c \ h \ a \ t]$ is equivalent to $[c \ a \ [t : c] \ h \ a \ t]$.

Restriction: A and B must denote languages, not relations. This restriction is inherent to how crossproduct is defined.

Projection

Recall that a relation relates two regular languages, called the upper language and the lower language.

- $A . u$ denotes the upper language of the relation of A
- $A . l$ denotes the corresponding lower language of A .

Restriction: None. If A denotes a language, A is interpreted as the identity relation on A . In that case, $A . u$ and $A . l$ denote the same language as A .

Reverse

- $A.r$ denotes the reverse of the language or relation A . For example, if A contains $\langle \text{"abc"}, \text{"xy"} \rangle$, the reverse relation $A.r$ contains $\langle \text{"cba"}, \text{"yx"} \rangle$. Be careful to distinguish REVERSE and INVERSE. Reverse exchanges left and right; inverse exchanges the upper and lower sides of a relation.

Restriction: None. A can be any regular expression.

Inverse

- $A.i$ denotes the inverse of the relation A . For example, if A contains $\langle \text{"abc"}, \text{"xy"} \rangle$, the inverse relation $A.i$ contains $\langle \text{"xy"}, \text{"abc"} \rangle$. Be careful to distinguish *inverse* and *reverse*. Inverse exchanges the upper and lower sides of a relation; reverse exchanges left and right.

Restriction: None. If A denotes a language, in $A.i$ it is interpreted as an identity relation. In this case $A.i$ is indistinguishable from A .

Composition

- $[A \circ B]$ denotes the composition of the relation A with the relation B . If A contains the string pair $\langle x, y \rangle$ and B contains $\langle y, z \rangle$, the composite relation $[A \circ B]$ contains the string pair $\langle x, z \rangle$.
Composition is associative but not commutative. We can write $[A \circ B \circ C]$ because the result is the same relation regardless of whether we interpret it as $[A \circ [B \circ C]]$ or $[[A \circ B] \circ C]$.

Restriction: None. If A or B denotes a language, in $[A \circ B]$ it is interpreted as the corresponding identity relation. If both A and B denote languages $[A \circ B]$ and $[A \ \& \ B]$ are indistinguishable; composition and intersection yield the same result in this case.

2.3.2 Examples

In order to understand the semantics of regular expressions it is useful to look at some simple examples in conjunction with the corresponding language or relation and the network that encodes it. We will start with the minimal examples in Table 2.1 and proceed to cover some of the operators introduced in the previous section.

Because we have not introduced a special symbol for the EMPTY LANGUAGE, i.e. the language that contains no strings, not even the empty string, we have to designate it by some complex expression such as $\sim[?^*]$ that denotes the complement of the universal language. It compiles into a network with a single non-final state with no arcs. The empty-string language, i.e. the language that contains only



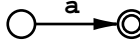
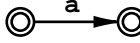
Expression	Language / Relation	Network
$\sim[?^*]$	$\{\}$	
$[\]$	$\{\text{""}\}$	
a	$\{\text{"a"}\}$	
(a)	$\{\text{"", "a"}\}$	

Table 2.1: Minimal Languages

the empty string, denoted by \emptyset or by $[\]$, corresponds to a network with a single final state with no arcs. Note that $[a]$ can also be interpreted as the identity relation $\{\langle \text{"a"}, \text{"a"} \rangle\}$. The network representation of the ANY symbol, $?$, is a complex issue. We will discuss it later in Section 2.3.4.

Table 2.2 illustrates the difference between the two iteration operators Kleene-star and Kleene-plus. Because the networks in Table 2.2 are cyclic, they contain an infinite number of paths. Looking at the middle column it is evident that a^* is equivalent to (a^+) and to $[a^+ \mid \emptyset]$. This is not as easily deduced from the corresponding minimal networks.


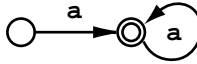
Expression	Language / Relation	Network
a^*	$\{\text{"", "a", "aa", \dots}\}$	
a^+	$\{\text{"a", "aa", \dots}\}$	

Table 2.2: Iteration

As the shape of network for a^+ in Table 2.2 suggests, a^+ is equivalent to the concatenation $[a \ a^*]$. Table 2.3 shows more examples of concatenation.

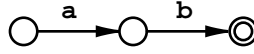
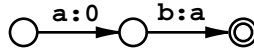
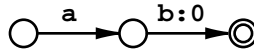
Expression	Language / Relation	Network
$a \ \emptyset \ b$	$\{\text{"ab"}\}$	
$a : \emptyset \ b : a$	$\{\langle \text{"ab"}, \text{"a"} \rangle\}$	
$a \ b : \emptyset$	$\{\langle \text{"ab"}, \text{"a"} \rangle\}$	

Table 2.3: Concatenation

The concatenation of two languages yields a language, the concatenation of two relations yields a relation. As the last example in Table 2.3 shows, we systematically ignore the difference between a LANGUAGE and its IDENTITY RELATION.

Thus $[a]$ is interpreted as the identity relation $\{ \langle "a", "a" \rangle \}$ when it is concatenated, unioned, or composed with a relation.

The last two regular expressions in Table 2.3 denote the same relation but they do not compile into the same network. This is an important point. There is no general method for deriving a unique canonical representation for a relation that pairs strings of unequal length. Two networks may encode the same relation but there is no general algorithm for deciding whether they do. In this respect regular relations are computationally more difficult to handle than regular languages. Every regular language has a unique minimal encoding as a network, but some relations do not. We will come back to this point later.

The crossproduct of two languages may also produce pairs of strings that are of different length, as in the second example in Table 2.4

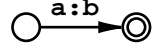
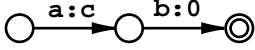
Expression	Language / Relation	Network
$a .x. b$	$\{ \langle "a", "b" \rangle \}$	
$[a b] .x. c$	$\{ \langle "ab", "c" \rangle \}$	

Table 2.4: Crossproduct

The pair of strings $\langle "ab", "c" \rangle$ could be encoded in a network in other ways, for example, by a path in which the successive labels are $a:0$ and $b:c$ instead of $a:c$ and $b:0$ as in Table 2.4. However, because it is convenient to have a unique encoding for crossproduct relations, the **Xerox** compiler pairs the strings from left to right, symbol by symbol, and introduces one-sided epsilon pairs only at the right end of the path if needed. This is an arbitrary choice.

The final set of examples in Table 2.5 illustrates some simple cases of composition. In the last example, we interpret b as the identity relation because composition is inherently an operation on relations and not on languages.

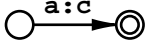
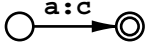
Expression	Language / Relation	Network
$a:b .o. b:c$	$\{ \langle "a", "c" \rangle \}$	
$a:b .o. b .o. b:c$	$\{ \langle "a", "c" \rangle \}$	

Table 2.5: Composition

Regular expressions were originally invented to describe languages. The formalism was subsequently extended to describe relations. But as we have noted in many places, some operators can be applied only to a subset of regular expressions. Let us now review this issue in more detail.

2.3.3 Closure

A set operation such as union has a corresponding operation on finite-state networks only if the set of regular relations and languages is CLOSED under that operation. Closure means that if the sets to which the operation is applied are regular, the result is also regular, that is, encodable as a finite-state network.

Regular languages are closed with respect to all the common set operations including intersection and complementation (= negation). This follows directly from Kleene's proof. Regular relations are closed under concatenation, iteration, union, and composition but not, in general, under intersection, complementation or subtraction (Kaplan and Kay, 1994; Roche and Schabes, 1997).

Kaplan and Kay give a simple example of a case in which the intersection of two finite-state relations is not a finite-state relation. Consider two regular expressions and the corresponding networks in Figure 2.2.

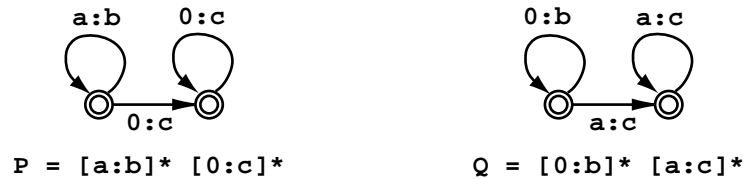


Figure 2.2: Networks for Two Regular Relations

P is the relation that maps strings of any number of **as** into strings of the same number of **bs** followed by zero or more **cs**. Q is the relation that maps strings of any number of **as** into strings of the same number of **cs** preceded by zero or more **bs**.

Table 2.6 shows the corresponding string-to-string relations and their intersection. The left side of Table 2.6 is a partial enumeration of the P relation. The right side partially enumerates the Q relation. The middle section of the table contains the intersection of the two relations, that is, the pairs they have in common.

It is easy to see that the intersection contains only pairs that have on their lower side (i.e. as the second component) a string that contains some number of **bs** followed by exactly the same number of **cs**.

The lower-side language, $b^n c^n$, is not a finite-state language but rather a CONTEXT-FREE LANGUAGE, generated by a phrase-structure grammar that crucially depends on center-embedding: $S \rightarrow \varepsilon$, $S \rightarrow b S c$. Consequently it cannot be encoded by a finite-state network (Hopcroft and Ullman, 1979). The same holds of course for any relation involving this language. No operation on the networks in Figure 2.2 can yield a finite-state transducer for the intersection of P and Q ; such a transducer does not exist.

The relations P and Q both have the property that a string in one language corresponds to infinitely many strings in the other language because of the iterated $0:b$ and $0:c$ pairs. It is this characteristic, the presence of “one-sided epsilon

P Regular	P & Q Not Regular	Q Regular
<“”, “”>	<“”, “”>	<“”, “”>
<“”, c>		<“”, b>
<“”, cc>		<“”, bb>
...		...
<a, b>		<a, c>
<a, bc>	<a, bc>	<a, bc>
<a, bcc>		<a, bbc>
<a, bccc>		<a, bbbc>
...		...
<aa, bb>		<aa, cc>
<aa, bbc>		<aa, bcc>
<aa, bbcc>	<aa, bbcc>	<aa, bbcc>
<aa, bbccc>		<aa, bbbcc>
...

Table 2.6: Non-Regular Intersection of P and Q

loops” in Figure 2.2 that makes it possible that their intersection is not regular.

From the fact that regular relations are not closed under intersection it follows immediately that they are not closed under complementation either. Intersection can be defined in terms of complementation and union. If regular relations were closed under complementation, the same would be true of intersection. It also follows that regular relations are not closed under the subtraction operation, definable by means of intersection and complementation.

The closure properties of regular languages and relations are summarized in Table 2.7 for the most common operations.

Operation	Regular Languages	Regular Relations
union	yes	yes
concatenation	yes	yes
iteration	yes	yes
reversal	yes	yes
intersection	yes	no
subtraction	yes	no
complementation	yes	no
composition	(not applicable)	yes
inversion	(not applicable)	yes

Table 2.7: Closure Properties

Although regular relations are not in general closed under intersection, a subset

of regular relations have regular intersections. In particular, equal-length relations, relations between pairs of strings that are of the same length, are closed under intersection, subtraction, and complementation. Such relations can be encoded by a transducer that does not contain any epsilon symbols. This fact is important for us because it is the formal foundation of the two-level rule formalism called **twolc**, introduced in Chapter 5. Transducers compiled from two-level rules can be intersected because 0 is treated as an ordinary symbol in the rule formalism and not as an empty string.

The **Xerox** calculus allows intersection to apply to all simple automata and to transducers that do not contain any one-sided epsilon pairs. The test is more restrictive than it should be in principle because the presence of one-sided epsilons in a transducer does not necessarily indicate that the relation it encodes is of the type that could yield a non-regular intersection.

2.3.4 The ANY Symbol

So far we have given examples of regular expressions and the corresponding networks for most of the operators introduced in Section 2.3.1. One notable exception is complementation. We will come to that shortly but first we need to understand the semantics of the ANY symbol, $?$, introduced in the beginning of Section 2.3.1 as one of the atomic expressions.

Let us recall that the term complement of a language A , denoted by $\setminus A$, is the union of the single-symbol strings that are not in A . For example, the language $[\setminus a]$ contains “b”, “c”, ... “z”. In fact $[\setminus a]$ is an infinite language because the set of atomic symbols is in principle unbounded. Our symbol alphabet is not restricted to the 26 letters of the English alphabet or to the 256 ASCII characters.

For this reason we provide a special symbol, $?$, to represent the infinite set of symbols in some yet unknown alphabet. It is called the ANY symbol. The regular expression $?$ denotes the language of all single-symbol strings. Note that this set does not include the empty string. Because we do not make a distinction between a language and an identity relation, $?$ can also be interpreted as the relation that maps any symbol into itself. The corresponding network is obviously the one in Figure 2.3. But note the annotation *Sigma*: $\{ ? \}$. We will explain it shortly.

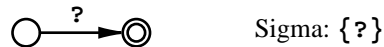


Figure 2.3: The Language of All Single-Symbol Strings $[?]$.

The correspondence between the regular expression $?$ and the network that encodes it in Figure 2.3 is deceptively straightforward. In fact the symbol $?$ in the regular expression does not have exactly the same interpretation that the arc label $?$ has in the network.

In the regular expression, $?$ stands for ANY symbol; the corresponding arc label $?$ represents any UNKNOWN symbol. In the case at hand the difference is subtle.

Because the alphabet of known symbols in this network contains just ϵ , the arc labeled ϵ does represent any symbol whatsoever in this case. We call the known alphabet of a network its SIGMA. Thus the annotation *Sigma*: $\{\epsilon\}$ in Figure 2.3 means that the sigma is empty except for ϵ .

The difference between ϵ as a regular expression symbol and ϵ as an arc label in networks is an unfortunate source of confusion for almost all users of the **Xerox** regular-expression calculus. Using two different symbols, say ϵ for ANY and ζ for UNKNOWN would make it explicit. However, employing two different symbols suggests that ϵ and ζ could both appear in regular expressions and as arc labels. This would be misleading. The **Xerox** calculus is designed and implemented in such a way that ANY is exclusively a regular expression concept and the concept of UNKNOWN is relevant only for finite-state networks. Because of this fact, we have chosen to “overload” the ϵ sign. We interpret ϵ as the ANY symbol in regular expressions and we also use ϵ to display the UNKNOWN symbol in network diagrams.

An expression that in some way involves complementation typically compiles into a network that contains an instance of the unknown symbol. In such cases it is crucial to know the sigma alphabet because it cannot in general be deduced from the arcs and the labels of the network. For example, the language $[\neg a]$, consisting of all single-symbol strings other than “a”, is encoded by the network in Figure 2.4.



Figure 2.4: The Language $[\neg a]$

In the network of Figure 2.4, ϵ does not include “a”, in the network of Figure 2.3 it does. $[\neg a]$ is equivalent to $[\epsilon - a]$.

The sigma alphabet includes every non-epsilon symbol that appears in the network either by itself or as a component of a symbol pair. As we see here, the sigma alphabet may also contain symbols that do not appear in the network but are “known” because they were present in another network from which the current one was derived by some operation that eliminated all the arcs where these symbols occurred.

Complex regular expressions that contain ϵ as a subexpression must be compiled carefully so that the sigma alphabet is correct and all the required arcs are present in the resulting network; this was a challenge for the algorithm writers. Table 2.8 illustrates the issue with a few examples. The first network in Table 2.8 is the result of concatenating the $[a]$ network shown in Table 2.1 with the $[\epsilon]$ network shown in Figure 2.3. The original networks both contain just one arc but the result of the concatenation has three arcs. The compiler adds a redundant “a” arc to the $[\epsilon]$ network to make explicit the fact that the string “a” is included in the language. This expansion is necessary because the sigma of the resulting $[a\epsilon]$ network in Table 2.8 contains a. Consequently the networks for $[a\epsilon]$ and

Regular Expression	Network	Sigma
$a ?$		$\{?, a\}$
$a \setminus a$		$\{?, a\}$
$a : ?$		$\{?, a\}$
$? : ?$		$\{?\}$
$a \setminus ?$		$\{\}$

Table 2.8: Expressions with ?

$[a \setminus a]$ correctly encode the fact that the language $[a ?]$ includes the string “aa” but the language $[a \setminus a]$ does not.

For the same reason, the network for $a : ?$ contains an explicit a -arc. Let us recall that the symbol pair $a : ?$ denotes the crossproduct $[a \cdot x \cdot ?]$. Because “a” is included in the language $[?]$, the crossproduct relation includes the pair $\langle \text{“a”}, \text{“a”} \rangle$ along with all the pairings of “a” with other single-symbol strings. This identity pair is encoded in the network by the a -arc; the arc labeled $a : ?$ covers all the other pairs.

The network for the expression $? : ?$ illustrates another subtle fact about the interpretation of the *unknown* symbol. The relation denoted by the regular expression $? : ?$ maps any symbol to any symbol including itself. That is, the relation $? : ?$ includes the identity relation denoted by the expression $?$. In the network for the relation $? : ?$, the identity part of the relation is represented by the arc labeled $?$, the non-identity mapping by the arc labeled $? : ?$.

The term complement $[\setminus ?]$ denotes the empty language. Concatenation with an empty language always yields the empty language. Consequently the concatenation $[a \setminus ?]$ is also empty, as the last example in Table 2.8 shows.

Figure 2.5 gives an example of a network compiled from an expression that contains \sim , the other complementation operator. Here we make use of the graphic convention that lets us represent any number of arcs that share the same origin and destination by a single multiply labeled transition. In fact the network in Figure 2.5 has six arcs of which only four are explicitly shown.

As the comparison between Figure 2.4 and Figure 2.5 shows, $[\setminus a]$ and $[\sim a]$ denote very different languages. The set of single-symbol strings other than “a”, $[\setminus a]$, is included in the language $[\sim a]$. In addition the latter language contains the empty string (the start state is final), and any string whatever whose length is

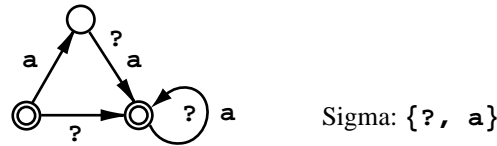


Figure 2.5: The Language $[\sim a]$

greater than one, for example, “aa”. $[\sim a]$ is equivalent to $[?^* - a]$.

The expression $?^*$ denotes the UNIVERSAL LANGUAGE, the set of all strings of any length including the empty string. The corresponding network is shown in Figure 2.6. We may also interpret $?^*$ as the UNIVERSAL IDENTITY RELATION: every string paired with itself.

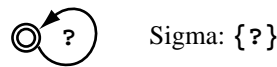


Figure 2.6: The Universal Language/Identity Relation $?^*$

If we added the symbols **a**, **b**, and **c** to the sigma of the network in Figure 2.6, the encoded language would be $[\backslash[a \mid b \mid c]]^*$.

The expression $[?:?]^*$ denotes the universal equal-length relation that pairs every string with all strings that have the same length. The corresponding network is shown in Figure 2.7. For the sake of clarity we draw here the two arcs explicitly instead of abbreviating the picture into a single arc labeled with two symbols.

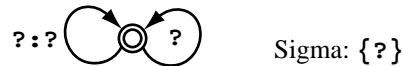
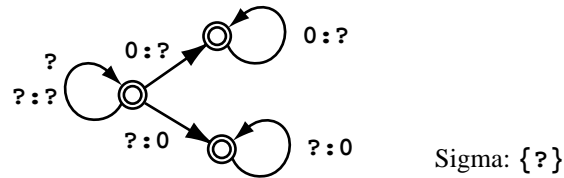


Figure 2.7: The Universal Equal-Length Relation $[?:?]^*$

As in the network for the relation $? : ?$ in Table 2.8, the arc labeled $?$ in Figure 2.7 represents the mapping of any unknown symbol into the same unknown symbol. The arc labeled $? : ?$ represents the mapping of any unknown symbol into a different unknown symbol. Both arcs are needed because the application of the network (in either direction) to a string such as “a” must yield two results: “a” itself and $?$ representing all other possibilities.

The expression $[?^* .x. ?^*]$ in Figure 2.8 is another interesting special case. It denotes the promiscuous UNIVERSAL RELATION that maps any string to any other string including itself.

The $?$ loop in the start state pairs any string with itself; the $? : ?$ loop pairs any string with any other string of equal length. The $0 : ?$ and $? : 0$ arcs allow a string to be paired with another string that does not have the same length. Because the relation $[?^* .x. ?^*]$ pairs any string with any string, the expression $[A .o. [?^* .x. ?^*] .o. B]$ is equivalent to the crossproduct of the “outer” languages of the A and B relations $[A .u. .x. B .l]$.

Figure 2.8: The Universal Relation $[?^* .x. ?^*]$

2.3.5 Operator Precedence

Like conventional arithmetic expressions, complex regular expressions can often be simplified by taking advantage of a convention that ranks the operators in a precedence hierarchy. For example, $[\backslash a [b]^*]$ may be expressed more concisely as $[\backslash a b^*]$ because by convention the unary operators, term complement and Kleene star, “bind more tightly” to their component expressions than concatenation. Similarly, the inner brackets in $[a | [b c]]$ are unnecessary because concatenation takes precedence over union; $[a | b c]$ denotes the same language. In turn, union has precedence over crossproduct, allowing us to simplify $[a .x. [b | c]]$ to $[a .x. b | c]$. The ranking of all the operators discussed in the previous sections is summarized in Table 2.9, including the restriction and replacement operators that will be introduced in Section 2.4.

Type	Operators
Crossproduct	:
Prefix	$\sim, \backslash, \$$
Suffix	$+, *, ^, .r, .u, .l, .i$
Ignoring	/
Concatenation	(whitespace)
Boolean	$, \&, -$
Restriction and replacement	$=>, ->$
Crossproduct and Composition	$.x., .o.$

Table 2.9: Precedence Ranking

Unbracketed expressions with operators that have the same precedence are disambiguated from left to right. Consequently, $[a | b \& b^*]$ is interpreted as $[[a | b] \& b^*]$ and not as $[a | [b \& b^*]]$. For the sake of clarity, we advise using brackets in such cases even if the ambiguity would be resolved in the intended way by the left-to-right ordering.

2.3.6 More About Symbols

In all of our examples only ordinary single letters have been used as symbols. In this section we discuss briefly how other types of symbols may be represented.

Because $\mathbf{0}$ and $\mathbf{?}$ have a special meaning in regular expressions, we need to

provide a way to use them as ordinary symbols. The same applies to the symbols used as regular expression operators: |, &, -, +, *, etc.

Special Symbols

The **Xerox** regular-expression compiler provides two ways to avoid the special interpretation of a symbol: prefixing the symbol with % or enclosing it in double quotes. For example, %0 denotes the string "0", containing the literal zero symbol, rather than the epsilon (empty string); %| denotes the vertical bar itself, as opposed to the the union operator |. The ordinary percent sign may be expressed as %% . Enclosure in double quotes has the same effect: "0", "?", and "% " are interpreted as ordinary letters.

On the other hand, certain other strings receive a special interpretation within a doubly quoted string. For example, "\n" is interpreted as the newline symbol, "\t" as a tab, etc. following C conventions. The backslash is also used as a prefix in octal and hexadecimal numbers: "\101" and "\x41" are alternate ways to specify the symbol A. Inside double quotes, the prefix \u followed by four hexadecimal numbers encodes a 16-bit Unicode character; e.g. "\u0633" is the Arabic *siin* character.

Xerox networks can store and manipulate 16-bit Unicode characters, but until Unicode-capable word processors and operating systems become generally available, input and output of such characters will be awkward.

Multicharacter Symbols

All the examples given so far involve symbols that consist of a single character. The **Xerox** calculus also admits multicharacter symbols such as **+Noun**. (In a regular expression this would have to be specified as "+Noun" or %"+Noun" to escape the special interpretation of +.) In linguistic applications, morphological and syntactic tags that convey information about part-of-speech, tense, mood, number, gender, etc. are typically represented by multicharacter symbols. It is often convenient to deal with HTML, SGML, and XML tags as atomic symbols rather than as a concatenation of single-character strings.

Multicharacter symbols are treated as atomic entities by the regular expression compiler; that is, the multicharacter symbol **+Noun** is stored and manipulated just like **a**, **b** and **c**. For example, the sigma alphabet of the network compiled from [{cat} "+Noun" : 0] consists of the symbols **a**, **c**, **t** and **+Noun**.

Because of multicharacter symbols the correspondence between a regular expression and the language it denotes is less direct than we have indicated. There can be multiple regular expressions that denote what could be seen as the same set of strings but compile into different networks because the strings are tokenized differently. See Table 2.10.

From the point of view of the finite-state calculus, these networks are not equivalent. Their intersection is the network for the empty set. To avoid confusion it is

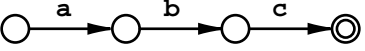
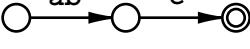
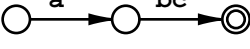
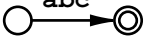
Expression	Network	Sigma
a b c		{a, b, c}
ab c		{ab, c}
a bc		{a, bc}
abc		{abc}

Table 2.10: Alternative Representations of {"abc"}

best to represent ordinary strings only as a sequence of single character symbols. The three other alternatives in Table 2.10 should be avoided although they are accepted by the regular expression compiler. The purpose of the curly-brace notation for concatenation (Section 2.3.1) is to facilitate representing words in the preferred format. We can write {abc} instead of [a b c] to tell the compiler to break or “explode” the string into a sequence of single-character symbols.

Tokenization of Input Strings into Symbols

The APPLY algorithms for finding the path or paths for a string in a network must take into account the network’s sigma alphabet and tokenize the input string into symbols accordingly. If the alphabet contains multicharacter symbols, there may not be a unique tokenization in all cases. For example, it might be the case that “+Noun” could be tokenized either as <+, N, o, u, n> or as a single symbol. The tokenizer resolves the ambiguity by processing the input string from left to right and by selecting at each point the longest matching symbol. The multicharacter symbol always has precedence. In order to avoid tokenization problems, multicharacter symbols should not be confusable with the ordinary alphabet. To help avoid such confusion, every multicharacter symbol should begin with some special character, or be surrounded by some special characters, that do not occur in ordinary alphabetic strings: “+Noun”, “[Noun]”, “^TOKEN”, “<TABLE>”, etc. For more on tokenization of input strings, see Sections 3.6.3 and 7.3.4.

2.4 Complex Regular Expressions

The set of regular-expression operators introduced in the preceding section is somewhat redundant because some of the operators could easily be defined in terms of others. For example, there is no particular need for the CONTAINS operator \$ since the expression [?* A ?*] denotes the same language as \$A. Its purpose is to allow this commonly used construction to be expressed more concisely.

In this section we will discuss regular expressions for RESTRICTION and RE-

PLACEMENT that are more complex than those we have covered so far. These rule-like expressions are specific to the regular-expression compiler in the **xfst** application described in Chapter 3. Like the CONTAINS operator, these new constructs are definable in terms of more primitive expressions. Thus they do not extend the descriptive power of the regular-expression language but provide a higher level of abstraction that makes it easier to define complex languages and relations.

2.4.1 Restriction

The restriction operator (Koskenniemi, 1983) is one of the two fundamental operators in the two-level calculus (i.e. the **twolc** language) described in Chapter 5. It is also used in the **xfst** regular-expression calculus.

Restriction

- $[A \Rightarrow L _ R]$ denotes the language of strings that have the property that any string from A that occurs as a substring is immediately preceded by some string from L and immediately followed by some string from R . We call L and R here the LEFT and the RIGHT context of A , respectively. For example, $[a \Rightarrow b _ c]$ includes all strings that contain no occurrence of “a” and all strings like “back-to-back” that completely satisfy the condition, but no strings such as “cab” or “pack”.
- $[A \Rightarrow L1 _ R1, L2 _ R2]$ denotes the language in which every instance of A is surrounded either by a pair of strings from $L1$ and $R1$ or by a pair of strings from $L2$ and $R2$. The list of allowed contexts, separated by commas, may be arbitrarily long.

Restriction: All components, A , L , R , etc. must denote regular languages, not relations.

The restriction operator has exactly the same definition in **xfst** as it has in the **twolc** two-level calculus. But note the restriction: all components must denote regular languages. Expressions such $[a:b \Rightarrow c:d _ e:f]$ are invalid in **xfst**. They are allowed in **twolc** rules because the two-level calculus actually treats symbol pairs as simple atomic symbols. In the **xfst** regular-expression language symbol pairs denote relations.

The restriction expressed by $[A \Rightarrow L _ R]$ is that the substrings in A must appear in the context $L _ R$. The same constraint could also be coded in more primitive terms by means of complementation, concatenation, iteration and union. The expression $[\sim[[\sim[?* b] a ?*] \mid [?* a \sim[c ?*]]]]$ denotes the same language as $[a \Rightarrow b _ c]$, but the latter expression is obviously more convenient and perspicuous.

The outer edges of the left and the right context extend implicitly to infinity. That is, $L _ R$ is compiled as $[?* L] _ [R ?*]$. A restriction with a completely empty context such as $[A \Rightarrow _]$ denotes the universal language as it

places no constraint on anything. Another consequence of the implicit extension is that an optional or a negative component at the outer edge of a context is vacuous. An expression such $[A \Rightarrow (b) _ \sim c]$ also denotes the universal language because $[?^* (b)]$ and $[\sim c ?^*]$ are both equivalent to $?^*$. The reason is that (b) and $\sim c$ both include the empty string. The concatenation between the universal language and any language that includes the empty string reduces to the universal language.

To refer to the beginning or the end of a string in a context specification, we need an explicit marker. The boundary symbol $.\#.$ indicates the beginning of a string in the left context and the end of a string in the right context. It has no special meaning anywhere else. The boundary symbol can be concatenated, unioned, etc. with other regular expressions like an ordinary symbol. For example, $[a \Rightarrow b _ c \mid .\#.]$ requires that **a** be followed by **c** or the end of the string. As Figure 2.9 shows, the boundary marker is compiled away and does not appear in the resulting network.¹

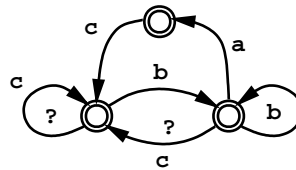


Figure 2.9: The Language $a \Rightarrow b _ c \mid .\#.$

It is easy to see that the network in Figure 2.9 encodes a language in which every occurrence of **a** is immediately preceded by a **b**, and that the string either terminates at the **a** or continues with a **c**.

With the help of the boundary symbol we can specify contexts that include the empty string. For example, $[a \Rightarrow .\#. \sim [b ?^*] _]$ expresses the constraint that **a** cannot be preceded by a string that starts with **b**. The set of permissible left-context strings, $\sim [b ?^*]$, includes the empty string. In order to restrict **a** in the intended way, the left context must start with the boundary symbol. Without it the restriction is vacuous: $[a \Rightarrow \sim [b ?^*] _]$ denotes the universal language $?^*$ where **a** may occur freely anywhere.

Restrictions with alternative contexts such as $[a \Rightarrow b _ c, d _ e]$ could also be expressed as an iterated union of elementary restrictions: $[[a \Rightarrow b _ c] \mid [a \Rightarrow d _ e]]^*$ denotes the same language. The Kleene star is needed here because the language must include not only strings like “bac” and “dae” but also strings like “bacdae” in which different instances of **a** are sanctioned by different contexts.

¹The sigma alphabet of the network in Figure 2.9 contains only symbols that are present in the network: a, b, c, and the unknown symbol. If there are no “hidden” symbols in the sigma alphabet, we generally do not list the sigma explicitly in our diagrams.

2.4.2 Replacement

This section introduces a large set of expressions for creating complex string-to-string relations. Replacement expressions describe strings of one language in terms of how they differ from the strings of another language. Because the differences may depend on contexts and other parameters, the syntax of replacement expressions involves many operators and special constructs.

The family of replace operators is specific to the **Xerox** regular-expression calculus. In the original version, developed by Ronald M. Kaplan and Martin Kay in the early 1980s (Kaplan and Kay, 1981; Kaplan and Kay, 1994), the goal was to model the application of phonological rewrite rules by finite-state transducers. Although the idea had been known before (Johnson, 1972), Kaplan and Kay were the first to present a compilation algorithm for rewrite rules. The notation introduced in this section is based on Kaplan and Kay's work with extensions introduced by researchers at XRCE (Karttunen, 1995; Karttunen and Kempe, 1995; Karttunen, 1996; Kempe and Karttunen, 1996).

We introduce the main types of replacement expressions first in their unconditional version and then go on to discuss how the operation can be constrained by context and by other conditions.

Unconditional Replacement

A very simple replacement of one string by another can be described as a crossproduct relation. For example, $[a\ b\ c\ .x.\ d\ e]$ maps the string "abc" to "de" and vice versa. The upper and the lower languages of this relation consist of a single string. In contrast, the replacement expressions introduced in this section generally denote infinite relations.

Simple Replacement

- $[A \rightarrow B]$ denotes the relation in which each string of the universal language (the upper language) is paired with all strings that are identical to it in every respect except that every instance of A that occurs as a substring in the original upper-language string is represented by a string from B.
- $[A \leftarrow B]$ denotes the inverse of $B \rightarrow A$.
- $[A (-\rightarrow) B]$ denotes an optional replacement, that is, the union of $[A \rightarrow B]$ with the identity relation on A.
- $[A (<-) B]$ is the optional version of $A \leftarrow B$.²

Restriction: A and B must denote regular languages, not relations.

²Because all the replace operators have an inverse and an optional variant we will not list them explicitly in the following sections. Optionality is indicated by parentheses around the operator, inversion by the leftward direction of the arrow.

For example, the relation $[a\ b\ c \rightarrow d\ e]$ contains the crossproduct $[a\ b\ c \cdot x \cdot d\ e]$ and infinitely many other pairs because the upper language of the relation is the universal language. The transducer that encodes the relation is unambiguous when applied downward. It maps “abcde” uniquely to “dede”. It is not unambiguous in the other direction. When applied upwards it maps “dede” to “abcabc”, “abcde”, “deabc” and “dede” because “de” in the lower language may be an “original” string or the result of a replacement.

If the B component of $[A \rightarrow B]$ denotes the empty string language, then the A substrings are in effect deleted in the lower language. If B contains more than one string, the relation is one-to-many. It may fail to be one-to-one even if B contains just one string. For example, $[a \mid a\ a \rightarrow b]$ pairs the upper-side string “aa” both with “b” and “bb”. We come back to this point below.

The replacement of a non-empty language by the empty language is in effect a prohibition of that non-empty language. For example, $[a\ b\ c \rightarrow \backslash?]$ denotes the same identity relation as $\sim\{[a\ b\ c]$, excluding all strings that contain “abc”. Except for this special case, the upper language of all \rightarrow replacements is the universal language. This holds also for expressions such as $[\backslash? \rightarrow a\ b\ c]$. If there is nothing to be replaced, replacement reduces to the universal identity relation.

If the set of strings to be replaced contains or consists of the empty string, as in $[0 \rightarrow \%+]$, the resulting transducer will contain an epsilon loop as shown in Figure 2.10.

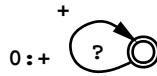


Figure 2.10: The Relation $0 \rightarrow \%+$

When applied to an input string in the downward direction, this transducer inserts arbitrarily long bursts of + signs in every position in the input word. (The notation $\%+$ denotes the literal plus sign rather than the Kleene-plus operator.) This is the correct behavior because the empty string has no length and any number of concatenations of the empty string with itself yield the empty string. Any upper-side input string will have an infinite number of lower-side outputs.

It is often desirable to define a relation that involves making just one insertion at each position in the input string. As this cannot be defined as a replacement for the empty string, we have to introduce special notation to make it possible. Dotted brackets indicate single insertion.

Single Insertion

- $[[\cdot A \cdot] \rightarrow B]$, where the input expression A is surrounded by the special “dotted brackets” $[\cdot$ and $\cdot]$, is equivalent to $[A \rightarrow B]$ if the language denoted by A does not contain the empty string. If the language of A does include the empty string, then $[[\cdot A \cdot] \rightarrow$

$B]$ denotes the relation that pairs every string of the universal language with all strings that are identical to the original except that every instance of A that occurs as a substring in the original is represented by a string from B , and all the other substrings are represented by a copy in which a string from B appears before and after every original symbol.

Restriction: A and B must denote regular languages, not relations.

Figure 2.11 shows the network compiled from $[[\cdot 0 \cdot] \rightarrow \%+]$. This transducer maps “cab” to “+c+a+b+”, inserting just a single plus sign in each original string position.

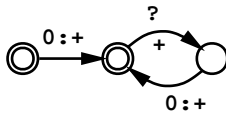


Figure 2.11: The Relation $[\cdot 0 \cdot] \rightarrow \%+$

The epsilon symbol is redundant in $[\cdot 0 \cdot]$. Just as we allow an empty pair of square brackets $[]$ as an alternate notation for the empty string, an empty pair of dotted brackets $[\cdot \cdot]$ is also accepted.

The dotted-bracket notation allows us to assign the desired interpretation to expressions such as $[[\cdot (a) \cdot] \rightarrow \%+]$, where the language denoted by the first expression consists of “a” and the empty string. The corresponding transducer replaces **a** by + and inserts a single + in all positions in the rest of the string. For example, it maps “cab” to “+c+++b+”.

The marking expression introduced below provides a general way of defining relations that change the input only by insertions of new material leaving the original parts unchanged.

Marking

- $[A \rightarrow B \dots C]$ denotes a relation in which each string of the upper-side universal language is paired with all strings that are identical to the original except that every instance of A that occurs as a substring is represented by a copy that has a string from B as a prefix and a string from C as a suffix.

The sequence of three periods is a defined operator and is required in such marking rules; it has no special meaning in other constructions. B or C can be omitted.

Restriction: A , B , and C must denote regular languages, not relations.

Expressions of the form $A \rightarrow B \dots C$ are typically used to mark or bracket instances of the language A in some special way. For example, the transducer for $[a|e|i|o|u \rightarrow \%[\dots \%]]$ maps “abide” to “[a]b[i]d[e]”, enclosing each vowel in literal square brackets.

Parallel replacement

- $[A \rightarrow B, C \rightarrow D]$ denotes the simultaneous replacement of A by B and C by D. A parallel replacement may include any number of components, separated by commas.

Restriction: A, B, and C, D, etc. must denote regular languages.

Parallel replacements allow two symbols to be exchanged for each other. For example, $[a \rightarrow b, b \rightarrow a]$ maps “baab” to “abba”, which is awkward to do with sequentially ordered rules. Similarly, $[\% \rightarrow \%, , \%, \rightarrow \%.]$ exchanges the comma and the period, mapping, for example, “1,000.0” to “1.000,0”. If the A and C components contain strings that overlap, the relation is not one-to-one. For example, $[a b \rightarrow x, b c \rightarrow y]$ maps “abc” to both “xc” and “ay” because the two replacements are made independently of one another in all possible combinations. We return to this issue below.

Conditional Replacement

All the different types of replace relations introduced in the previous section can be constrained to apply only if the string to be replaced or its replacement appears in a certain context.

Replacement contexts are specified in the same way as the contexts for restriction: $L _ R$, where L is the left context, R is the right context, and $_$ marks the site of the replacement. The boundary symbol $\#$ marks the beginning and the end of a string. Any number of contexts may be given, with a comma as the separator.

The notion of context is self-evident in the case of restriction expressions because they denote simple languages, but this is not the case for replacements that involve relations. The notation introduced below provides special symbols, $|$, $|$, $//$, \backslash , and $\backslash/$, to distinguish among four different ways of interpreting the context specification.

Conditional Replacement

The four expressions below all denote a relation that is like $[A \rightarrow B]$ except that the replacement of an original upper-side substring by a string from B is made only if the indicated additional constraint is fulfilled. Otherwise no change is made. Note that these restrictions are stated as they apply to right-arrow rules, where the upper side is naturally treated as the input side.

- $[A \rightarrow B \mid \mid L _ R]$
Every replaced substring in the upper language is immediately preceded by an upper-side string from L and immediately followed by an upper-side string from R.

- $[A \rightarrow B // L _ R]$
Every replaced substring in the upper language is immediately followed by an upper-side string from R and the lower-side replacement string is immediately preceded by a string from L.
- $[A \rightarrow B \backslash \backslash L _ R]$
Every replaced substring in the upper language is immediately preceded by an upper-side string from L and the lower-side replacement string is immediately followed by a string from R.
- $[A \rightarrow B \backslash / L _ R]$
Every lower-side replacement string is immediately preceded by a lower-side string from L and immediately followed by a lower-side string from R.

Restriction: A, B, L, and R must denote regular languages,

In other words, in $| |$ replacements both the left and right contexts are matched in the upper-language string. In $//$ replacements, the left context is matched on the lower-side and the right context on the upper side of the replacement site. In $\backslash \backslash$ replacements, conversely, the left context is matched on the upper side and the right context on the lower side. In $\backslash /$ replacements, both the left and the right context are matched on the lower side. The slant of the context separators is intended to convey where each context is matched.³

In practice, when writing right-arrow phonological/orthographical alternation rules, it is usually desirable to constrain a replacement by the original upper-side context. That is, most rules written in practice are $| |$ rules, and many users of the **Xerox** calculus have no need for the other variants. However, there are linguistic phenomena such as vowel harmony, tone spreading, and umlaut that are most naturally described in terms of a left-to-right or right-to-left process in which the result of a replacement itself serves as the context for another replacement. The two-level rule formalism discussed in Chapter 5 gives the rule writer a similar option to control the distribution of all symbols and symbol pairs in terms of both the upper and the lower context or in any combination of the two.

Figure 2.12 illustrates the difference between the $| |$ and $//$ versions of the same expression, the deletion of an initial **a**.



Figure 2.12: $[a \rightarrow 0 | | .\# . _]$ vs. $[a \rightarrow 0 // .\# . _]$

³The slant of the context operators suggests where contexts are matched in *right-arrow* rules. For left-arrow rules, where the input side is the lower side, the slant of the context operators is interpreted in the inverse way. See Section 3.5.5.

When applied downward to an input string such as “aab”, the transducer on the left in Figure 2.12 deletes only the initial **a**, yielding “ab” whereas the // version yields “b” because the deletion of the initial **a** creates the lower-side context for the deletion of the next one. Similarly, [a -> 0 || _ .#.] deletes only the final **a**, but [a -> 0 \\ _ .#.] deletes all the **as** at the end of a word like “baa”. See Section 3.5.5.

Any number of parallel replacements may be conditioned jointly by one or more contexts, as in [a -> b, b -> a || .#. _ , _ .#.] that exchanges **a** and **b** at the beginning and at the end of a word. Because the comma is used as a separator between alternate contexts, we need to introduce a second separator, a double comma ,, to make it possible to express the parallel combination of conditional replacements.

In practical morphological applications, // replacement can be useful for languages with vowel harmony whereas \\ replacement can be useful for languages with umlaut. Most applications need only || replacement.

Parallel Conditional Replacement

We give just one example to illustrate the syntax.

- [A -> B || L1 _ R1 ,, C -> D || L2 _ R2] replaces A by B in the context of L1 and R1 and simultaneously C by D in the context of L2 and R2.

Restriction: All components must denote regular languages.

As the reader must wonder what use there could possibly be for such complex expressions, we give in Figure 2.13 one example. This expression denotes the mapping from the first hundred Arabic numerals to their Roman counterparts. For example, it maps “0” to the empty string, “4” to “IV”, “44” to “XLIV”, and so on.

```
%0 -> 0 || _ (?) .#. ,,
1 -> I,      2 -> I I,      3 -> I I I,
4 -> I V,    5 -> V,        6 -> V I,
7 -> V I I,  8 -> V I I I,    9 -> I X || _ .#. ,,
1 -> X,      2 -> X X,      3 -> X X X,
4 -> X L,    5 -> L,        6 -> L X,
7 -> L X X,  8 -> L X X X,    9 -> X C || _ ? .#.
```

Figure 2.13: A Relation from Arabic to Roman Numerals

The three contexts are necessary in Figure 2.13 because zero is mapped to an epsilon everywhere, whereas the Roman value of the nine other digits depends

on their position. The corresponding transducer is unambiguous in the Arabic-to-Roman direction, but it gives multiple results in the other direction because every Roman numeral in the lower language can be interpreted as an original one or as a result of the replacement. An unambiguous bidirectional converter can be derived by composing the relation in Figure 2.13 with $\sim\$[I|V|X|L|C]$ on the upper side and with $\sim\$\[%0|1|2|3|4|5|6|7|8|9]$ on the lower side. These constraints eliminate all Roman numerals from the upper language and all Arabic numerals from the lower language, leaving the relation otherwise intact.

Directed Replacement

As we already mentioned in connection with a couple of examples, replacement relations introduced in the previous sections are not necessarily one-to-one even if the replacement language contains just one string. The transducer compiled from $[a | a a \rightarrow b]$ maps the upper language “aa” to both “b” and “bb”. The transducer for $[a b \rightarrow x, b c \rightarrow y]$ gives two results, “xc” and “ay”, for the upper-language input string “abc”.

This nondeterminism arises in two ways. First of all, possible replacements may overlap. We get a different result in the “abc” case depending on which of the two overlapping substrings is replaced. Secondly, there may be more than one possible replacement starting at the same point, as in the beginning of “aa”, where either “a” or “aa” could be replaced.

The family of directed replace operators introduced in the following section eliminates this type of nondeterminism by adding directionality and length constraints. Directionality means that the replacement sites are selected starting from the left or from the right, not allowing any overlaps. Whenever there are multiple candidate strings starting at a given location, the longest or the shortest one is selected.

Directed Replacement

The four expressions below denote a relation that is like $[A \rightarrow B]$ except that the substrings to be replaced are selected under the specified regimen.

The new replace operators can be used in all types of replace expressions introduced so far in place of \rightarrow .

- $[A @\rightarrow B]$
Replacement strings are selected from left to right. If more than one candidate string begins at a given location, only the longest one is replaced.
- $[A \rightarrow@ B]$
Replacement strings are selected from right to left. If more than one candidate string begins at a given location, only the longest one is replaced.

- $[A @> B]$
Replacement strings are selected from left to right. If more than one candidate string begins at a given location, only the shortest one is replaced.
- $[A >@ B]$
Replacement strings are selected from right to left. If more than one candidate string begins at a given location, only the shortest one is replaced.

Restriction: A and B must denote regular languages.

The additional constraints attached to the directed replace operators guarantee that any upper-language input string is uniquely factorized into a sequence of two types of substrings: the ones that are replaced and the ones that are passed on unchanged.

Figure 2.14 illustrates the difference between $[a \mid a a \rightarrow b]$ and the left-to-right, longest match version $[a \mid a a @\rightarrow b]$. It is easy to see that the transducer for the latter maps the upper language string “aa” unambiguously to “b”.



Figure 2.14: The Relation $[a \mid a a \rightarrow b]$ vs. $[a \mid a a @\rightarrow b]$

The left-to-right, longest-match replace operator $@\rightarrow$ is commonly used for text normalization, tokenization, and for “chunking” regions of text that match a given pattern. For example $[["\t" | "\n" | " "]+ @\rightarrow "\n"]$ yields a transducer that reduces a maximally long sequence of tabs, newlines, and spaces into a single newline character.

To give a simple example of chunking, let us assume that a noun phrase consists of an optional determiner, (d), any number of adjectives, a^* , and one or more nouns, n^+ . The expression $[(d) a^* n^+ @\rightarrow \%[\dots \%]]$ compiles into a transducer that inserts brackets around maximal instances of the noun phrase pattern. For example, it maps “dannvaan” into “[dann]v[aa]n”, as shown in Figure 2.15.

```

d a n n   v   a a n
-----
[ d a n n ] v [ a a n ]

```

Figure 2.15: Application of $[(d) a^* n^+ @\rightarrow \%[\dots \%]]$ to “dannvaan”

Although the input string “dannvaan” contains many other instances of the noun

phrase pattern, “dan”, “an”, “nn”, etc., the left-to-right and longest-match constraints pick out just the two maximal ones.

Directional replacement and marking expressions may be further constrained by specifying one or more contexts. For example, if C and V stand for consonants and vowels, respectively, a simple syllabification rule may be expressed as in Figure 2.16.

$$C^* V^+ C^* @-> \dots \text{"-"} \text{"|"} \text{"|"} \text{"-"} C V$$

Figure 2.16: A Simple Syllabification Rule

The marking expression in Figure 2.16 compiles into an unambiguous transducer that inserts a hyphen after each longest available instance of the $C^* V^+ C^*$ pattern that is followed by a consonant and a vowel. The relation it encodes consists of pairs of strings such as the example in Figure 2.17. The choice between $|$ and $/$ makes no difference in this case, but the two other context markers, \backslash and \wedge , would not yield the intended result here.

```

s t r u k   t u   r a   l i s   m i
s t r u k - t u - r a - l i s - m i

```

Figure 2.17: Application of $[C^* V^+ C^* @-> \dots \text{"-"} \text{"|"} \text{"|"} \text{"-"} C V]$

2.5 Properties of Finite-State Networks

In this section we consider briefly the formal properties of finite-state automata. All the networks presented in this chapter have the three important properties defined in Table 2.11.

EPSILONFREE	There are no arcs labeled with the epsilon symbol.
DETERMINISTIC	No state has more than one outgoing arc with the same label.
MINIMAL	There is no other network with exactly the same paths that has fewer states.

Table 2.11: Properties of Networks

The **Xerox** regular-expression compiler always makes sure that the result is epsilonfree, deterministic and minimal. If the network encodes a regular language, it is guaranteed that it is the best encoding for the language in the sense that any other network for the same language has the same number of states and arcs and differs only with respect to the order of the arcs, which generally is irrelevant.

The situation is more complex in the case of regular relations. Even if a transducer is epsilonfree, deterministic, and minimal in the sense of Table 2.11 there

may still be another network with fewer states and arcs for that same encoded relation. If the network has arcs labeled with a symbol pair that contains an epsilon on one side, these one-sided epsilons could be distributed differently, or perhaps even eliminated, and this might reduce the size of the network.

For example, the two networks in Figure 2.18 encode the same relation, $\{ \langle \text{“aa”}, \text{“a”} \rangle, \langle \text{“ab”}, \text{“ab”} \rangle \}$. They are both deterministic and minimal, but one is smaller than the other due to a more optimal placement of the one-sided epsilon transition.

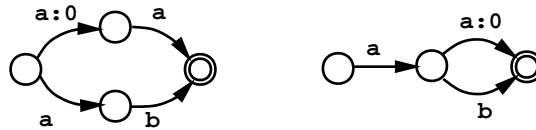


Figure 2.18: $[a:0 \ a \mid a \ b]$ vs. $[a \ [a:0 \ \mid \ b]]$

In the general case there is no way to determine whether a given transducer is the best encoding for an arbitrary relation.

For transducers, the intuitive notion of determinism makes sense only with respect to a given direction of application. But there still are two ways to think about determinism, as defined in Table 2.12.

UNAMBIGUOUS SEQUENTIAL	For any input there is at most one output No state has more than one arc with the same symbol on the input side
---------------------------	---

Table 2.12: Properties of Transducers

Although both transducers in Figure 2.18 are in fact unambiguous in both directions, the one on the left is not SEQUENTIAL in either direction. When it is applied downward, to the string “aa”, there are two paths that have to be pursued initially even though only one will succeed. The same is true in the other direction as well. In other words, there is *local* ambiguity at the start state because **a** may have to be deleted or retained. In this case, the ambiguity is resolved by the next input symbol one step later.

If the relation itself is unambiguous in the relevant direction and if all the ambiguities in the transducer resolve themselves within some fixed number of steps, the transducer is called SEQUENTIABLE. That is, we can construct an equivalent sequential transducer in the same direction (Roche and Schabes, 1997; Mohri, 1997). Figure 2.19 shows the downward sequentialized version of the leftmost transducer in Figure 2.18.

The sequentialization algorithm combines the locally ambiguous paths into a single path that does not produce any output until the ambiguity has been resolved. In the case at hand, the ambiguous path contains just one arc. When a **b** is seen, the delayed **a** is produced as output, and then the **b** itself in a one-sided epsilon

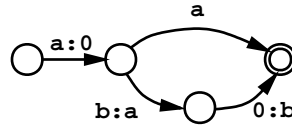


Figure 2.19: Downward Sequentialized Version of $[a:0 a \mid a b]$

transition. Otherwise, an **a** must follow, and in this case there is no delayed output. In effect the local ambiguity is resolved with one symbol lookahead.

The network in Figure 2.19 is sequential but only in the downward direction. Upward sequentialization produces the second network shown in Figure 2.18, which clearly is the best encoding for this little relation.

Even if a transducer is unambiguous, it may well be unsequentializable if the resolution of a local ambiguity requires an unbounded amount of lookahead. For example, the simple transducer for $[a^+ @ \rightarrow 0 \mid \mid b _ c]$ in Figure 2.20 cannot be sequentialized in either direction.

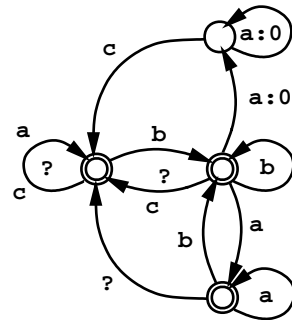


Figure 2.20: Unsequentializable Transducer $[a^+ @ \rightarrow 0 \mid \mid b _ c]$

This transducer reduces any sequence of **as** that is preceded by a **b** to an epsilon or copies it to the output unchanged, depending on whether the sequence of **as** is followed by a **c**. A sequential transducer would have to delay the decision until it reaches the end of an arbitrarily long sequence of **as**. It is obviously impossible for any finite-state device to accumulate an unbounded amount of delayed output.

However, in such cases it is always possible to split the unambiguous but non-sequentializable transducer into a BIMACHINE. A bimachine for an unambiguous relation consists of two sequential transducers that are applied in a sequence. The first half of the bimachine processes the input from left-to-right; the second half of the bimachine processes the output of the first half from right-to-left. Although the application of a bimachine requires two passes, a bimachine is in general more efficient to apply than the original transducer because the two components of the bimachine are both sequential. There is no local ambiguity in either the left-to-right or the right-to-left half of the bimachine if the original transducer is unambiguous in the given direction of application. Figure 2.21 shows a bimachine derived from

the transducer in Figure 2.20.



Figure 2.21: Bimachine for $[a^+ @ \rightarrow 0 \mid \mid b - c]$

The left-to-right half of the bimachine in Figure 2.21 is only concerned about the left context of the replacement. A string of **a**s that is preceded by **b** is mapped to a string of **a1**s, an auxiliary symbol to indicate that the left context has been matched. The right-to-left half of the bimachine maps each instance of the auxiliary symbol either to **a** or to an epsilon, depending on whether it is preceded by **c** when the intermediate output is processed from right-to-left. Figure 2.22 illustrates the application of the bimachine to three input strings, “aaa”, “baaa” and “baaac”.

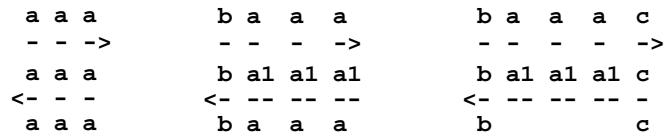


Figure 2.22: Application of a Bimachine

The bimachine in Figure 2.21 encodes exactly the same relation as the transducer in Figure 2.20. The composition of the left-to-right half L of the bimachine with the reverse of the right-to-left half R yields the original single transducer T . That is, $T = [L \circ R \cdot r]$ when the auxiliary symbols introduced in the bimachine factorization are removed from the sigma alphabet.

2.6 Exercises

Construction of finite-state networks by hand, as we did in the exercises of Chapter 1, is a tedious affair and very error-prone except for the most trivial cases. With the help of the regular-expression calculus we can give a precise specification of the language or the relation we wish to construct and get the regular-expression compiler do the rest of the work for us.

As a warm-up exercise, let us first construct a regular expression that denotes the language of the Simple Cola Machine in Chapter 1, redrawn in Figure 2.23.

The strings of the language are made up of three symbols, **N** (nickel = 5 cents), **D** (dime = 10 cents), and **Q** (quarter = 25 cents). The language consists of strings whose value is exactly 25 cents or, equivalently, five nickels.

The simplest method to construct the network in Figure 2.23 is to take advantage of the fact that the value of each symbol in the language can be expressed in

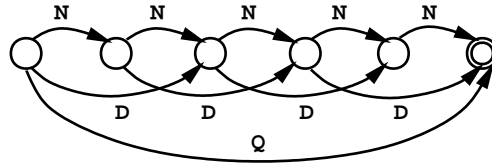


Figure 2.23: The Simple Cola Machine

terms of nickels. A dime is two nickels, a quarter is five nickels. We can encode these facts as a relation from higher-valued coins to the equivalent sequence of nickels: $[D \rightarrow N^2, Q \rightarrow N^5]$. The upper language of this relation is the universal language in which the three symbols **N**, **D**, and **Q** occur in any combination any number of times along with any other symbols.

But for our Cola Machine, we are only interested in sequences whose value adds up to five nickels and there should be no other symbols. To derive that sublanguage, we simply compose the lower side of the relation with $[N^5]$ and extract the upper side of the result. The expression in Figure 2.24 compiles exactly into the network displayed in Figure 2.23.

$$[[D \rightarrow N^2, Q \rightarrow N^5] \circ N^5].u$$

Figure 2.24: A regular expression for the Simple Cola Machine

2.6.1 The Better Cola Machine

Modify the regular expression in Figure 2.24 so that it compiles into a transducer that maps any sequence of coins whose value is exactly 25 cents into the multicharacter symbol **[COLA]**.

Having succeeded in that task, modify the expression once more to describe a vending machine that accepts any sequence of coins and outputs “[COLA]” every time when the value is equal to a multiple of 25 cents.

Finally, make the machine honest. Any extra money should be returned to the customer. For example, the sequence “QDDNNN” should map into two colas and a dime: “[COLA][COLA]D”. If the customer does not insert enough money for a cola, the money should be returned; that is, the transducer should map a string such as “DN” into itself or into some equivalent sequence of coins such as “NNN”.

A solution to this problem is given in Appendix D, page 621. It is of course not the only one. There are infinitely many regular expressions that denote any given regular relation.

Chapter 3

The `xfst` Interface

Contents

3.1	Introduction	84
3.1.1	What is <code>xfst</code> ?	84
3.1.2	Getting Started	84
	Invoking <code>xfst</code>	84
	Help	85
	Apropos	85
	The Read-Eval-Print Loop	87
	Exiting from <code>xfst</code>	87
3.2	Compiling Regular Expressions	87
3.2.1	<code>xfst</code> Regular Expressions	87
3.2.2	Basic Regular-Expression Compilation and Testing	87
	<code>define</code>	88
	<code>read regex</code>	88
	Basic Network Testing	89
	Basic Pushing and Popping	90
3.2.3	Basic Regular Expressions Denoting Languages	91
3.2.4	Basic Stack Operations	100
	Understanding The Stack	100
	Pushing and Popping	101
	Pushing and Popping Defined Variables	104
3.2.5	Basic Regular Expressions Denoting Relations	106
	<code>crossproduct</code>	106
	Experimenting with the <code>crossproduct</code> Operation	107
	Projections	110
	Orientations	110
3.3	Interacting with the Operating System	112

3.3.1	File I/O, Defined Variables and The Stack	112
	Regular Expression Files	112
	Binary Files and The Stack	113
	Binary Files and Defined Variables	114
	Wordlist or Text Files	115
	Spaced-Text Files	118
	lexc Files	120
	Prolog Files	120
3.3.2	Referring to Binary Files within Regular Expressions .	120
3.3.3	Scripts in xfst	121
	Script Files	121
	echo	122
3.3.4	Tokenized Input Files for apply up and apply down .	123
3.3.5	System Calls	124
3.4	Incremental Computation of Networks	125
3.4.1	Modularization	125
3.4.2	Computing with Defined Variables	125
3.4.3	Computing on The Stack	127
	Stack-Based Finite-State Algorithms	127
	Reordering Networks on The Stack	132
3.5	Rule-Like Notations	134
3.5.1	Context Restriction	134
3.5.2	Simple replace rules	136
3.5.3	Rule Ordering and Composition	141
	The <i>kaNpat</i> Exercise	141
	The Linguistic Phenomena to Capture	141
	A replace rule Grammar for <i>kaNpat</i>	142
	Testing Your <i>kaNpat</i> Grammar	143
	Experiments	145
	Putting replace rules in Context	145
	Parallel Rules	147
	Ordering Rule Networks on The Stack	148
3.5.4	More Exercises	149
	Southern Brazilian Portuguese Pronunciation Exercise	149
	Capturing Phonemics vs. Orthography	149
	The Facts to be Modeled	150
	Testing Portuguese Pronunciation	155
	The Bambona Language Exercise	155
	Some Preliminaries	155
	The Facts	157

	Compose the Rules under the Lexicon	163
	Bambona Using a Single Script File	165
	The Monish Language Exercise	166
	The Monish Guesser	172
3.5.5	More Replace Rules	173
	Right-Arrow Rules	173
	Double-Vertical-Bar Rules	173
	Double-Slash Rules	173
	Double-Backslash Rules	177
	Longest Match	177
	Shortest Match	178
	Backslash-Slash Rules	178
	Epenthesis Rules	179
	Bracketing or Markup Rules	180
	Left-Arrow Rules	184
	Double-Arrow Rules	186
3.6	Examining Networks	186
3.6.1	Boolean Testing of Networks	187
	N-ary Boolean Tests	187
	Unary Boolean Testing	187
3.6.2	Printing Words and Paths	187
	Print Commands	187
	Variables Affecting Print Commands	188
3.6.3	Alphabets of a Network	189
	Printing Alphabets	189
	Symbol Tokenization of Input Strings	191
3.6.4	Printing Information about Networks	192
3.6.5	Inspection of Networks	194
3.7	Miscellaneous Operations on Networks	194
3.7.1	Substitution	194
	Substitution Commands	194
	Substitution Applications	196
	Shuffling	196
	Adding Roots to a Network	197
3.7.2	Network Properties	198
3.7.3	Housekeeping Operations	199
3.7.4	Generation of Derived Networks	199
3.8	Advanced <code>xfst</code>	200
3.8.1	Word-Number Mapping	200
3.8.2	Modifying the Behavior of <code>xfst</code>	201

3.8.3	Command Abbreviations	202
3.8.4	Command Aliases	204
3.8.5	xfst Command-Line Options	204
3.9	Operator Precedence	206
3.10	Conclusion	206

3.1 Introduction

3.1.1 What is *xfst*?

This chapter is a hands-on tutorial on **xfst**, a general-purpose interactive utility for creating and manipulating finite-state networks. You should have the **xfst** application installed and running in front of you as you work your way through the examples and exercises. We will review the various regular-expression notations, already introduced in Chapter 2, while learning about the **xfst** interface and its various commands.

xfst provides a compiler for regular expressions, creating new networks according to your specifications. It can also read in and manipulate networks previously compiled by **lexc** (Chapter 4) and **twolc** (Chapter 5). You can also, from within **xfst**, invoke the **lexc** compiler on a **lexc** source file, and there are auxiliary compilers for wordlists and other text-like formats. Where necessary, **xfst** gives you direct access to the algorithms of the Finite-State Calculus, including concatenation, union, intersection, composition, complementation, etc.

3.1.2 Getting Started

We introduce **xfst** with a brief tour that shows how the application is launched, queried and exited in a UNIX-like operating system.

Invoking *xfst*

The basic way to invoke **xfst** is simply to enter `xfst` at the UNIX command-line prompt.

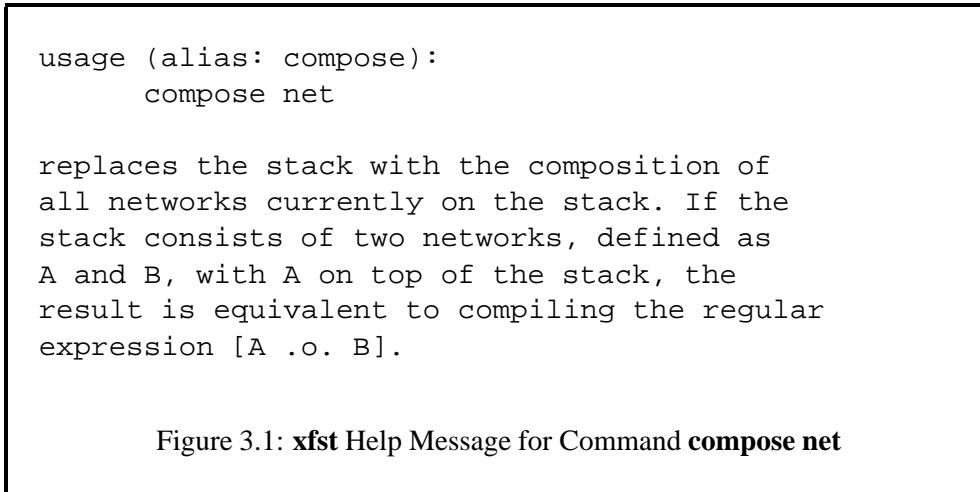
```
unix> xfst
```

xfst will respond with a welcome banner and an **xfst** prompt, and it then waits for you to type a command after the prompt.

```
Copyright © Xerox Corporation 1997-2002
Xerox Finite-State Tool, version 7.9.0
```

```
Enter "help" to list all commands available
or "help help" for further help.
```

```
xfst[0]:
```



The zero in the prompt (`xfst[0]`) will be explained later, as will command-line flag options available when invoking **xfst**.

Help

If you enter `help` or a question mark (?) at the **xfst** prompt, **xfst** will display a long menu of available commands (Table 3.1), organized into subclasses. Don't be intimidated by the wealth of choices; you will need only a few commands to get started.

```
xfst[0]: help
```

For short documentation on a particular command, e.g. **compose net**, enter `help` followed by the name of that command.

```
xfst[0]: help compose net
```

xfst will respond with a useful summary (Figure 3.1) of the usage and semantics of the command.

Apropos

If you don't know the exact format of the command you need, the **apropos** command can be very useful. For example to find information about commands that involve the `compose` operation, enter `apropos compose` and **xfst** will return a list of relevant commands and switches as shown in Figure 3.2.

```
xfst[0]: apropos compose
```

```

COMMAND CLASS: System commands

alias, quit, set, show, source, system

COMMAND CLASS: Input/output and stack commands

clear stack, define, list, load defined, load stack, pop stack,
push defined, read lexc, read prolog, read regex, read spaced-
text,
read text, rotate stack, save defined, save stack, turn stack,
undefine, unlist, write prolog, write spaced-text, write text,

COMMAND CLASS: Display commands

apply up, apply down, apropos, echo, help, inspect net,
print aliases, print defined, print directory, print file-info,
print flags, print labels, print label-tally, print lists,
print longest-string, print longest-string-size,
print lower-words, print name, print net, print nth-lower,
print nth-upper, print num-lower, print num-upper,
print random-lower, print random-upper, print random-words,
print sigma, print sigma-tally, print sigma-word-tally,
print size, print stack, print upper-words, print words,
write properties

COMMAND CLASS: Tests of network properties

test equivalent, test lower-bounded, test lower-universal,
test non-null, test null, test overlap, test sublanguage,
test upper-bounded, test upper-universal

COMMAND CLASS: Operations on networks

add properties, cleanup net, compact sigma, compile-
replace lower,
compile-replace upper, complete net, compose net, concate-
nate net,
crossproduct net, determinize net, edit properties,
eliminate flag, epsilon-remove net, intersect net, invert net,
label net, lower-side net, minimize net, minus net, name net,
negate net, one-plus net, prune net, read properties, re-
verse net,
shuffle net, sigma net, sort net, substitute defined,
substitute label, substitute symbol, substring net, union net,
upper-side net, zero-plus net

```

Table 3.1: The Command Menu from **xfst**

```
compose net
  compose the finite-state networks in the stack
variable flag-is-special
  when ON, composition treats flag diacritics as epsilons
variable recursive-apply
  when ON, compose apply works depth-first
regex: compose net
  regular expression [A .o. B]
```

Figure 3.2: Response from **apropos compose**

The Read-Eval-Print Loop

The **xfst** interface is a read-eval-print loop; it reads your command typed manually at the prompt, executes it, and then displays a response and a new prompt. As we shall see later, it is also possible to direct **xfst** to perform a series of commands that were previously edited and stored in a file called a **SCRIPT** (see Section 3.3.3).

Exiting from **xfst**

To exit from the **xfst** interface and return to the host operating system, simply invoke the **exit** command or the **quit** command.

```
xfst[0]: exit
```

```
xfst[0]: quit
```

3.2 Compiling Regular Expressions

3.2.1 **xfst** Regular Expressions

xfst includes a compiler for the **Xerox** regular-expression metalanguage, which includes many useful operators that are not available in other implementations. Traditional textbook notations have been modified where necessary to allow them to be typed as linear strings of ASCII characters. Readers who are already acquainted with regular expressions and want to get started quickly should review the syntax already presented in Section 2.3.1 and consult the online regular-expression summaries available from <http://www.fsmbook.com/>.

3.2.2 Basic Regular-Expression Compilation and Testing

This section is intended for those who are already fairly comfortable with the **Xerox** regular-expression notation and want to jump into compilation and test-

ing. Later we will review, more slowly and incrementally, the various regular-expression operators and **xfst** commands available.

define

There are two **xfst** commands, **define** and **read regex**, that invoke the regular-expression compiler. The schema for the **define** command is the following:

```
xfst[0]: define variable regular-expression ;
```

The variable name is chosen by the user. The regular expression must start on the same line, but it can be arbitrarily complicated and can extend over multiple lines. The regular expression must be terminated with a semicolon followed by a newline (carriage return).

The effect of such a command, e.g.

```
xfst[0]: define MyVar [ d o g | c a t | h o r s e ] ;
```

is to invoke the compiler on the regular expression, create a network, and assign that network to the indicated variable, here **MyVar**. Once defined in this way, the variable can be used in subsequent regular expressions.

The **define** command in fact comes in two easily confusable variants. When you call **define** to compile a regular expression, as in the examples just shown, that regular expression must begin on the same line as the **define** command. The other variant of the **define** command will be shown below.

read regex

The other compiler command, **read regex**, similarly reads and compiles a regular expression, but the resulting network is added or PUSHED onto a built-in STACK.

```
xfst[0]: read regex regular-expression ;
```

We will have much more to say about The Stack below, but for now it suffices to know that many **xfst** operations refer by default to the top network on The Stack, or they POP their arguments from The Stack and push the result back onto The Stack.

The **xfst** prompt includes a counter indicating how many networks are currently stored on The Stack.

If we perform the following command:

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
10 states, 11 arcs, 3 paths.
xfst[1]:
```


and have not made any mistakes in typing the regular expression, **xfst** will respond with a message indicating the number of states, arcs and paths in the resulting network. The digit **1** printed in the new prompt indicates that there is one network saved on The Stack.

Because they can extend over multiple lines, regular expressions inside **define** and **read regex** commands must be explicitly terminated with a semicolon followed by a newline (carriage return). Other commands not containing regular expressions cannot extend over multiple lines and are not terminated with a semicolon.

Basic Network Testing

Once we have, on the top of The Stack, a network that encodes a language, and if the language is finite, we can then use the **print words** command to have the language of the network enumerated on the terminal.

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
10 states, 11 arcs, 3 paths.
xfst[1]: print words
dog
cat
horse
```

Other useful commands are **apply up**, which performs ANALYSIS or LOOKUP, and **apply down**, which performs GENERATION, also known as LOOKDOWN; both **apply up** and **apply down** use the network on the top of The Stack. In the formal terminology of networks, **apply up** “applies the network in an upward direction” to an input string, matching the input against the lower side of the network and returning any related strings on the upper side of the network. The following example looks up the word “dog” and returns “dog” as the response, indicating success. Note that it is not necessary to recompile the regular expression if the network is already compiled and on the top of The Stack.

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
xfst[1]: apply up dog
dog
xfst[1]:
```

Any attempt to look up a word not in the language encoded by the network will result in failure, and **xfst** will simply display a new prompt.

```
xfst[1]: apply up elephant
xfst[1]:
```

If you are hand-testing a number of input words, you may soon tire of retyping **apply up** over and over again. It happens that the **apply up** command, like most commands in **xfst**, has an abbreviation, in this case **up**.

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
xfst[1]: up dog
dog
xfst[1]:
```

In addition, there is an **APPLY UP MODE** that you invoke when you enter **apply up**, or just **up**, without an argument. The new prompt becomes `apply up>`, and you simply enter input strings one at a time at the prompt. To exit this apply-up mode and return to the **xfst** prompt, enter `END;`, all in capital (upper-case) letters, and including the semicolon. On a Unix-like system, you can also escape apply-up mode by entering Control-D. In a Windows system, you can type Control-Z.

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
xfst[1]: apply up
apply up> dog
dog
apply up> cat
cat
apply up> elephant
apply up> END;
xfst[1]:
```

Finally, if you have a list of input words stored in a file, with one word per line, you can analyze the whole file by entering **apply up**, followed by a left angle bracket (<) and the name of the file. As usual, **apply up** will apply the network on the top of The Stack.

```
xfst[1]: apply up < filename
```

Basic Pushing and Popping

There are a number of commands for manipulating The Stack itself, but for now we'll introduce just two. Use the **pop stack** command to pop the top network off The Stack and discard it. The counter in the prompt will decrement by one.

```
xfst[1]: pop stack
xfst[0]:
```

Use the **clear stack** command to pop all the networks off of The Stack, leaving it empty. The counter in the new prompt will be 0.

```
xfst[4]: clear stack
xfst[0]:
```

Be sure to distinguish commands from regular expressions. You type commands, including the **help**, **apropos**, **define**, **read regex**, **pop stack**, **clear stack**, **apply up** and **apply down** commands, to the **xfst** interface. Only two of these commands, **define** and **read regex**, can *include* a regular expression as part of the command. You must therefore learn the language of commands for running **xfst** and the metalanguage of regular expressions, which has its own syntax.

3.2.3 Basic Regular Expressions Denoting Languages

Through the rest of this chapter we will review the regular-expression metalanguage progressively, presenting examples and introducing useful **xfst** commands as we go. We start with the basic SYMBOL notations shown in Table 3.2. Every basic alphabetic symbol is, by itself, a valid regular expression. Case is significant everywhere in **xfst**, so **z** is a separate symbol from **Z**.

a	Alphabetic characters like a, b, c, etc.
"a"	Double-quoted normal alphabetic letter like "a" is equivalent to a.
"+"	A literal plus sign symbol; i.e. not a Kleene star.
%+	A literal plus sign; alternate notation.
"\o" "\oo" "\ooo"	A symbol expressed as an octal value, where o is a digit 0-7, e.g. "\123".
"\xHH"	A symbol expressed as a hexadecimal (hex) value, where H is 0-9, a-f or A-F, e.g. "\xA3".
"\uHHHH"	A 16-bit Unicode character, e.g. "\u0633".
"\n"	Newline symbol in escape notation as per Unix convention. Also \t (tab), \b (backspace), \r (carriage return), \f (formfeed), \v (vertical tab), and \a (alert).

Table 3.2: Regular-Expression Notations for Basic Symbols

Note that the backslash notations for symbols, e.g. `\142`, `\x63` and `\u0633`, and the special control characters like `\n` and `\t`, can appear only inside double quoted strings.

The simplest regular expression is therefore just a symbol like `a`, which by itself denotes the language consisting of the single string “a”. You compile such an expression using the **read regex** command or the **define** command. Like all regular expressions in **xfst**, it must be terminated with a semicolon and a newline.

```
xfst[0]: clear stack
xfst[0]: read regex a ;
2 states, 1 arc, 1 path.
xfst[1]: print words
a
```

xfst responds with a short message indicating the number of states, arcs and paths in the resulting network, and then displays a new prompt. The **1** in the new prompt indicates that there is one network stored on The Stack.

In addition to basic symbols, **xfst** regular expressions also allow MULTICHARACTER SYMBOLS, which have multicharacter print names but which are manipulated in networks exactly like all the other symbols. Multicharacter symbols are not declared in **xfst** but are simply written in regular expressions with no spaces between the letters of the name, as shown in Table 3.3. Any multicharacter symbol is, by itself, a valid regular expression.

A trivial regular expression consisting of a single multicharacter symbol is shown here:

```
xfst[0]: read regex "+Noun" ;
2 states, 1 arc, 1 path.
xfst[1]: print words
+Noun
```

Three special symbol-like notations are listed in Table 3.4; we will encounter more later when dealing with restrictions and replace rules. **xfst** uses `0` and `[]` to denote the empty string because the traditionally used epsilon symbol (ϵ) is not available on ASCII keyboards. In regular expressions the question mark denotes any single symbol, including all possible non-alphabetic and multicharacter symbols.

The finite-state grouping, iteration and optionality operators are shown in Table 3.5. In this table, *A* represents an arbitrarily complex regular expression.

Do not confuse square brackets, used for grouping, with parentheses, which denote optionality.

cat	Compiled as the single multicharacter symbol cat . Users are advised <i>not</i> to define multicharacter symbols like cat , consisting of only normal alphabetical characters, because they are too easy to confuse with the concatenation of three separate symbols c , a and t .
" +Noun "	Compiled as the single multicharacter symbol +Noun. The surrounding double quotes cause the plus sign to be treated as a literal letter, i.e. not the Kleene plus. Users are encouraged to include a punctuation symbol in the spelling of multicharacter symbols; this helps distinguish them visually from concatenations of alphabetic characters.
%+Noun	Another way to notate the multicharacter symbol +Noun. The percent sign literalizes the plus sign.
%^HIGH	The multicharacter symbol ^HIGH, starting with a literal circumflex.
% [HIGH %]	The multicharacter symbol [HIGH], starting with a literal left square bracket and ending with a literal right square bracket.
" [HIGH] "	Another way to notate the multicharacter symbol [HIGH].

Table 3.3: Notations for Multicharacter Symbols

Consider the regular expression `a+`, which denotes the infinite language of all strings consisting of one or more **a**s: “a”, “aa”, “aaa”, “aaaa”, etc. Such an expression can be compiled and tested like any other; **apply up** will succeed for strings like “aa” and “aaaaaaaa” but fail for any string that is not composed exclusively of one or more **a** symbols.

```

xfst[0]: read regex a+ ;
xfst[1]: apply up a
a
xfst[1]: apply up aa
aa
xfst[1]: apply up aaaaaaaaaa
aaaaaaaaaa
xfst[1]: apply up aaaab
xfst[1]:

```

The language denoted by `a*` includes all the words in `a+` plus the empty string (epsilon). Optionality is indicated by surrounding an expression with parentheses, as in `(a)`. The numbered iteration operators have many flavors, the simplest being

?	Denotes ANY symbol.
0	Denotes the empty (zero-length) string, also called epsilon.
[]	Denotes the empty string; equivalent to 0.

Table 3.4: Special Symbol-Like Notations

[]	Grouping brackets. [A] is equivalent to A.
A*	The Kleene star. Denotes zero or more concatenations of A with itself.
A+	The Kleene plus. Denotes one or more concatenations of A with itself. A+ is equivalent to [A A*].
(A)	Optional. (A) is equivalent to [A 0].
A ⁿ	Where <i>n</i> is an integer, denotes <i>n</i> concatenations of A with itself.
A ^{n, m}	Where <i>n</i> and <i>m</i> are integers, denotes <i>n</i> to <i>m</i> concatenations of A with itself.
A ^{<n}	Where <i>n</i> is an integer, denotes fewer than <i>n</i> concatenations of A with itself.
A ^{>n}	Where <i>n</i> is an integer, denotes greater than <i>n</i> concatenations of A with itself.

Table 3.5: Grouping, Iteration and Optionality

exemplified by a^5 , which denotes the language consisting of the single string “aaaaa”.

The basic traditional finite-state operations include union, intersect, minus (subtraction), complement (negation) and concatenation. Table 3.6 shows how these are notated in **xfst** regular expressions. In the table, A and B denote arbitrarily complex regular expressions.

Note in particular the operation of concatenation, which has no explicit operator. Symbols to be concatenated are simply typed one after another, separated by whitespace. The expression [c a t], for example, denotes the string “cat”, which consists of the three separate symbols **c**, **a**, and **t**.

Compile and test, using **apply up**, the regular expression [a b]*, which denotes the infinite language of strings containing zero or more concatenations of “ab”. Don’t neglect to put a space between the **a** and the **b** in the regular expression or you will inadvertently be defining a new multicharacter symbol **ab**, which is not what you want here.

$A \mid B$	The union of A and B.
$A \ \& \ B$	The intersection of A and B. Here A and B must denote languages, not relations.
$A - B$	The subtraction of B from A. Here A and B must denote languages, not relations.
$\sim A$	The language complement of A, i.e. $[?^* - A]$. Here A must denote a language, not a relation.
$A \ B$	The concatenation of B after A. The operands are separated by white space; there is no explicit concatenation operator.
$\{cat\}$	Compiled as a concatenation of the three symbols c , a and t . Surrounding a string of symbols with curly braces “explodes” them into separate symbols, here equivalent to $[c \ a \ t]$.

Table 3.6: Regular-Expression Notations for Basic Symbols

When one or both of the operands to be concatenated are themselves delimited, then no separating white space is necessary; for example

$[l \ o \ o \ k] \ [i \ n \ g]$

and

$[l \ o \ o \ k][i \ n \ g]$

are equivalent expressions, both denoting the concatenation of the language containing the string “look” with the language containing the string “ing”, to denote a new language containing the string “looking”.

The curly braces in expressions like $\{dog\}$ are sometimes called “explosion” braces because they tell the compiler to “explode” the contained string into separate symbols rather than treating them as the name of a multicharacter symbol; the regular expression $\{dog\}$ is equivalent to $[d \ o \ g]$. In practice it is often more convenient to type and read $\{elephant\}$ than $[e \ l \ e \ p \ h \ a \ n \ t]$, especially if there are many such strings to type.

<code>?*</code>	Denotes the universal language, which contains all possible strings, including the empty string (epsilon).
<code>~[?*]</code>	One of an infinite number of ways to denote the empty language, which contains no strings at all.
<code>[a - a]</code>	Another way to denote the empty language.

Table 3.7: The Universal Language and the Empty Language

Warning: It is all too easy to inadvertently write a multicharacter symbol, e.g. `cat`, in a regular expression when you really intended to write `c a t` or `{cat}`, indicating the concatenation of three separate symbols **c**, **a** and **t**. By **Xerox** convention, multicharacter symbols include a non-alphabetic character like the plus sign, e.g. `+Noun`, or the circumflex, e.g. `^HIGH`, or they surround a name in punctuation, e.g. `[Noun]` or `[Verb]`, to help them stand out and to avoid visual confusion with simple concatenations of alphabetic characters.

From these regular-expression notations and others to come, you will note that almost all the ASCII punctuation characters have been pressed into service as operators. The punctuation characters are therefore special characters in **xfst**. To notate literal plus signs, asterisks, vertical bars, etc. they must be preceded with the literalizing percent sign, as in `%+`, `%*` and `%|`, or included inside double quotes, as in `"+"`, `"*"` and `"|"`.

Table 3.7 illustrates expressions that denote the universal language and the empty language. The regular expression `?*` denotes the universal language, which contains all possible strings, of any length, including the empty string. The empty language, which contains no strings at all, not even the empty string, can be notated many ways, including `~[?*]` (i.e. the complement of the universal language), `[a - a]`, `[b - b]`, etc.

The union operator `|` constructs a regular language or relation that contains all the strings (or ordered string pairs) of the operands; e.g. `a | b` denotes the language that contains the string "a" and "b". Square brackets can be used freely to group expressions, e.g. `[c a t | d o g] - [d o g]`, either to make expressions easier for people to read or to force the compiler to perform certain operations before others. By convention, concatenation has higher precedence than union. The precedence of the finite-state operators is described below on page 206.

Using these operators, which are already familiar to most programmers, we can start to write more interesting regular expressions and experiment with the compiled networks. Follow along with the following examples.


```
xfst[0]: read regex d o g | c a t | h o r s e ;
10 states, 11 arcs, 3 paths.
xfst[1]:
```

This regular expression denotes a language containing just the three words “dog”, “cat” and “horse”. Once the corresponding network is compiled and pushed on the top of The Stack, we can cause **xfst** to enumerate the words of the language with the command **print words**.

```
xfst[1]: print words
dog
cat
horse
xfst[1]:
```

The following expression subtracts one language of strings from another. Compile it and use **print words** to see the words in the resulting language.

```
xfst[0]: read regex [ d o g | c a t | r a t |
e l e p h a n t ] - [ d o g | r a t ] ;
10 states, 10 arcs, 2 paths.
xfst[1]: print words
cat
elephant
```

Also compile and test the expression

```
(r e)[[m a k e] | [c o m p i l e]]
```

which denotes a language containing the words “make”, “remake”, “compile” and “recompile”. Remember to separate the symbols with spaces to avoid creating unwanted multicharacter symbols.

Similarly, use **read regex** to compile the following regular expression, which denotes an intersection of two small languages, and then enumerate the words of the language using **print words**. In the **read regex** command, remember to terminate the regular expression with a semicolon and a newline.

```
[ d o g | c a t | r a t | e l e p h a n t | p i g ]
& [ c a t | a a r d v a r k | p i g ]
```

You should find that the resulting language contains just two words, “cat” and “pig”. If you get tired of spacing out all the letters, try the alternative notation using curly braces.

```
[ {dog} | {cat} | {rat} | {elephant} | {pig} ]
& [ {cat} | {aardvark} | {pig} ]
```

Try to remember *not* to type just plain `dog`, without curly braces, if you really intend `[d o g]` or `{dog}`. Plain `dog` will be compiled as a single multicharacter symbol, which is quite different from a concatenation of three separate symbols. Unwanted multicharacter symbols can come back to haunt you later with mysterious problems.

The intersection operator `&` operates only on networks encoding languages (not on transducers) and constructs a language that contains all the strings common to the operands.

The two complement operators, shown in Table 3.8, deserve some special attention because learners very often confuse them. Where A is an arbitrarily complicated regular expression denoting a language, the *language* complement, e.g. $\sim A$, denotes the universal language (all possible strings, of any length) minus all the strings in A ; $\sim A$ is therefore equivalent to `[?* - A]`. If language A does not contain the empty string, then $\sim A$ will contain the empty string, as well as all other possible strings not in A .

In contrast, the *symbol* complement, e.g. $\backslash A$, denotes the language of all *single-symbol* strings, minus the strings in A . The most typical uses of the symbol complement operator involve a language A which denotes either a single-symbol string or a union of single-symbol strings. Thus we typically see practical examples like $\backslash z$, which denotes the language of all single-symbol strings “a”, “b”, “c”, “d”, etc., except for the string “z”; or $\backslash [a|b|c]$, which denotes the language of all single-symbol strings except for the strings “a”, “b” and “c”.¹ From the matching point of view, an expression like $\backslash a$ must match a single symbol and can never match the empty string or a string of length greater than one.

$\sim A$	The language complement of A , i.e. <code>[?* - A]</code> . Here A must denote a language, not a relation.
$\backslash A$	The symbol complement of A , i.e. <code>[? - A]</code> . Thus $\backslash a$ contains “b”, “c”, “d”, etc., but not “a” or the empty string or any string of length greater than one.
$\backslash ?$	Another notation for the null language.

Table 3.8: Language Complement vs. Symbol Complement

The semantics and compilation of the complement operators are discussed in detail in Sections 2.3.1 and 2.3.4 and will not be repeated here. If A denotes a finite

¹Programmers familiar with regular expressions in **Perl** and **emacs** may compare the **xfst** $\backslash z$ notation to the equivalent **Perl/emacs** `[^z]` notation, and $\backslash [a|b|c]$ to the **Perl/emacs** `[^abc]`.

language, then $\sim A$ denotes an infinite language that cannot be enumerated. **xfst** calls such networks “Circular”, indicating that while they contain a finite number of states and arcs, the arcs include loops that result in the network containing an infinite number of possible paths.

```
xfst[0]: read regex ~[c a t] ;
5 states, 20 arcs, Circular.
xfst[1]:
```

In this example, the language being denoted is the complement of the language containing the single word “cat”, i.e. the universal language minus the language containing just “cat”. This is therefore an infinite language, and the network that encodes it is circular. The **print words** command will print out a few examples but has special logic that keeps it from trying to display an infinite number of words. The **print random-words** command will print a random selection of words from the language.

More useful in such cases is the **apply up** command that looks up (analyzes) an input word using the network on top of The Stack. In the case of $\sim[c a t]$, **apply up** will fail for the input word “cat” and succeed for all other words, including “cats”, “dog”, “hippopotamus”, “monkey” and “wihuiehgwe”. When **apply up** succeeds, one or more strings are returned; when **apply up** fails, it returns nothing, and **xfst** simply displays a new prompt.

```
xfst[0]: read regex ~[c a t] ;
5 states, 20 arcs, Circular.
xfst[1]: apply up cat
xfst[1]: apply up cats
cats
xfst[1]: apply up dog
dog
xfst[1]: apply up hippopotamus
hippopotamus
xfst[1]: apply up monkey
monkey
xfst[1]: apply up wihuiehgwe
wihuiehgwe
```

The CONTAIN and IGNORE operations, defined in Section 2.3.1, are summarized in Table 3.9. Try compiling and testing the regular expression $\$a$, which denotes the language of all strings that contain at least one **a**, including “a”, “ab”, “zzzzza”, “alphabet”, etc. Also compile and test $a+/b$, which denotes the language of all strings consisting of one or more **as**, i.e. “a”, “aa”, “aaa”, etc., plus all these strings with any number of **b** symbols appearing as noise: “ab”, “bbbab”, “bababab”, “aabbbbab”, etc.

$\$A$	Denotes the language of all strings (or the relation of all ordered string pairs) that contain A, i.e. $[?^* A ?^*]$.
$\$.A$	Denotes the language of all strings that contain exactly one occurrence of a string from A.
$\$?A$	Denotes the language of all strings that contain at most one occurrence of a string from A.
A/B	Denotes the language A, ignoring interspersed “noise” strings from B. In other words, A/B denotes the set of all strings that would be in A if all substrings from B, considered as a kind of “noise”, were removed.
$A. / .B$	Denotes the language A, ignoring <i>internally</i> interspersed “noise” strings from B.

Table 3.9: Contain and Ignore

3.2.4 Basic Stack Operations

Understanding The Stack

Before continuing with our review of regular-expression operators, let’s look more closely at the built-in stack used by **xfst** to store and manipulate networks. The Stack is a last-in, first-out (LIFO) data structure that stores networks, and when **xfst** is launched, The Stack is empty. Networks can be PUSHED onto the top of The Stack, where they are added on top of any previously pushed networks. Networks can also be POPPED or taken off the top of The Stack.

Last-in, first-out (LIFO) stacks are often compared to the spring-loaded pop-up stacks of plates found in many cafeterias. When you take a plate, you take it from the very top of The Stack; and any newly washed plates are added or “pushed” back onto The Stack from the top. Plates in the middle or at the bottom of The Stack are inaccessible until the plates above them have been “popped” off.

Many **xfst** commands refer to The Stack in some way, usually popping their arguments one at a time from the top of The Stack, computing a result, and then pushing the result back onto The Stack. Other commands, like **print words**, **apply up** and **apply down**, refer by default to the topmost network on The Stack without actually popping or modifying it. Users of **xfst** must constantly be aware of what is on The Stack. Luckily, there are a number of commands to help visualize and manipulate The Stack.

Pushing and Popping

We have already seen that the **read regex** command reads a regular expression, compiles it into a finite-state network, and pushes the result onto The Stack. For the user's information, the number in the **xfst** prompt indicates at each step how many networks are currently on The Stack. In the following example session, three networks are compiled and pushed successively on The Stack.

```
xfst[0]: clear stack
xfst[0]: read regex a ;
2 states, 1 arc, 1 path.
xfst[1]: read regex c a t ;
4 states, 3 arcs, 1 path.
xfst[2]: read regex [ {dog} | {cat} | {elephant} ] ;
12 states, 13 arcs, 3 paths.
xfst[3]:
```

In this case, the resulting stack, pictured in Figure 3.3, will have the last-pushed network, the one corresponding to `[{dog} | {cat} | {elephant}]`, on the top, with the network for `{cat}` underneath it, and the network for `a` on the very bottom. Try typing in the same commands yourself, recreating The Stack state shown in Figure 3.3. Continuing the same session, if we invoke **print words** it will refer by default to the top network on The Stack. The other networks below the top are temporarily inaccessible.

```
xfst[3]: print words
elephant
dog
cat
```

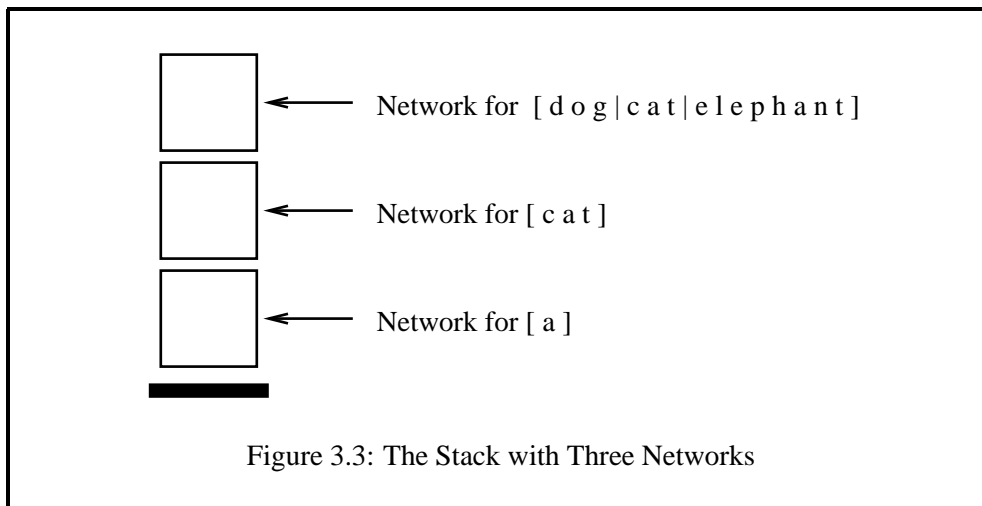


Figure 3.3: The Stack with Three Networks

To see what is on The Stack at any time, invoke the **print stack** command:

```
xfst[3]: print stack
0: 12 states, 13 arcs, 3 paths.
1: 4 states, 3 arcs, 1 path.
2: 2 states, 1 arc, 1 path.
xfst[3]:
```

print stack displays the size of each network on The Stack, starting at the top network, which it numbers 0.

To see detailed information about a particular network, including its sigma alphabet (see Section 3.2.4) and a listing of states and arcs, use the **print net** command. If it is invoked without an argument, **print net** by default displays information about the top network on The Stack.

```
xfst[3]: print net
Sigma: a c d e g h l n o p t
Size: 11
Net: EE3E0
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0: c -> s1, d -> s2, e -> s3.
s1: a -> s4.
s2: o -> s5.
s3: l -> s6.
s4: t -> fs7.
s5: g -> fs7.
s6: e -> s8.
fs7: (no arcs)
s8: p -> s9.
s9: h -> s10.
s10: a -> s11.
s11: n -> s4.
```

In this display, the notation s_0 indicates the non-final start state that is numbered 0. A notation like fs_7 , with an initial **f**, indicates a final state, numbered 7. The notation $s_3: l \rightarrow s_6$ indicates that there is an arc labeled **l** from state 3 to state 6. The sigma alphabet is a list of all the single symbols that occur either on the upper or lower side of arcs. The arity is 1 for networks encoding simple languages and 2 for networks (transducers) encoding non-identity relations.

If you call the **define** command and set a variable to a network value, you can also indicate that variable as an argument to **print net** or **print words**. Try the following:

```

xfst[3]: define XXX ( r e ) [ l o c k | c o r k ]
[ i n g | e d | s | 0 ] ;
13 states, 17 arcs, 16 paths.
xfst[3]: print net XXX
Sigma: c d e g i k l n o r s
Size: 11
Net: EE438
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0:  c -> s1, l -> s2, r -> s3.
s1:  o -> s4.
s2:  o -> s5.
s3:  e -> s6.
s4:  r -> s7.
s5:  c -> s7.
s6:  c -> s1, l -> s2.
s7:  k -> fs8.
fs8: e -> s9, i -> s10, s -> fs11.
s9:  d -> fs11.
s10: n -> s12.
fs11: (no arcs)
s12: g -> fs11.
xfst[3]: print words XXX
lock
locking
locks
locked
cork
corking
corks
corked
relock
relocking
relocks
relocked
recork
recorking
recorks
recorked

```

To pop the top network off of The Stack and discard it, use the **pop stack** command. The counter in the prompt will decrement by one, and the next network will then become the new top network, becoming visible to commands like **print words** and **print net**.

```

xfst[3]: pop stack
xfst[2]: print words
cat

```

```

xfst[2]: print net
Sigma: a c t
Size: 3
Net: EE388
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0:  c -> s1.
s1:  a -> s2.
s2:  t -> fs3.
fs3: (no arcs)
xfst[2]:

```

To pop all of the networks off of The Stack, leaving it completely empty, use the **clear stack** command. The counter in the new **xfst** prompt will then be zero.

```

xfst[2]: clear stack
xfst[0]:

```

Play with The Stack until you become comfortable with it. Formally speaking, The Stack of **xfst** is a LIFO (Last-In, First Out) storage mechanism that is familiar to most computer scientists but a bit of a challenge to those from other backgrounds. Failure to understand the operation of The Stack is a common source of confusion and frustration when learning and using **xfst**.

Pushing and Popping Defined Variables

We have already introduced the **define** command in one variant that calls the regular-expression compiler. For example,

```
xfst[0]: define Var [ {dog} | {cat} | {snake} ] ;
```

compiles the indicated regular expression and stores the network value in the indicated variable, here **Var**, without affecting The Stack. The other variant of the **define** command, however, does not include a regular expression (or a terminating semicolon); instead it pops the top network off of The Stack and assigns that value to the indicated variable. The following sequence of operations is equivalent, in the end, to the **define** statement just above.

```

xfst[0]: read regex [ {dog} | {cat} | {snake} ] ;
xfst[1]: define Var
xfst[0]:

```


Note that the counter in the prompt decrements after the call to **define**, which pops The Stack to get the value to assign to Var.

xfst commands that end with regular expressions, like **read regex** and some **define** commands, must have those regular expressions terminated with a semicolon and a newline. **xfst** commands that do not contain a regular expression are not terminated with a semicolon.

When a **define** command includes a regular expression to be compiled (see page 88), that regular expression must begin on the same line as the command name; otherwise **xfst** cannot distinguish it from the **define** variant that pops its argument off of The Stack. It is a very common error to write

```
xfst[0]: define myvariable
[ d o g | c a t | b i r d ] ;
```

expecting the regular expression to be compiled and the variable to be set to the network value. To invoke the regular expression compiler with **define**, at least one character of the regular expression to be compiled must appear on the same line as the **define** statement, e.g.

```
xfst[0]: define myvariable [
d o g | c a t | b i r d ] ;
```

After a network value has been stored in a defined variable, it may later become necessary to push it back on The Stack using the **push defined** command. Note in the following example how the counter increments after the push.

```
xfst[0]: read regex a b c* d (e) f+ ;
xfst[1]: define VarX
xfst[0]: push defined VarX
xfst[1]:
```

The command `push defined VarX` is completely equivalent to the command `read regex VarX ;` (but note that the latter command compiles **VarX** as a regular expression and so needs to be terminated with a semicolon).

Pushing the value of the defined variable **VarX** onto The Stack does not reset or destroy **VarX**; its value persists until **VarX** is reset by another **define** or undefined using the **undefine** command.

```
xfst[1]: undefine VarX
```

Undefinition causes the memory tied up by **VarX**, which may be considerable, to be recycled and made available for other networks.

$U .x. L$	The crossproduct of U and L , where U and L are regular expressions denoting languages, not relations. The result is a relation that maps every string in U , the upper language, to every string in L , the lower language.
$U:L$	The crossproduct of U and L , where U and L are regular expressions denoting languages, not relations. The expression $U:L$ is equivalent to $[U .x. L]$. The colon is a general-purpose crossproduct operator, but it has higher precedence than the $.x.$ operator; in particular, it has higher precedence than concatenation.
$a:a$	Denotes the relation $\langle "a", "a" \rangle$ but is, by convention, stored as a simple label a in the network. The regular expression a is therefore equivalent to $a:a$.
$?:?$	Denotes the relation consisting of all pairs of ANY symbol mapped to ANY symbol.
$?$	Denotes the relation consisting of all symbols identity-mapped to themselves.

Table 3.10: Regular Expressions Denoting Relations

3.2.5 Basic Regular Expressions Denoting Relations

crossproduct

Returning to our review of regular-expression operators, the basic regular expressions for denoting relations are shown in Table 3.10. Other rule-like notations for denoting relations will be presented later.

The crossproduct operation, denoted with the $.x.$ operator, which consists of a period, an x , and another period, without intervening spaces, is the basic operation for defining a relation.

In the $U .x. L$ notation for the crossproduct, both U and L must denote languages, not relations. The overall expression $U .x. L$ denotes the relation wherein each string of U is mapped to each string of L . The language denoted by U is the upper language and the language denoted by L is the lower language.

The colon notation was originally implemented only as a convenient shorthand for notating the crossproduct of two single symbols, e.g. $a:b$ is equivalent to $[a .x. b]$ and a bit more convenient to type. Now the colon can be used as a

general-purpose crossproduct operator, but it has a precedence higher than that of `.x.` (see page 206).

The colon (`:`) and the operator `.x.` are both general-purpose crossproduct operators, but the colon has higher precedence. In particular, it is important to note that the precedence of the colon operator is higher than that of concatenation, while the precedence of `.x.` is lower than that of concatenation.

Experimenting with the crossproduct Operation

This section will continue with some simple examples using the crossproduct operators. If you are comfortable with the notion of crossproducts, proceed to the next section.

Let us suppose that language *Y* consists of the two strings “dog” and “cat” and that language *Z* consists of the two strings “chien” and “chat”, which happen to mean “dog” and “cat” respectively in French. In **xfst**, they might be defined in the following way:

```
xfst[0]: clear stack
xfst[0]: define Y [ d o g | c a t ] ;
xfst[0]: define Z [ c h i e n | c h a t ] ;
```

If we then compute the crossproduct of *Y* and *Z* using **read regex**, the result will be a single transducer network on The Stack.

```
xfst[0]: read regex Y .x. Z ;
xfst[1]:
```

Invoke **xfst**, compile the relation as shown, and follow along in the tests below. Remember that in the expression `Y .x. Z`, the *Y* language is the upper-side of the relation and the *Z* language is the lower-side.

We now use **apply up** and **apply down** to test the behavior of our transducer. If we **apply up** or analyze the string “chien” we get back the two strings “dog” and “cat”. Note that “dog” and “cat” are all the strings in the upper language *Y*.

```
xfst[1]: apply up chien
dog
cat
```

Similarly, if we analyze the string “chat”, we get the same answer.

Now try using **apply down** to the strings “dog” and “cat”. You should see the following output.

```

xfst[1]: apply down dog
chien
chat
xfst[1]: apply down cat
chien
chat

```

The response should be “chien” and “chat” in both cases; these are all the strings in the lower language Z. This behavior results from the fact that the crossproduct operator relates each string of one language to every string of the other.

The crossproduct operator, $.x.$, is a binary operator between two regular languages Y and Z, e.g. $[Y .x. Z]$, such that Y is the upper-level language, Z is the lower-level language, each string in Y is related to each string of Z, and each string of Z is related to each string of Y. X and Y themselves must both denote regular languages, not relations. The crossproduct $[Y .x. X]$ denotes a relation.

Students often wonder what possible use there is for an operator like $.x.$ that relates each string in one language to every string of the other. In practice, however, the languages Y and Z often contain only a single string as in

```

xfst[0]: clear stack
xfst[0]: read regex [ [ d o g .x. c h i e n ] |
                    [ c a t .x. c h a t ] ] ;

```

Try entering this expression and using **apply up** and **apply down** to test the resulting network. The upper-side language again consists of the two strings “dog” and “cat”, and the lower-side language again consists of the two strings “chien” and “chat”, but the relation between the languages has changed. Now when you **apply down** “dog”, you should get only “chien”, and when you **apply down** “cat” you should get only “chat”.

- With the relation denoted above, try **apply up** on “chien” and “chat”. What are the results?
- Try **apply up** on “dog” and “cat”. Try **apply down** on “chien” and “chat”. What are the results? Why?
- Define a relation such that the upper-side language contains the two strings “dog” and “cat”, the lower-side language contains the four strings “chien”, “chienne”, “chat” and “chatte”, and so that “dog” is related to “chien” and “chienne” and so that “cat” is related to “chat” and “chatte”. (A “chienne” is a female dog (bitch) and a “chatte” is a female cat, while

“chien” and “chat” are masculine/unmarked.) Compile your description using **read regex**. Test the network using **apply up** and **apply down**.

You may have noted that the last two relations perform a kind of dictionary lookup or “translation” between English and French words. Although this particular kind of mapping, i.e. French-English, is not very efficiently stored as a finite-state transducer, the concept of relations as bilingual mappers or translators of words between two languages is valid and useful.

In a full-sized Spanish morphological analyzer, for example, the lower-side language is a set of millions of strings that look like conventionally spelled words of Spanish. The upper-side language, defined entirely by the linguist, consists of strings that typically contain baseforms and multicharacter TAG symbols that indicate tense, number, person, mood, aspect, etc. The relation between these two languages is defined, by the linguist, using finite-state grammars, compiled into transducers, so that when you **apply up** an orthographic (surface) word like “canto”, you get back related upper-side or lexical strings like “canto+Noun+Masc+Sg” and “cantar+Verb+PresIndic+1P+Sg” that represent useful ANALYSES of the surface input. Conversely, if you **apply down** “canto+Noun+Masc+Sg” or “cantar+Verb+PresIndic+1P+Sg”, the relation returns the surface word “canto”.²

It is often necessary to designate a relation that has a certain symbol, e.g. **b**, on the upper side and the empty string (epsilon) on the lower side; this is notated $b:0$ or $[b .x. 0]$ or $[b .x. []]$. Similarly, the relation having **b** on the lower side and the empty string (epsilon) on the upper side can be notated $0:b$ or $[0 .x. b]$ or $[[] .x. b]$.

In the regular expression $a:b$, remember that the **a** is on the upper side and the **b** is on the lower side of the resulting relation.

Try compiling the following relation:

```
xfst[0]: clear stack
xfst[0]: read regex [ s w i : a m | s w i m ] ;
```

Test it using **apply down** and **apply up**. The same relation could be notated as $[s:s w:w i:a m:m | s:s w:w i:i m:m]$ because in a regular expression denoting a relation, the symbol s is compiled like $s:s$. This particular relation could also be defined as $[\{swim\}:\{swam\} | \{swim\}]$, as well as many other ways. There are, in fact, an infinite number of regular expressions that denote the same language or relation and compile into equivalent networks.

²While the surface strings to be analyzed are usually pre-defined for us by the standard orthography, the upper-side analysis strings are designed according to the tastes and needs of the developers. This topic is discussed in detail in Chapter 6.

The colon (:) is a general-purpose composition operator, but unlike `.x.` it has higher precedence than concatenation. The regular expression `[s w i:a m | s w i m]` is therefore equivalent to `[s w [i:a] m | s w i m]`.

Projections

As explained in Chapter 2, a transducer encodes a relation between two regular languages, an upper language and a lower language. The upper language is also called the UPPER PROJECTION, and the lower language is also called the LOWER PROJECTION of the relation. Given a transducer N , the operations in Table 3.11 return one of the projections of N , i.e. they take a transducer and return a simple automaton encoding one of the two related languages.

The following `xfst` session defines a small relation and sets the variable `myvar`. Then the upper projection of `myvar` is computed and pushed onto the top of The Stack, and the words of that upper language are enumerated. Then the lower-side language is extracted and enumerated. It should be clear that the operators in Table 3.11 disassemble a relation into its component languages. When N denotes a language, $N.u$ and $N.l$ are equivalent to N .

```
xfst[0]: clear stack
xfst[0]: define myvar [ [ d o g .x. c h i e n ] |
                       [ c a t .x. c h a t ] ] ;
9 states, 9 arcs, 2 paths.
xfst[0]: read regex myvar.u ;
6 states, 6 arcs, 2 paths.
xfst[1]: print words
cat
dog
xfst[1]: pop stack
xfst[0]: read regex myvar.l ;
7 states, 7 arcs, 2 paths.
xfst[1]: print words
chat
chien
```

Orientations

Using the finite-state operations in Table 3.12, one can compute both vertical and left-to-right re-orientations of a network. Where N is a transducer, relating an upper

N.u	Return the upper projection of network N.
N.l	Return the lower projection of network N.

Table 3.11: Operations for Extracting Projections of a Transducer

N.i	Return the upper-lower inverse of network N.
N.r	Return the left-to-right reverse of network N.

Table 3.12: Re-Orienting Operations

language U and a lower language L , the INVERSE of N , traditionally notated N^{-1} , is an upside-down version of N , with the original upper language becoming the lower language, and vice versa. In **xfst** regular expressions, the inverse of N is denoted $N.i$. If N is a simple automaton, denoting a language, the inverse operation has no effect. Any network N is equivalent to $[[N.i].i]$.

The following **xfst** session starts with the same trivial relation/transducer as in the previous example. After enumerating the language of the original upper side, it then inverts the network and enumerates the resulting upper side, which is the original lower-side language. It should be obvious that the inversion operation simply turns the transducer upside-down.

```
xfst[0]: clear stack
xfst[0]: define myvar [ [ d o g . x . c h i e n ] |
                       [ c a t . x . c h a t ] ] ;
9 states, 9 arcs, 2 paths.
xfst[0]: read regex myvar.u ;
6 states, 6 arcs, 2 paths.
xfst[1]: print words
cat
dog
xfst[1]: pop stack
xfst[0]: read regex [myvar.i].u ;
7 states, 7 arcs, 2 paths.
xfst[1]: print words
chat
chien
```

If N is any network, the notation $N.r$ denotes the left-to-right REVERSE of N .

```

xfst[0]: clear stack
xfst[0]: define myvar [ d o g | c a t ] ;
xfst[0]: read regex myvar.r ;
xfst[1]: print words
tac
god

```

3.3 Interacting with the Operating System

3.3.1 File I/O, Defined Variables and The Stack

So far we have used **xfst** as an interactive command-line interface; we type a command at the prompt, and **xfst** executes the command and returns with another prompt. In this section, we will learn how **xfst** can use files for output and for input of data and commands.

Regular Expression Files

We are already acquainted with the **read regex** command and how it allows us to enter a regular expression manually and have it compiled into a network.

```

xfst[0]: read regex ( r e )
[ l o c k | c o r k | m a r k | p a r k ]
[ i n g | e d | s | 0 ] ;
xfst[1]:

```

Recall that the regular expression can extend over any number of lines, and that it must be terminated with a semicolon followed by a newline.

When you progress beyond trivial regular expressions, typing them manually into the **xfst** interface will soon become tedious and impractical; a single error will prevent compilation and force you to retype the entire regular expression. **xfst** therefore allows you to type your regular expression into a REGULAR-EXPRESSION FILE, using your favorite text editor,³ and to compile it using the **read regex** command, adding the < symbol to indicate that the input is to be taken from the file. The command schema is the following:

```
xfst[0]: read regex < filename
```

The regular-expression file should contain only a single regular expression, terminated, as always in **xfst**, with a semicolon and a newline. Your regular-expression file might appear as in Figure 3.4.

³When editing regular-expression files in a Unix-like system, be sure to use a text editor like **vi** or **xemacs** that stores its output as plain text. Word processors typically try to store files in proprietary formats, including invisible markup codes that will confuse the **xfst** regular-expression compiler. In Windows, the Notepad editor creates plain-text files.


```
( r e )
[ l o c k | c o r k | m a r k | p a r k ]
[ i n g | e d | s | 0 ] ;
```

Figure 3.4: A Regular-Expression File Contains a Single Regular Expression with a Terminating Semicolon and Newline

The text in a regular-expression file must contain a single regular expression, and, as with regular expressions typed manually in the interface, it must be terminated with a semicolon *and a newline*. Failure to add the newline after the semicolon is a common mistake. It is an unfortunate quirk of the regular-expression compiler that an invalid regular-expression file, lacking just a final newline after the semicolon, is visually indistinguishable from a valid regular-expression file containing the final newline.

If the file is named `fragment.regex`, then the command

```
xfst[0]: read regex < fragment.regex
```

will cause the text to be read in and compiled into a network that is pushed onto The Stack, just as if the text had been typed manually at the command line. The advantage, of course, is that the regular expression needs to be typed only once, even if it is compiled many times; and you simply use your text editor to make any corrections or additions to the file.

By convention at **Xerox**, regular-expression filenames are given the extension `.regex`.

Inside a regular-expression file, it is often desirable to add comments. In **xfst** regular-expression files, any text from an exclamation mark to the end of the line, as in Figure 3.5, is a comment and is ignored by the compiler.

Binary Files and The Stack

Once one or more networks are compiled and pushed onto The Stack, they can be saved to a BINARY FILE using the **save stack** command, specifying a filename as an argument. E.g. to save a network on The Stack to `myfile.fst`:

```

! This is a comment
( r e )                ! prefix
[ l o c k | c o r k | m a r k | p a r k ] ! root
[ i n g | e d | s | 0 ] ;          ! suffix

```

Figure 3.5: A Regular-Expression File Containing Comments

```

xfst[0]: clear stack
xfst[0]: read regex [ {dog} | {cat} | {elephant} ] ;
xfst[1]: save stack myfile.fst
xfst[1]:

```

As this example shows, saving The Stack to file does not clear or pop The Stack. The resulting file does not contain the regular expression [{dog} | {cat} | {elephant}] but rather a binary representation of the compiled network; such a binary file is not human-readable.

By convention at **Xerox**, binary files are often given the extension `.fst` or `.fsm`.

The **save stack** command saves all the networks on The Stack, so the resulting binary file may store any number of pre-compiled networks. The network or networks saved in a binary file can be read back onto The Stack, in the original order, using the **load stack** command.

```

xfst[0]: clear stack
xfst[0]: load stack myfile.fst
xfst[1]:

```

If the binary file contains multiple networks, then multiple networks will be pushed onto The Stack, and the appropriate number will be displayed in the prompt. Loading a binary file pushes the stored networks back onto The Stack, on top of any other networks that may already be there.

Binary Files and Defined Variables

Compiled networks in **xfst** must be stored on The Stack, in defined variables, or in a binary file. You are already acquainted with the variant of the **define** command that is followed by a regular expression, terminated with a semicolon and a newline, as in

```
xfst[0]: define Var [ m o u s e |
r a b b i t |
b i r d ] ;
```

You are also acquainted with the variant of **define** that has no overt argument and so, like so many **xfst** commands, takes as its default argument the top network on The Stack, popping it and assigning the value to the variable.

```
xfst[0]: clear stack
xfst[0]: read regex [ m o u s e |
r a b b i t |
b i r d ] ;
xfst[1]: define Var
xfst[0]:
```

We saw above that a user might well want to save a set of networks on The Stack to file, and the same may apply to a set of networks stored in variables. The command to save all the current defined-variable networks to file is **save defined**, which takes a filename argument.

```
xfst[0]: save defined myfile.vars
```

The command to read such a file back in, restoring all the previous variable definitions, is **load defined**.

```
xfst[0]: load defined myfile.vars
```

The **save defined** command allows you to save potentially useful networks to file and restore them later, using **load defined**, perhaps in a future **xfst** session.

Wordlist or Text Files

In practical natural-language processing, it is often useful to take a simple WORDLIST file and compile it into a network. By simple wordlist file we mean a file like that in Figure 3.6 containing a list of words, one word to a line, where each word consists of normal alphabetic characters (i.e. there are no multicharacter symbols). The example shows words listed in alphabetical order, but this is not required.

One tedious way to compile this list of words, which effectively enumerates a language, into a network is first to edit the file and convert it into an appropriate regular-expression file like that in Figure 3.7 or the equivalent in Figure 3.8. The converted file can then be compiled using **read regex <filename**.

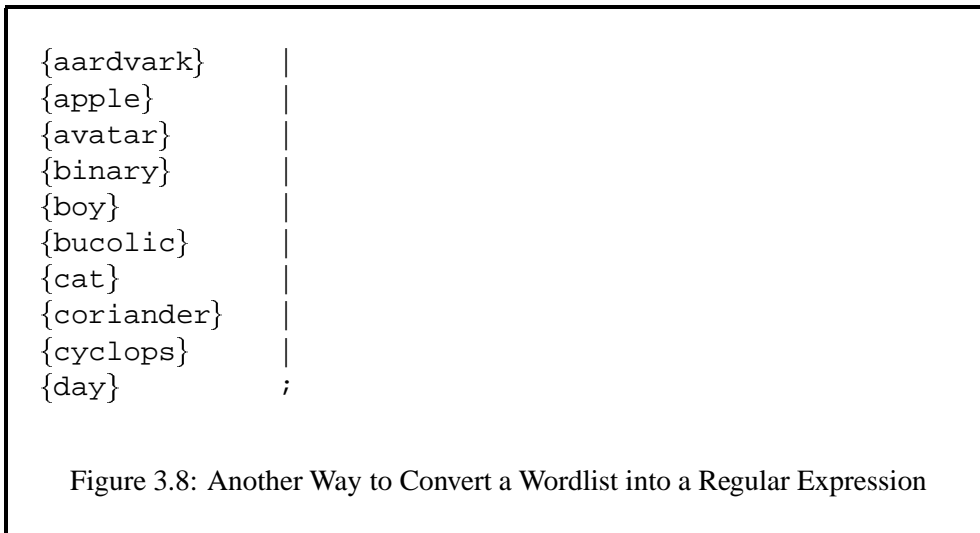
Mercifully, **xfst** provides a special compiler called **read text** that takes as input a simple wordlist as shown in Figure 3.6 and compiles it into the network that encodes the language. **read text** automatically “explodes” each word, compiling it

```
aardvark  
apple  
avatar  
binary  
boy  
bucolic  
cat  
coriander  
cyclops  
day
```

Figure 3.6: A Typical Wordlist

```
a a r d v a r k |  
a p p l e |  
a v a t a r |  
b i n a r y |  
b o y |  
b u c o l i c |  
c a t |  
c o r i a n d e r |  
c y c l o p s |  
d a y |  
;
```

Figure 3.7: A Wordlist Converted into a Regular Expression



as a concatenation of alphabetic symbols, and all the words in the list are unioned to form the result. The result, as usual, is pushed onto The Stack.

```

xfst[0]: read text myfile.wordlist
xfst[1]:

```

In such **read text** commands, the name of the file to be read from can optionally be preceded with a left angle bracket.

```

xfst[0]: read text < myfile.wordlist
xfst[1]:

```

If it's small enough to be humanly convenient, the wordlist to be compiled by **read text** can be typed in interactively. If one types **read text** followed by a newline, **xfst** goes into a text-entry mode and displays the prompt `text>`. After typing one string per line, the user exits text-entry mode by typing the pseudo-entry `END;` (in all capital letters, and including the semicolon), e.g.

```

xfst[0]: read text
text> dog
text> elephant
text> gorilla
text> whale
text> salamander
text> END;

```

```

1.0 Kb. 30 states, 33 arcs, 5 paths.
xfst[1]:
xfst[1]:

```

Instead of typing `END;`, Unix users can type Control-D to escape from text-entry

mode; on Windows, the equivalent is Control-Z.

The reverse operation, **write text** > *filename*, takes the top network on The Stack, which must encode a language (not a relation), and outputs all the words recognized by the network to an indicated file.

```
xfst[0]: write text > myfile.wordlist
xfst[1]:
```

The name of the output file must be preceded with a right angle bracket. If no output file is specified, then **write text** is effectively equivalent to **print words**, outputting the strings to the standard output (the terminal).

Spaced-Text Files

The **read text** and **write text** commands just presented read and write simple wordlist files that contain straightforward alphabetic words, typed one to a line. Such a file contains a set of words and therefore enumerates a finite language. The routines are reciprocal in that any wordlist file created from a network by **write text** can be read back in and compiled into an equivalent network using **read text**.

If a network contains multicharacter symbols like +Noun, +Sg and +Pl, or if it is a transducer (encoding a relation, which is a set of ordered string pairs), then **write text** and **read text** cannot be used. **xfst** will prompt you in such cases to use **write spaced-text** and **read spaced-text** instead.

read text reads a plain wordlist file and compiles it into a simple automaton that recognizes that set of words. **write text** does the reverse. For transducers and for any network containing multicharacter symbols, use **read spaced-text** and **write spaced-text**.

Figure 3.9 shows part of a typical output file from **write spaced-text**, with pairs of strings separated with a blank line, and with individual symbols spaced out. The first string of each pair is taken to represent the upper member of the ordered pair, and all symbols are spaced, allowing +Noun, +Sg and +Pl to be recognized and handled appropriately as multicharacter symbols. The routines are reciprocal in that any wordlist file created from a network by **write spaced-text** can be read back in and compiled into an equivalent network using **read spaced-text**.

The syntax is similar to that of **read text** and **write text**, e.g. the following commands would result in two equivalent networks being placed on The Stack.

```
xfst[0]: clear stack
xfst[0]: read regex [ d o g %+Noun %+Sg .x. d o g |
d o g %+Noun %+Pl .x. d o g s ] ;
xfst[1]: write spaced-text > myfile.spaced
```

```
d o g +Noun +Sg
d o g

d o g +Noun +Pl
d o g s

e l e p h a n t +Noun +Sg
e l e p h a n t

e l e p h a n t +Noun +Pl
e l e p h a n t s

l o u s e +Noun +Sg
l o u s e

l o u s e +Noun +Pl
l i c e

o x +Noun +Sg
o x

o x +Noun +Pl
o x e n

m o u s e +Noun +Sg
m o u s e

m o u s e +Noun +Pl
m i c e
```

Figure 3.9: Typical File Output from **write spaced-text**

```
xfst[1]: read spaced-text myfile.spaced
xfst[2]:
```

If **write spaced-text** is not given a filename argument, it will output to the terminal.

lexc Files

Source files written in the **lexc** language, which will be presented in Chapter 4, can be compiled from within **xfst**.

```
xfst[0]: read lexc < lexc_source_file
xfst[1]:
```

This command, parallel to **read regex** from a file, requires the left-angle-bracket, and it handles only a single input file at a time. The resulting network is pushed onto The Stack.

Prolog Files

For completeness we mention yet another kind of input-output file, the PROLOG FILE. The command **write prolog** outputs networks to file in a Prolog format that can be read back in by **read prolog**. See the **help** messages for further information.

3.3.2 Referring to Binary Files within Regular Expressions

A file storing a single binary network, a regular expression, a wordlist, a spaced wordlist or a prolog representation can be referenced and used directly in regular expressions. Assume, for example, that a network was compiled and stored in the binary file `myfile.fst`. Then `@"myfile.fst"` is a regular expression that denotes the value of that network. Such expressions can appear inside other regular expressions within **read regex** and **define** commands, and in regular-expression files. The following example causes two networks to be read from two previously stored binary files, `myfile.fst` and `yourfile.fst`, unions the networks together, and pushes the result on The Stack.

```
xfst[0]: read regex @"myfile.fst" | @"yourfile.fst" ;
xfst[1]:
```

Table 3.13 summarizes the various regular-expression operators that are available for referencing a file. Binary files are by far the most used in practice. Double quotes are shown around the *filename* in all examples, but they are optional unless, as in the example just above, the *filename* contains non-alphabetic characters like the period.

@ <i>filename</i> "	Denotes the network value stored in the binary file <i>filename</i> . The file format, which is compressed, is read and decompressed.
@bin" <i>filename</i> "	Equivalent to @" <i>filename</i> ".
@re" <i>filename</i> "	Denotes the network value represented in the regular-expression file <i>filename</i> . The regular-expression compiler is called.
@txt" <i>filename</i> "	Denotes the network value represented by the wordlist file <i>filename</i> . The wordlist (text) compiler is called.
@stxt" <i>filename</i> "	Denotes the network value represented in the spaced-text file <i>filename</i> . The spaced-text compiler is called.
@pl" <i>filename</i> "	Denotes the network value stored in the Prolog file <i>filename</i> . The Prolog notation is read and rebuilt into the original network.

Table 3.13: Regular Expressions Referencing a File

3.3.3 Scripts in *xfst*

Script Files

When working interactively with *xfst*, you type a command at the command line, and *xfst* executes it and returns a new prompt. An *xfst* session consists of a sequence of commands and responses. For complex operations, and especially those that will be performed multiple times, you can type a sequence of commands into an *xfst* SCRIPT file using any convenient text editor and then command *xfst* to execute the entire script. The effect is the same as if all the commands in the script were retyped manually. The advantages, of course, are that the script can be edited until it is correct and that the entire script can then be rerun as many times as desired without retyping.

If *myfile.xfst* is an *xfst* script file, you can execute it from within the interactive interface by using the **source** command.

```
xfst[0]: source myfile.xfst
```

The filename may in fact be a complex pathname. When you first invoke *xfst* from the UNIX command line, you can also specify a script name after the **-f** flag.

```
unix> xfst -f myfile.xfst
unix>
```

When invoked in this way, **xfst** will execute the script and then exit automatically back to the operating system.

By convention, script files are given the extension `.xfst` or `.script`.

To execute a start-up script and then continue to enter commands manually in the **xfst** interactive loop, invoke **xfst** with the `-l` flag.

```
unix> xfst -l myfile.xfst
xfst[0]:
```

In general, any sequence of **xfst** commands that you might type manually at the command line can be put in a script file, executed with **source** (or from the UNIX command line using the `-l` and `-f` flags), re-edited until the commands are correct, and then re-executed whenever you desire. The construction of some linguistic products, especially tokenizers and taggers, traditionally involves the writing of complex **xfst** script files.

xfst beginners often confuse regular-expression files and script files. A regular-expression file contains only a single regular expression, terminated by a semicolon and a newline, and is read using the command `read regex < filename`. A script file, in contrast, contains a sequence of **xfst** commands; scripts are executed using the **source** command. Inside a script file, there may of course be **define** and **read regex** commands that include regular expressions.

echo

The **echo** command displays its string argument, which is terminated by a newline, to the terminal. **echo** is primarily useful in scripts to give the developer some runtime feedback on the progress of time-intensive computations. The general template for **echo** commands is the following:

```
echo text_terminated_by_a_newline
```

The following script for building a lexical transducer includes potentially useful **echo** commands. The asterisks and angle brackets are not necessary but help to make the echoed text stand out on your terminal.

```
echo *** Script to build lt.fst ***
clear stack
```

```

echo << Creating the lexicon FST >>
read regex < lex.regex
define LEX
echo << Creating the rule FST >>
read regex < rul.regex
define RUL
echo << Creating the lexical transducer >>
read regex LEX .o. RUL ;
save stack lt.fst
echo << Output written to binary file lt.fst >>
echo << End of Script >>

```

3.3.4 Tokenized Input Files for **apply up** and **apply down**

Tokenized files of words, i.e. files with one word per line, can also be used as input to the **apply up** and **apply down** commands. First recall that the top network on the stack can be applied to individual words by typing the word manually after **apply up** or **apply down** as appropriate. These examples were produced with an English morphological-analyzer transducer on The Stack.

```

xfst[1]: apply up sink
sink+Noun+Sg
sink+Verb+Pres+Non3sg
xfst[1]: apply down sink+Noun+Sg
sink

```

The abbreviations **up** and **down** can be used instead of the full names.

If you need to test a number of words, then typing “**apply up**” or “**apply down**” (or even just “**up**” or “**down**”) before each one of them soon becomes tedious. If you enter simply **apply up** or **up** without an argument, **xfst** will switch into apply-up mode and display the prompt **apply up>**. Input words can then be typed without any prefix.

```

xfst[1]: apply up
apply up> sink
sink+Noun+Sg
sink+Verb+Pres+Non3sg
apply up> sank
sink+Verb+PastTense+123SP
apply up> sunk
sink+Verb+PastPerf+123SP
sunk+Adj
apply up> END;
xfst[1]:

```

To escape from apply-up mode back to normal **xfst** interactive mode, enter “END;” as shown, including the semicolon. Apply-down mode works in just the same way.

Finally, **apply up** and **apply down** can take batch input from pre-edited tokenized files that have one string (i.e. token) per line.⁴ If the tokenized file is named `english-input` and contains the two strings

```
left
leaves
```

and the same English transducer is on the top of The Stack, then the output is the following:

```
xfst[1]:  apply up < english-input
Opening file english-input...

left
leave+Verb+PastBoth+123SP
left+Adv
left+Adj
left+Noun+Sg

leaves
leave+Verb+Pres+3sg
leave+Noun+Pl
leave+Verb+Pres+3sg
leaf+Verb+Pres+3sg
leaf+Noun+Pl
Closing file english-input...
xfst[1]:
```

The output for each input word consists of a blank line, the input word itself, and a set of solutions. The **apply down** utility can also take its input from a tokenized file in exactly the same way.

3.3.5 System Calls

The **system** command passes its string argument, which is terminated by a newline, to the host operating system. In a Unix system, for example, the `ls` command causes the contents of the current directory to be displayed on the terminal. The following **system** command calls the Unix `ls` command without having to exit from **xfst**.

```
xfst[0]:  system ls
```

⁴Each whole line of the input file, including internal spaces, is treated as a string of input. The tokenized file therefore cannot contain comments of any kind.

In modern multiwindow operating systems, where **xfst** can be run in one window and the bare operating system in another, **system** calls are seldom used.

Another little used command in **xfst** is **print directory**, which prints out the contents of the current directory. It is sometimes useful when inputting and outputting files.

```
xfst[0]: print directory
```

In a Unix system, this is the equivalent of invoking

```
xfst[0]: system ls -F
```

See **help print directory** and **help system** for more information.

3.4 Incremental Computation of Networks

3.4.1 Modularization

Any finite-state language or relation can theoretically be described in a single, monolithic regular expression. However, as you model more and more complicated languages and relations, your regular expressions will tend to get bigger, harder to edit, and much harder to read. In extreme cases, a single large regular expression may become difficult for your computer to compile. Whether for human or computational considerations, it is often useful to break up the task into smaller modules.

xfst offers two approaches to modularization, which can be characterized as computing with defined variables and computing on The Stack. The two approaches are not incompatible at all, and may be mixed as the developer finds necessary and convenient.

3.4.2 Computing with Defined Variables

The **define** utility gives us a way to compile a regular expression, set a variable to the network value, and then use the variable in subsequent regular expressions. In this approach, The Stack may be left completely untouched.

As an exercise, define the following three variables.

```
xfst[0]: define FEE [ a | b | c | d ] ;
xfst[0]: define FIE [ d | e | f | g ] ;
xfst[0]: define FOO [ f | g | h | i | j ] ;
```

Recall that once such a variable is defined, you can make the system display all the

words in the language by invoking the **print words** command, e.g.

```
xfst[0]: print words FEE
d
c
b
a
xfst[0]:
```

To see more detailed information about the network, use the **print net** command:

```
xfst[0]: print net FEE
Sigma: a b c d
Size: 4
Net: EE3E0
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0:  a -> fs1, b -> fs1, c -> fs1, d -> fs1.
fs1:  (no arcs)
xfst[0]:
```

Once networks are defined and named, the names can be used in subsequent regular expressions. For example, the intersection of FEE and FIE can be computed and the resulting language printed to the screen with the following commands.

```
xfst[0]: define mylang1 FEE & FIE ;
xfst[0]: print words mylang1
d
xfst[0]:
```

Try the following operations, using **define** and variables.

- Compute the intersection of FIE and FOO. What word(s) does the resulting network recognize?
- Compute the intersection of FEE and FOO. What word(s) does the resulting network recognize?
- Compute the union of FEE and FIE. What word(s) does the resulting network recognize?
- Compute FEE minus FIE. What word(s) does it recognize?
- Define three variables:

1. *Prefix*, including the words “un” and “re”
2. *Root*, including the words “lock” and “cork”
3. *Suffix*, including the words “ing”, “ed”, “s” and the empty string

and then concatenate them in the order *Prefix* followed by *Root* followed by *Suffix* using another regular expression. What words does the resulting network recognize?

Computing with defined variables is intuitive to most users and leads to more readable and maintainable scripts (see Section 3.3.3).

3.4.3 Computing on The Stack

Stack-Based Finite-State Algorithms

As an alternative to building networks via regular-expressions, where syntactic operators denote finite-state operations such as union (`()`), intersection (`&`), subtraction (`-`) and composition (`.o.`), **xfst** also provides explicit spelled-out commands like **union net**, **intersect net**, **minus net**, **concatenate net** and **compose net** that invoke the algorithms directly, popping their arguments off of *The Stack* and pushing the resulting network back onto *The Stack*.

The way to compute networks on *The Stack* is to push suitable arguments and then invoke a command like **union net**; this **union net** pops its arguments off *The Stack*, computes the result, and pushes the resulting network, encoding the union of the arguments, back onto *The Stack*. Different commands need different numbers of arguments. Some unary commands, such as **negate net**, **invert net**, **reverse net** and **one-plus net**, refer to and operate on only one net, the top network on *The Stack*. Some binary commands, such as **crossproduct net** and **minus net**, operate on the top two networks on *The Stack*, and other “n-ary” commands, such as **union net**, **intersect net**, **concatenate net** and **compose net**, operate on all the networks on *The Stack*.

Try entering the following sequence of commands. Trace what is happening at each step, sketching diagrams as necessary.

```
xfst[0]: clear stack
xfst[0]: read regex d o g | c a t | e l e p h a n t ;
xfst[1]: read regex b i r d | s n a k e | h o r s e ;
xfst[2]: print stack
xfst[2]: union net
xfst[1]: print stack
xfst[1]: print net
xfst[1]: print words
```

The commands above compute the union of two networks, encoding two small languages, that have been pushed onto *The Stack*. Similarly, the commands below

compute the intersection of three small networks. Again, trace what is happening at each step, sketching diagrams as necessary. Recall that the **read regex** commands can extend over multiple lines; as always, the regular expression part of the command must be terminated with a semicolon followed by a newline.

```

xfst[0]: clear stack
xfst[0]: read regex
d o g |
c a t |
e l e p h a n t |
b i r d |
w o r m ;
xfst[1]: read regex
b i r d |
s n a k e |
h o r s e |
w o r m ;
xfst[2]: read regex
s n a k e |
d o n k e y |
c r a b |
f i s h |
w o r m |
f r o g ;
xfst[3]: print stack
xfst[3]: intersect net
xfst[1]: print stack
xfst[1]: print net
xfst[1]: print words

```

If you want more practice, redo the two previous exercises using defined variables and perform the union or intersection using regular-expression operators.

intersect net and **union net** are two COMMUTATIVE finite-state operations where the order of the arguments is irrelevant. The arguments in the two examples above could have been pushed onto The Stack in other orders and the result would have been completely equivalent. With other non-commutative operations, however, the order is significant and a clear understanding of The Stack and its LIFO (Last-In, First-Out) behavior becomes crucial.

The order-dependent (non-commutative) operations we will exemplify here are **concatenate net** and **minus net**. Another one, **compose net**, is also order-dependent, but composition is a more difficult operation to grasp and will be handled later.

Concatenation (from *CATENA*, the Latin word for “chain”) refers to the linear chaining together of languages, one after another. If we have two languages A

and B, and we wish to concatenate B after A, we would notate this in the regular expression metalanguage as $[A B]$. It should be obvious that this denotes quite a different language from $[B A]$, which is the concatenation of B before A. To compute $[A B]$ on The Stack, we must be aware that the first argument is A and the second argument is B; but because The Stack is LIFO, i.e. Last-In First-Out, we must first push B on The Stack and then A before we invoke the **concatenate net** command. See Figure 3.10.

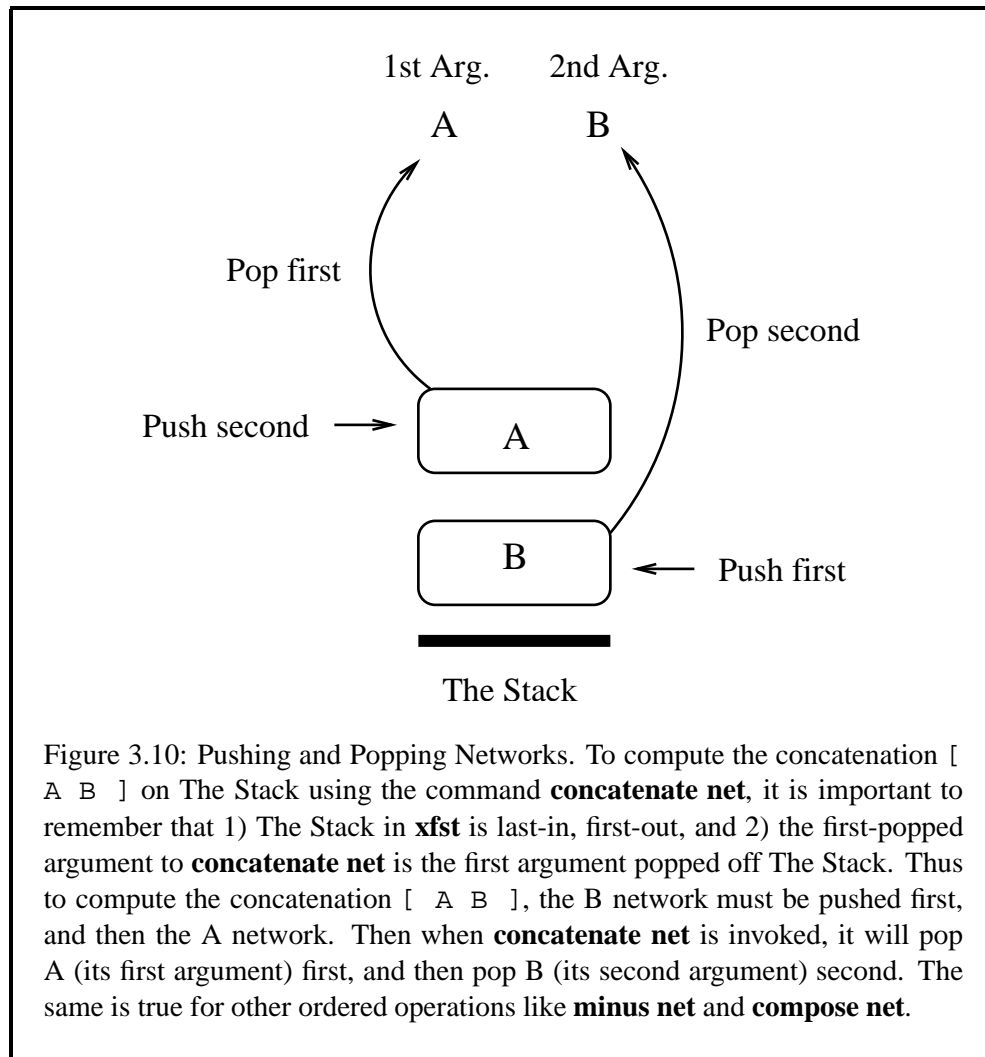


Figure 3.10: Pushing and Popping Networks. To compute the concatenation $[A B]$ on The Stack using the command **concatenate net**, it is important to remember that 1) The Stack in **xfst** is last-in, first-out, and 2) the first-popped argument to **concatenate net** is the first argument popped off The Stack. Thus to compute the concatenation $[A B]$, the B network must be pushed first, and then the A network. Then when **concatenate net** is invoked, it will pop A (its first argument) first, and then pop B (its second argument) second. The same is true for other ordered operations like **minus net** and **compose net**.

concatenate net will then pop the top network, A, off The Stack as its first argument, after which it will pop the next network, B, off The Stack as its second argument, computing the concatenation of B after A or $[A B]$; the resulting network will then be pushed onto The Stack. For ordered operations like **concatenate net**, the order in which you push arguments on The Stack is the opposite of how they are popped off as arguments; this seems backwards to most users, and it

is a cause of many errors and much confusion.

The following example generates a network whose language contains a few regular English verbs. Note especially that the language of verb endings is pushed on The Stack first.

```
xfst[0]: clear stack
xfst[0]: read regex e d | i n g | s | [ ] ;
xfst[1]: print words
xfst[1]: read regex t a l k | w a l k | k i c k ;
xfst[2]: print stack
xfst[2]: print net
xfst[2]: print words
xfst[2]: concatenate net
xfst[1]: print words
```

When the **concatenate net** command is called, it pops its first argument (the network encoding the verb stems) off The Stack and then it pops its second argument (encoding the verbal endings) off The Stack. The concatenation is then performed and the result is pushed back onto The Stack.

The **minus net** operation is another one where the order obviously makes a difference: [A - B] is usually very different from [B - A], just as the arithmetic expression (5 - 2) has a value distinct from (2 - 5). As an exercise, define a language (think of it as the Big Language) containing the words

```
apple
peach
pear
orange
plum
watermelon
cherry
medlar
```

In a regular expression, remember to put white space between symbols that are to be concatenated or to enclose strings with curly braces to “explode” them. Then define another language (think of it as the Small Language) containing just the words

```
peach
plum
cherry
banana
```

Using the **minus net** command and The Stack, subtract the Small Language from the Big Language and print the words in the resulting language. Trace the effect

Unary Commands
invert net
lower-side net
negate net
one-plus net
reverse net
upper-side net
zero-plus net
Binary Commands
crossproduct net
minus net
N-ary Commands
compose net
concatenate net
intersect net
union net

Table 3.14: The Main Stack-Based Finite-State Commands. Most users will find it wise to avoid these commands in favor of computing with defined variables. See Section 3.4.2.

of the commands, sketching a representation of The Stack and showing how the various nets are pushed and popped during the processing.

The Stack is a Last-In, First-Out data structure. If you have an ordered operation like concatenation whose arguments are A, B, C, and D, in that order, then to compute the operation equivalent to $[A B C D]$ on The Stack, you must push D first, C second, B third and finally A. Then when you invoke **concatenate net** the arguments will then be popped off The Stack one at a time in the desired order.

The main commands that operate on stack-based networks are shown in Table 3.14. Computing on The Stack typically appeals to experts who are experimenting interactively and perhaps even debugging new finite-state algorithms; **xfst** started out as an algorithm debugging tool and only gradually added the capability of defining networks via regular expressions, inputting from and outputting to files, etc. For most practical users, computing with defined variables and regular expressions is usually more perspicuous.

Reordering Networks on The Stack

When working interactively with **xfst**, users often push networks onto The Stack in the wrong order for subsequent stack-based operations. **xfst** offers the commands **rotate stack** and **turn stack** that can often resolve the order.

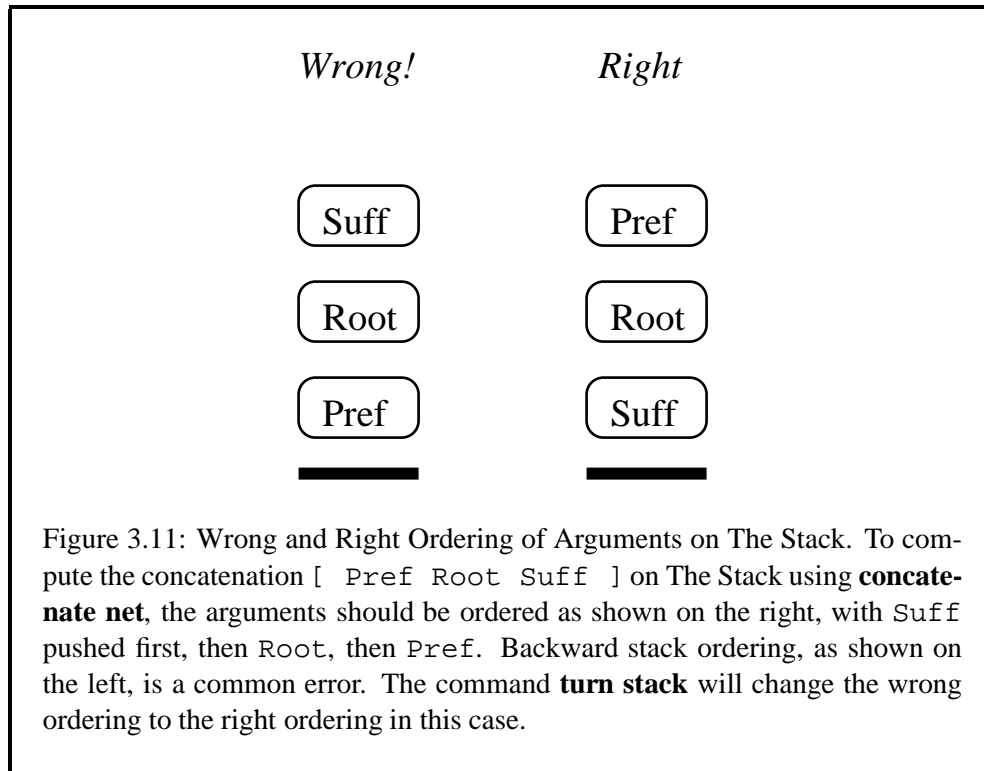
Let's assume that we want to define a simple language consisting of prefixes, root and suffixes, and that we want to concatenate them on The Stack. Intuitively, but incorrectly, most users begin by defining, and pushing onto The Stack, a network for prefixes first, then a network for the roots, and then a network for the suffixes, as in this **xfst** session:

```
xfst[0]: echo ! This is Wrong !
xfst[0]: clear stack
xfst[0]: read regex {re} | {un} | 0 ;
xfst[1]: read regex {cork} | {lock} ;
xfst[2]: read regex {ing} | {ed} | s | 0 ;
xfst[3]: concatenate net
```

The intent would then be to invoke **concatenate net** to pop and concatenate the three networks and push the resulting network back onto The Stack. Unfortunately, this intuitive way of going about the definition leaves the networks on The Stack in the wrong order; for **concatenate net** to work, The Stack must have the leftmost network to be concatenated on the top of the stack and the rightmost network on the bottom. See Figure 3.11.

In this example, we can recover from our error by invoking **turn stack**, which flips the order of all the networks on The Stack.

```
xfst[3]: turn stack
xfst[3]: concatenate net
xfst[1]: print words
recork
recorking
recorks
recorked
relock
relocking
relocks
relocked
uncork
uncorking
uncorks
uncorked
unlock
unlocking
unlocks
```



```

unlocked
cork
corking
corks
corked
lock
locking
locks
locked

```

Try this example, with and without invoking **turn stack** before invoking **concatenate net**.

xfst also provides the command **rotate stack** (see Figure 3.12), which pops the top network and reinserts it on the bottom of The Stack. When there are only two networks on The Stack, **rotate stack** is equivalent to **turn stack**.

For most users, it is best to avoid using stack-based commands like **concatenate net**, **minus net** and **compose net**; the ordering of arguments on The Stack causes much confusion. Instead, define variables and use the variables in subsequent regular expression as shown in Section 3.4.2, page 125.

Before rotate stack *After rotate stack*

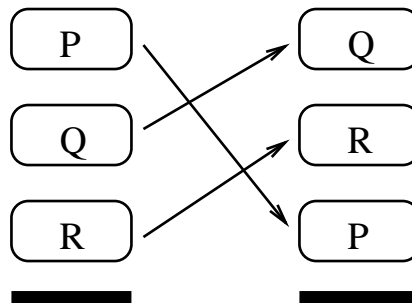


Figure 3.12: Rotation of Arguments on The Stack. The **rotate stack** command pops the top network off The Stack and inserts it on the bottom of The Stack.

3.5 Rule-Like Notations

xfst offers a large selection of rule-like notations. They are all part of the extended **Xerox** regular-expression metalanguage, and they compile into networks in the usual ways, using the **define** and **read regex** commands.

3.5.1 Context Restriction

The restriction operator, $=>$, consisting of an equal sign followed immediately by a right angle bracket, forms regular expressions according to the following template

$$A \Rightarrow L _ R$$

where A , L and R are regular expressions denoting languages (not relations), and L and R are optional. The notation is rule-like in that the L functions as a left context and the R as a right context. The overall restriction expression denotes the language of all strings with the restriction that if a string contains a substring from

A, that substring must be immediately preceded by a substring from L and immediately followed by a substring from R. In other words, $A \Rightarrow L _ R$ denotes the universal language minus all strings that do not satisfy the contextual restriction.

As a simple concrete example, consider

$x \Rightarrow a _ e$

The language denoted by this regular expression includes all strings that do not contain the symbol x ; in addition, it contains all strings containing x where each x is immediately preceded by a and immediately followed by e . Thus the language includes the empty string, “a”, “fish”, “zzzzz”, “axe”, and “zzzzaxemmmm” but not “oxe” or “x” or “axi” that do not satisfy the contextual restrictions on the presence of x .

The restriction written as $A \Rightarrow L _ R$ is in fact compiled as if it were written $A \Rightarrow ?^* L _ R ?^*$, extending the left context on the left, and the right context on the right, with the universal language $?^*$. This reflects the requirement that an A must be immediately preceded by an L and immediately followed by an R; but it allows the L to be preceded by other symbols and the R to be followed by other symbols. A restriction such as $x \Rightarrow _$, which is compiled as $x \Rightarrow ?^* _ ?^*$, effectively restricts nothing and is just another notation for the universal language.

To block either contextual extension, the special $.\#.$ notation can be used to indicate the absolute beginning or end of a string. Thus $A \Rightarrow .\#. L _ R$ requires that any A be immediately preceded by an L that is at the beginning of the word; $A \Rightarrow .\#. _ R$ requires that the A itself be at the beginning of the word; $A \Rightarrow L _ R .\#.$ requires that A be immediately followed by an R which is at the end of a word; and $A \Rightarrow L _ .\#.$ requires that the A itself be at the end of a word. Both the beginning and the end of the word can be specified, as in $A \Rightarrow .\#. L _ R .\#.$

The notation $.\#.$ in rule-like regular expressions designates the beginning or the end of a word, but it is not a symbol and does not appear as a label in a compiled network. In a network, the notion beginning-of-string is inherent in the start state, and the notion end-of-string is inherent in the final state(s).

Restriction expressions are just convenient shorthands for longer and less readable regular expressions using more fundamental operators (see Section 2.4.1). They are compiled into networks in the usual ways, using the **define** and **read regex** commands. Try the following and make sure that you understand the outputs.

```
xfst[0]: read regex x => a _ a ;
```

```

xfst[0]: apply up dog
dog
xfst[0]: apply up xylophone
xfst[0]: apply up laxative
laxative
xfst[0]: apply up axe
xfst[0]: apply up lax

```

Another restriction example, [$A < B$], read as “A before B”, denotes the language of all strings such that no substring from B can precede, even at a distance, any substring from A. Similarly, [$A > B$], read “A after B”, denotes the language of all strings such that no substring from B can follow, even at a distance, any substring from A. The restriction operators are summarized in Table 3.15.

$A \Rightarrow L _ R$	Any substring from A must be immediately preceded by a substring from L and immediately followed by a substring from R.
$A < B$	A before B. Denotes the language of all strings wherein no substring from B can precede any substring from A.
$A > B$	A after B. Denotes the language of all strings wherein no substring from B can follow any substring from A.

Table 3.15: Restriction Operators

3.5.2 Simple replace rules

As part of its extended regular-expression notation, **xfst** includes REPLACE RULES. These rules are very much like the rewrite rules used in traditional phonology, including classic works like *The Sound Pattern of English* (Chomsky and Halle, 1968) and the *Problem Book in Phonology* (Halle and Clements, 1983). replace rules are an extension of regular expressions—they are in fact just shorthand notations for complicated and rather opaque regular expressions built with more basic operators. Like all regular expressions, replace rules are compiled using the **read regex** and **define** utilities available in the **xfst** interface. Grammars of replace rules can be written in regular-expression files, to be compiled with **read regex <filename>** commands; and complex sequences of **xfst** commands for compiling and manipulating rule grammars can be written in **xfst** script files, which are run using the **source** command (see Section 3.3.3).

replace rules do not increase the descriptive power of regular expressions, but they allow us to define complicated finite-state relations in perspicuous rule-like notations that are already familiar to formal linguists. The overall replace rule

notation is very rich, and **Xerox** researchers continue to augment it. However, in this section will concentrate on the simplest and most often used forms of replace rules, and do some exercises, before moving on to the more esoteric types.

The most straightforward kind of replace rule, the right-arrow unconditional replace rule, is built on the following template:

A -> B

where A and B are regular expressions denoting languages (not relations), and the right-arrow operator consists of a hyphen and a right angle bracket, with no space between them. The overall right-arrow rule denotes a relation, and (with one exception to be discussed below) the upper language of the relation denoted by a right-arrow rule is the universal language. The relation is the identity relation, except that wherever an upper string contains a substring from A, the string is related to a surface string or strings containing a substring from B in place of the substring from A.

In a more down-to-earth example, the rule [a -> b] denotes the relation wherein every symbol **a** in strings of the upper-side language is related to a symbol **b** in strings of the lower-side language. The upper-side language of such a relation is the universal language; and any upper-side string that does not contain **a** simply maps to itself on the lower side. For every upper-side string that does contain **a**, it is mapped to a lower-side string that contains **b** in the place of **a**.

The fact that the upper-side language is the universal language means that the resulting network can be applied successfully in a downward direction to any input string. If the network for the rule is applied downward to the string “dog”, the output is “dog”. If it is applied downward to “aardvark”, the output is “bbrdvrk”. When this right-arrow rule is used for generation, we often think of it procedurally as “not matching” the input “dog” and just passing the word through; but it “matches” the string “aardvark”, which contains **as**, and outputs a modification of the string, “bbrdvrk”, with the **as** “replaced” by **bs**. Of course, no such procedural replacement is going on; the rule denotes a relation, and compiles into a transducer. The application of the rule for inputs “dog” and “aardvark” simply returns the output strings related to each input string.

The expression [a -> b] defines a regular relation between two languages. It is tempting, and occasionally irresistible, to think of this as a little algorithmic program that “changes” **as** into **bs** during generation. However, no such program or process is involved, and such rules are properly thought of as representing an infinite set of pairs of strings that are related to each other in a certain way.

Replace rules are compiled into networks and pushed on The Stack in the usual way. Try reproducing the following example step by step.

```
xfst[0]: read regex a -> b ;
```

As with any other regular expression, it is also possible to define a variable to hold the network value.

```
xfst[0]: define Rule1 a -> b ;
```

When such a relation is compiled into a network and pushed on the top of The Stack, you can test it as usual with the **apply down** and **apply up** commands. The **apply down** operation applies the network on the top of The Stack in a downward direction to the input string, what we normally think of as generation. For example, if the network compiled from [a -> b] is applied downward to the input string “abcda”, the result is the string “bbcdb”. That is, the relation states that for every **a** in an upper-side string, there must be a **b** in related strings on the lower-side. Try the following:

```
xfst[0]: clear stack
xfst[0]: read regex a -> b ;
xfst[1]: apply down dog
dog
xfst[1]: apply down aardvark
bbrdvbrk
xfst[1]: apply down abcda
bbcdb
xfst[1]: clear stack
xfst[0]: read regex c -> r ;
xfst[1]: apply down cat
rat
xfst[1]: apply down dog
dog
```

Note that if the rule does not “match” an input string, as in the case of “dog”, the string is identity mapped (i.e. it is passed through unchanged).

In the general pattern for simple replace rules:

A -> B

A and B are not limited to single symbols but can be arbitrarily complicated regular expressions that denote languages, but not relations; the overall rule denotes a relation. A and B do not have to be the same length or conform to any limitations other than denoting regular languages.

Replace rules can also specify left and right contexts, as in the template

A -> B || L _ R

which indicates that any lexical or upper-side string containing a string from A is related to a string or strings on the lower side containing a string from B but only when the left context ends with L and the right context begins with R. From a purely syntactic point of view, the arrow consists of a hyphen and a right angle bracket, with no space in between; and the context separator `||` is a single operator consisting of two vertical-bar characters, with no space in between. Otherwise, white space can be used freely as in other regular expressions. The underscore character indicates the site of the replacement between the two contexts.

The double-vertical-bar operator between the replacement specification and the context indicates that both the context expressions must match on the upper-side of the relation. Other variations of this notation are explained in Chapter 2, below in Section 3.5.5, and in the online documentation. The right-arrow `||` rules are the most commonly used and correspond most closely to traditional rewrite rules.

As with the restriction notation in Section 3.5.1, the left and right contexts are automatically extended with the universal language, such that the rule written

```
A -> B || L _ R
```

is compiled as

```
A -> B || ?* L _ R ?*
```

A, B, L and R can denote arbitrarily complex languages (not relations), and L and R are optional. If a context L or R is empty, the resulting left or right context is then the universal language (i.e. any possible context is allowed). The overall rule denotes a relation and is compiled, using the usual **define** or **read regex** utilities, into a transducer.

```
xfst[0]: read regex A -> B || L _ R ;
```

From the naive algorithmic point of view, we often think of this rule as specifying that a lexical (upper-side) substring from A is to be replaced obligatorily by the substrings from B in the specified context; however, the rule in fact states a restriction on the relation between two regular languages and compiles into a transducer.

As explained in Section 2.4.2, replace rules can have multiple replacements that are not contextually conditioned, as in

```
[ A -> B, B -> A ]
```

or multiple replacements that share the same context; the multiple replacements are again separated by commas. The use of square brackets around such rules often improves readability, even if they aren't formally required.

```
[ A -> B, C -> D, E -> F || L _ R ]
```

In addition, a rule can have multiple contexts, similarly separated by commas, e.g.

```
A -> B || L1 _ R1, L2 _ R2
```

A single rule can have both multiple replacements and multiple contexts.

```
A -> B, C -> D, E -> F || L1 _ R1, L2 _ R2, L3 _ R3
```

When regular expressions containing rules are compiled by **define** or **read regex** commands, the regular expressions must, as always, be terminated with semicolons. In practice, to avoid confusing the compiler and yourself, it is often wise to surround replace rules with square brackets.

```
xfst[0]: read regex [ a -> b || l _ r ] ;
xfst[1]:
```

In replace-rule contexts, use the special symbol `.#.` to refer to either the absolute beginning or the absolute end of a string. The `.#.` is a special notation spelled with a period, a pound sign, and another period, with no intervening spaces. The end-of-word notation is not a symbol and does not appear in the sigma alphabet of the resulting network.

```
e -> i || .#. p _ r
```

This rule “replaces” lexical (upper-side) **e** with surface (lower-side) **i** when it appears before **r** and after a **p** at the beginning of a word. When a `.#.` appears in the left context, it refers to the beginning of the word; when one appears in the right context, it refers to the end of the word. It is possible to define rules whose contexts refer to both the beginning and the end of a word.

```
g o -> w e n t || .#. _ .#. 
```

Occasionally one wants to specify that an alternation occurs either in a specific symbol context or at a word boundary. For example, if A maps to B either before a right context R or at the very end of the word, this could be notated as either

```
A -> B || _ R , _ .#. 
```

or equivalently as

```
A -> B || _ [ R | .#. ]
```

Optional right-arrow mapping is indicated with the `(->)` operator, which is just the right arrow with parentheses around it. For example, the rule `[a (->) b]` maps each upper-side **a** both to **b** and to the original **a**.

```

xfst[0]: clear stack
xfst[0]: read regex a (->) b ;
xfst[1]: apply down abba
bbbb
bbba
abbb
abba

```

The upper side language of a right-arrow rule is usually the universal language. The one exceptional case is when a language is mapped to the null language. Recall that $?^*$ denotes the universal language, which contains all possible strings. The notation $\sim[?^*]$ therefore denotes the complement of the universal language, which is the null or empty language containing no strings at all. There are in fact an infinite number of ways to denote the empty language, including $[a - a]$, which effectively subtracts a language from itself, leaving no strings. The exceptional replace rules then look like the following:

$A \rightarrow \sim[?^*]$

or

$A \rightarrow [a - a]$

In such cases, all strings containing a string from A are removed from the upper language, leaving an identity relation of all strings that do not contain A . Thus $[A \rightarrow \sim[?^*]]$ is equivalent to $\sim\$\{A\}$, the language of all strings that do not contain a substring from A .

Before looking at any other replace-rule types, we will consolidate our understanding of the right-arrow rules with some exercises.

3.5.3 Rule Ordering and Composition

The composition operation was introduced formally in Section 2.3.1. We reintroduce it here in examples and exercises that involve replace rules and rule ordering. You are urged to reproduce the examples as you read.

The *kaNpat* Exercise

The Linguistic Phenomena to Capture As a first exercise in writing replace rules, and learning about rule ordering and composition, consider a fictional language, where *kaNpat* is an abstract lexical string consisting of the morpheme *kaN* (containing an underspecified nasal morphophoneme *N*) concatenated with the suffix *pat*. Here, as in many natural languages, strange things happen at morpheme boundaries. It so happens that in this language, an underspecified nasal **N** that occurs just before *p* gets REALIZED as (or “replaced by”) an **m**. This is formalized in a replace rule as follows

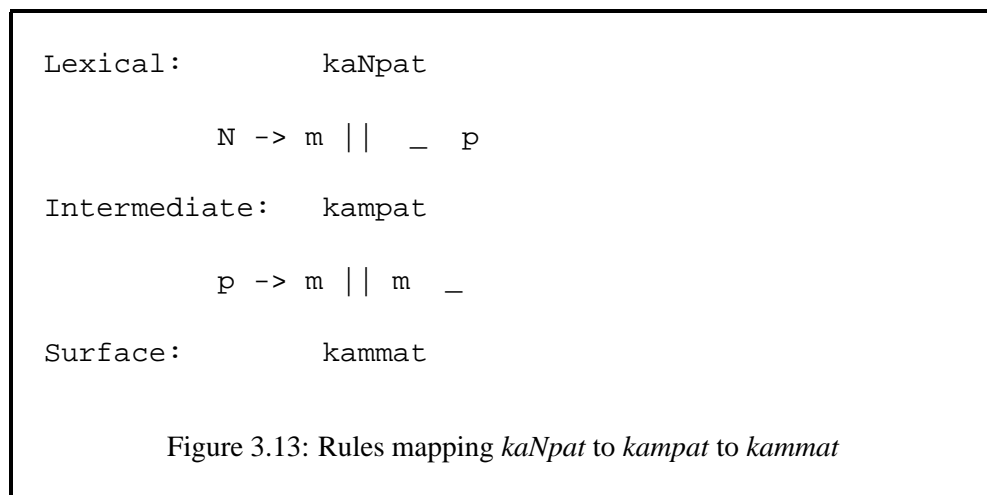
$$N \rightarrow m \mid _ p$$

Read this rule informally as “N is obligatorily replaced by m when it appears in the context before p”. This is a garden-variety kind of nasal assimilation that occurs in many languages.

Notice that the left context in the preceding rule is empty, which means that the left context doesn’t matter; any left context will match. The application of this rule to the input string “kaNpat” yields the new intermediate string “kampat”. A second rule in this language states that a **p** that occurs just after an **m** gets realized as **m**, and this rule is formalized as

$$p \rightarrow m \mid m _$$

Note that in this formalism, the linguist must keep track of the rule order. In this language, the first rule can “feed” the second; i.e. the **m** in the left context of the second rule can be either a lexical (underlying) **m** or an **m** that replaced a lexical **N**. A complete DERIVATION from the underlying string “kaNpat” through “kampat” to the final surface string “kammat” is shown in Figure 3.13.



A replace rule Grammar for *kaNpat* Each replace rule is compiled into a finite-state transducer, and the mathematics of finite-state transducers tells us that if a first transducer maps from initial “kaNpat” to intermediate “kampat”, and that if a second transducer maps from intermediate “kampat” to final “kammat”, then there exists, mathematically, a single transducer that maps, in a single step, directly from “kaNpat” to “kammat”, with no intermediate level. Algorithmically, that single transducer is the COMPOSITION of the two original networks, and at **Xerox** we have an efficient composition algorithm that can compute it in most practical cases.

```
[ N -> m || _ p ]
.o.
[ p -> m || m _ ] ;
```

Figure 3.14: A Regular Expression Denoting a Cascade of Rules. Rules in a cascade are combined via the composition operation. The square brackets in this example are not formally necessary.

```
xfst[0]: clear stack
xfst[0]: define Rule1 [ N -> m || _ p ] ;
xfst[0]: define Rule2 [ p -> m || m _ ] ;
xfst[0]: read regex Rule1 .o. Rule2 ;
```

Figure 3.15: A Cascade of Rules Implemented via Defined Variables and Composition

Composition, which is an ordered operation, is indicated in the **Xerox** regular-expression formalism by the `.o.` operator as shown in Figure 3.14. The composition operator is spelled with a period, the small letter **o** and another period, with no intervening spaces. Composition is an n-ary operation, meaning that any number of transducers can be composed together in a cascade.

Alternatively, one can compile the two rules separately, defining appropriate variables, and then compose them together in a subsequent regular expression as in Figure 3.15. One could also compile the rules separately, using **read regex**, pushing the results onto The Stack and then invoking **compose net** as in Figure 3.16. In such a case, it is vitally important to compile and push the second (lower) rule `[p -> m || m _]` first, so that the network for the upper rule `[N -> m || _ p]` will be pushed on top of it before **compose net** is called.

As when using other ordered operations such as **minus net** and **concatenate net**, the order of the arguments pushed on The Stack is critical. It is up to you to ensure that the “top” rule network in a cascade is on the top of The Stack, and that the overall final order of arguments on the stack mirrors the graphic ordering of the cascade, before you invoke the **compose net** command. Review Section 3.2.4 if you are still uncomfortable with the ordering of arguments on The Stack. Better yet, avoid performing such operations on The Stack in favor of computing with defined variables as shown in Figure 3.15.

Testing Your *kaNpat* Grammar

```

xfst[0]: clear stack
xfst[0]: read regex [ p -> m || m _ ] ;
xfst[1]: read regex [ N -> m || _ p ] ;
xfst[2]: compose net
xfst[1]:

```

Figure 3.16: A Cascade of Rules Computed on The Stack. As usual, when invoking stack-based operations like **compose net**, the user must be extremely careful to push the operands onto The Stack in the correct order. See Section 3.4.3.

- Type the grammar using a text editor as a regular-expression file named something like `kaNpat.regex`. Remember that a regular-expression file contains just a single regular expression and must be terminated with a semicolon followed by a newline. Your regular-expression file should look like Figure 3.14, making sure that the semicolon is followed by a newline.
- Then launch the **xfst** interface and compile your grammar using the command **read regex** *<filename>*.

```
xfst[0]: read regex < kaNpat.regex
```

If there are no mistakes in the file, the grammar should compile cleanly, and the resulting transducer network will be pushed onto The Stack.

- Once you have the rule network on The Stack, you can use **apply down** to test the effect of the grammar on various input strings. Using **apply down**, the input string is matched against the upper side of the transducer, and the output string or strings are read off the lower side of the transducer. We often think of such downward application as GENERATION or LOOKDOWN.
- If you apply a network in the upward direction to an input string, using the **apply up** command, the input string is matched against the lower side of the transducer, and the solution or solutions are read off of the upper side of the transducer. We often think of upward application of the network as LOOKUP or ANALYSIS of the input string.
- In this example, we are primarily interested in generating from the abstract or underlying string “kaNpat”, hoping that our rules will produce the correct surface form as output. With the rule grammar compiled and on the top of The Stack, try


```
xfst[1]: apply down kaNpat
```

If you typed the grammar correctly, the system should return “kammat”. You have just written your first finite-state replace rules.

Experiments

1. Perform **apply down** on the string *kampat*. You should be able to explain the output by tracing its derivation through the rules.
2. Perform **apply down** on the string *kammat*. Again, trace the course of the derivation.
3. Perform **apply up** on the string *kammat*. Why are there multiple analyses? Hint: Remember that transducers are bidirectional mappers between strings in the lexical language and strings in the surface language.
4. Start over and compile the same grammar but with the order of the two rules reversed. Does it still work? Why or why not?

Putting replace rules in Context The regular-expression notation we call replace rules corresponds very closely to the traditional rewrite rules used by phonologists, and much serious phonological work has been done with such rules. One grammar of Mongolian (Street, 1963) has a cascade of about 60 such rules mapping from abstract morphophonemic strings to surface strings written in the Cyrillic orthography.

Traditional rewrite-rule grammars were always somewhat problematic because

- The grammars usually existed only on paper and were checked tediously, and very incompletely, by hand.
- Where researchers did try to computerize their rewrite-rule grammars, they did not understand that the rules could be implemented as finite-state transducers. The solution was often to implement the cascade of rules as a cascade of ad hoc programs that matched the contexts and performed the appropriate replacements, passing the output to the next ad hoc program in the cascade. A grammar of 60 rules would require 60 different processing steps at runtime.
- Where grammars were successfully computerized as cascades of ad hoc programs, they tended to run only in a downward or generating direction. Running these grammars *backwards* to do analysis usually proved impossible or at least extremely inefficient. For example, generating the *kaNpat* example in a **Perl** (version 5.6) script would be especially easy, as shown in Figure 3.17, but it is impossible to run the script backwards to perform analysis because the Perl substitution commands are really procedural programs.

```

$_ = "kaNpat" ;
s/N(?=p)/m/g ;
s/(?<=m)p/m/g ;
print ;

```

Figure 3.17: The *kaNpat* Example in **Perl** 5.6 Works for Generation Only. This example takes the input string “kaNpat” and outputs “kammat”.

- Even when some researchers did understand, after Johnson’s work (Johnson, 1972), that rewrite rules could theoretically be implemented as finite-state transducers, there were no rule compilers or composition algorithms available.
- The **Xerox** Finite-State Calculus now includes the algorithms necessary to compile and compose replace rules, resulting in single transducers that not only run efficiently but also run backwards (for analysis) as well as forwards (for generation). Transducers are inherently bidirectional. replace rules can be used directly for building morphological analyzer/generators, or they can be used as an auxiliary formalism for filtering and customizing transducers built using the **Xerox** finite-state compilers **lexc** and **twolc**.
- replace rules are notational abbreviations of regular expressions built with more basic operators, and they compile into finite-state transducers. **twolc** rules, which we will introduce in Chapter 5, use a rather different notation that was introduced by Kimmo Koskenniemi in his 1983 dissertation entitled *Two-level morphology: a general computational model for word-form recognition and production* (Koskenniemi, 1983); but **twolc** rules also compile into transducers. Because a finite-state transducer made one way is mathematically as good as a finite-state transducer made any other way, the choice of using replace rules vs. **twolc** rules is ultimately a matter of human taste and convenience rather than theory or mathematics.
- Writing a grammar with replace rules usually requires writing a cascade of rules, with rules potentially feeding each other, and great care must be taken to order the rules correctly in the cascade.⁵
- Current research and development in finite-state computing involves improving the algorithms and providing yet more perspicuous high-level notations

⁵With a grammar of **twolc** rules, all rules in a grammar apply simultaneously, avoiding any problems with rule ordering, but introducing the complication of rule conflicts; each rule in a **twolc** grammar must potentially be aware of what all the other rules are trying to do simultaneously. Neither way of writing grammars is simple, but at least you get to choose your headaches to some extent.

for linguists to use. **XRCE** in particular continues work on replace rules, adding more operators and features; and we will introduce some of them below. **twolc** rules, being an established and widely-used notation, are being left pretty much as they are.

Parallel Rules

Sometimes rules cannot conveniently be ordered in a cascade but must apply in parallel. The classic example involves the exchange of two symbols, e.g. to replace all **as** by **bs** and, simultaneously, all **bs** by **as**. Thus for an input string “abba”, the desired output from generation would be “baab”. The individual rules are just [a -> b] and [b -> a], but any attempt to order them yields the wrong results.

Consider the following attempt, where [a -> b] is ordered first.

```
xfst[0]: clear stack
xfst[0]: read regex a -> b .o. b -> a ;
xfst[1]: apply down abba
aaaa
```

Here the output is “aaaa”, resulting conceptually from the top rule mapping the input string “abba” to the intermediate string “bbbb”, which in turn is mapped to “aaaa” by the second rule in the cascade.

In the second attempt, we order [b -> a] first:

```
xfst[0]: clear stack
xfst[0]: read regex b -> a .o. a -> b ;
xfst[1]: apply down abba
bbbb
```

which also fails to produce the desired result. It is obvious that neither ordering is right.

Luckily there is a way not to order the rules but to have them apply in parallel. For simple rules having no context specification, the parallel rules are simply separated by commas. The multiple rules are compiled into a single transducer which performs the desired mapping.

```
xfst[0]: clear stack
xfst[0]: read regex b -> a , a -> b ;
xfst[1]: apply down abba
baab
```

For rules having contexts, the rule separator is not the comma but the double comma; this complication is required because even a single rule can have multiple left-hand-sides or contexts, separated by single commas (see Section 3.5.2). The following rather artificial example illustrates parallel rules with contexts, separated by a double comma.

```

xfst[0]: clear stack
xfst[0]: read regex
b -> a || .#. s ?* _ ,, a -> b || _ ?* e .#. ;
xfst[1]: apply down sabbae
sbaabe

```

In most practical cases, ordered cascades of rules are sufficient and are more efficiently compiled; you should therefore avoid writing parallel rules if you can.

Parallel rules must share the same template, i.e. all must be of the general form [A -> B], or all of the form [A -> B || L - R], etc.

Ordering Rule Networks on The Stack

Consider the case where the user intends to compose a rule network below a lexicon network, but mistakenly pushes the lexicon network on The Stack first:

```

xfst[0]: clear stack
xfst[0]: define Cons b|c|d|f|g|h|j|k|l|m|n|p|q|r|
s|t|v|w|y|z ;
xfst[0]: read regex
[ {kick} | {try} | {bore} ]
[ %+Prog:{ing} | %+Pres3PSg:s | %+Past:{ed} | %+Bare:0 ] ;
xfst[1]: read regex
[ y -> i || Cons _ e d .#. ,, y -> i e || Cons _ s .#. ]
.o.
e -> 0 || Cons _ [ {ing} | {ed} ] ;
xfst[2]: print stack
0: 17 states, 222 arcs, Circular.
1: 14 states, 18 arcs, 12 paths.

```

As shown by **print stack**, these commands leave two networks on The Stack, the rule network above the lexicon network, which is the wrong order if one then intends to call **compose net** to compose the rules on the bottom of the lexicon. The solution here is to call **turn-stack**, which reverses the order of all the networks stored on The Stack. The new output of **print stack** shows that the order of the networks has indeed been reversed, and now **compose net** will yield the desired result.

```

xfst[2]: turn stack
xfst[2]: print stack
0: 14 states, 18 arcs, 12 paths.
1: 17 states, 222 arcs, Circular.
xfst[2]: compose net

```

```

xfst[1]: print lower-words
trying
try
tries
tried
kicks
kick
kicking
kicked
bored
boring
bore
bores

```

If you have more than two networks on the stack and want to turn the whole stack upside down, use **turn stack**.

3.5.4 More Exercises

Southern Brazilian Portuguese Pronunciation Exercise

Capturing Phonemics vs. Orthography When linguists have written finite-state morphology systems at **Xerox**, the practical goal has usually been to map between an abstract lexical level, wherein the strings consist of a canonical base-form plus specially defined multicharacter symbol tags, and a surface level, where the strings are words written in a standard orthography. For example, one of the analyses of the Portuguese surface word “canto”, using a large Portuguese lexical transducer built at **XRCE**, is “cantar+Verb+PresInd+1P+Sg”, having the baseform “cantar” (used by convention as the citation form in traditional printed dictionaries) and the tags +Verb, +PresInd (present indicative), +1P (first person), and +Sg (singular). The multicharacter-symbol tags were chosen and defined by the linguists who wrote the system; there’s nothing mysterious or sacred about them. By **Xerox** convention, the lexical side is always represented graphically on the upper side of the relation with the surface side on the lower side.

```

Lexical:  cantar+Verb+PresInd+1P+Sg
Surface:  canto

```

The emphasis on generating and analyzing orthographical words reflects the fact that **Xerox** researchers deal with natural language mainly in the form of online written texts. However, traditional rewrite rules were most often used by linguists attempting to capture the phonological, rather than orthographical, facts of a language. Replace rules, which closely resemble the traditional rewrite rules, are also

well suited for formalizing phonological alternations which may be only partially or awkwardly reflected in the standard orthography.

For this exercise, the task is to create a cascade of rules that maps from orthographical strings in Portuguese (this will be the lexical level) down to strings that represent their pronunciation (this will be the surface side). There will not be a lexicon. A sample mapping of written “caso” to spoken “kazu” looks like this, with, by convention, the lexical string on top and the surface string on the bottom.

```
Lexical:  caso
Surface:  kazu
```

After studying the facts listed below, you have a choice of approaches:

1. You can write your grammar as a single, monolithic regular expression, in a regular-expression file named something like `port-pronun.regex`. You would then compile it into a network and push it on The Stack by entering **read regex < port-pronun.regex**.
2. You can write your grammar as an **xfst** script, defining variables and then using the variables in subsequent regular expressions (see Section 3.4.2). The script, named something like `port-pronun.xfst` or `port-pronun.script`, would then be run by entering **source port-pronun.xfst**. The script should leave the final network on The Stack to allow immediate testing.

When writing grammars involving cascades of rules, it is usually more convenient to write the grammar in the form of a script, first defining a variable for each rule, and then compiling a regular expression that refers to the rule variables and composes them together in the proper order. This facilitates any necessary re-ordering of rules in the cascade during debugging. Either way, the resulting network will be tested by using the **apply down** utility as in the *kaNpat* exercise. You will once again need to write a cascade of replace rules, and the relative ordering of the rules will be very important.

Standard Portuguese orthography is not always a complete guide to the pronunciation of a word (especially in the case of the letter **x** and the vowels written **o** and **e**). As usual, we will restrict and simplify the data slightly to make the solution manageable as an exercise.

The Facts to be Modeled

- The following description is based on the rather conservative pronunciation of Portuguese in Porto Alegre, Rio Grande do Sul, Brazil. Because the orthography is even more conservative, the rules will roughly characterize the phonological changes that have occurred in this one dialect since the orthography fossilized.

- The final transducer produced by your regular expression or script should generate (using **apply down**) from lexical strings like the following, written in standard Brazilian Portuguese orthography. We will limit the input to lowercase words in this exercise.

casa
 cimento
 me
 disse
 peruca
 simpático
 braço
 árvore

- The surface level produced by your grammar will be written in a kind of crude phonemic alphabet, with special use of the symbols in Table 3.16. Because we have limited our input words to lowercase letters, the six special characters will appear only in surface strings, never at the lexical level. The dollar sign \$ character is special in regular expressions, so you will need to literalize it with a preceding percent sign (%) or with surrounding double quotes.

J	palatalized d , similar to the phoneme spelled j in English “judge”
C	palatalized t , similar to the phoneme spelled ch in “church”
\$	alveopalatal sibilant, like the phoneme spelled sh in English “ship”
L	phoneme spelled lh in Portuguese <i>filho</i> (or gli in Italian <i>figlio</i>)
N	phoneme spelled nh in Portuguese <i>ninho</i> (like the French gn in <i>digne</i>)
R	phoneme spelled rr inside words, single r at the beginning of words

Table 3.16: Special Symbols Used on the Lower-Side to Represent Portuguese Phonemes and Allophones

- The mapping from orthography (lexical side) to pronunciation (surface side) includes the following alternations:
 - The orthographical (lexical-side) **ç** is always pronounced /s/; in other words, a **ç** on the upper side always corresponds to an **s** on the lower

side. In these explanations, we follow the IPA convention of indicating phonemes, the lower-side symbols, between slashes. These slashes should not appear in the output strings.

```
Lexical:      braço
Surface:      brasu
```

Hint: Your rule cascade should include a rule that looks like [ç -> s]. Warning: It will have to be ordered carefully with respect to the other rules in the cascade so that the **s** is not changed by subsequent rules in the cascade.

- The orthographical **ss** is always pronounced /s/. In this and following illustrations, the lexical and surface strings are lined up character pair by character pair, with the **0** (zero, also called epsilon) filling out the places where a lexical symbol maps to the empty string. These zeros are for illustration only and should not actually appear in the surface language of your transducer.

```
Lexical:      interesse
Surface:      interes0i
```

Hint: Your cascade should include a simple rule that looks like [s s -> s], and it will have to be ordered carefully relative to the other rules. Remember that replace rules are regular expressions and that the two symbols **s** and **s** on the left side of the arrow must be separated by white space to indicate concatenation; if you write them together as **ss**, the compiler will treat them as a single multicharacter symbol, which is not what you want.

- The orthographical **c** before **e** or **i**, or before accented versions of these vowel letters, is always pronounced /s/.

```
Lexical:      cimento
Surface:      simentu
```

- The orthographical digraph **ch** is pronounced /ʃ/.

```
Lexical:      chato
Surface:      $0atu
```


Your grammar should include a rule [*c h* -> %\$]. The literalizing percent sign % is required because the dollar sign \$ is a special character in **xfst** regular expressions. Alternatively, one can literalize the dollar sign by putting it inside double quotes, i.e. [*c h* -> "\$"].

- Elsewhere (i.e. not **ch**, and not **ci** or **ce**), orthographical **c** is always pronounced /k/.

Lexical:	casa
Surface:	kaza

No **c** should appear in surface strings.

- The orthographical digraph **lh** is realized as /L/.

Lexical:	filho
Surface:	fiL0u

- The orthographical digraph **nh** is realized as /N/.

Lexical:	ninho
Surface:	niN0u

Remember that the zeros shown in these examples are for illustration only and should not appear in your real output strings.

- Elsewhere, **h** is silent and is simply realized as **0** (zero, the empty string).

Lexical:	homem
Surface:	0omem

- The orthographical digraph **rr** is always realized as /R/. Also, the single **r** at the beginning of a word is always realized as /R/. Elsewhere, **r:r**, i.e. lexical **r** is realized as /r/.

Lexical:	carro	rápido	caro	cantar
Surface:	kaR0u	Rápidu	karu	kantar

- The unaccented **e** is pronounced /i/ at the end of a word, and when it appears in the context between **p** and **r** at the beginning of a word; e.g.

Lexical:	peruca	case
Surface:	piruka	kazi

An unaccented **e** is also pronounced /i/ before an **s** at the end of a word.
Elsewhere **e:e**.

Lexical:	cases
Surface:	kazis

- An unaccented **o** is pronounced /u/ at the end of a word.

Lexical:	braço	caso
Surface:	brasu	kazu

An unaccented **o** is also pronounced /u/ before an **s** at the end of a word.
Elsewhere **o:o**.

Lexical:	braços
Surface:	brasus

- A single **s** is pronounced /z/ when it appears between two vowels.

Lexical:	camisa	case
Surface:	kamiza	kazi

Elsewhere **s:s** (but see above where **s s** -> **s**).

- A word-final **z** is pronounced as /s/.

Lexical:	vez
Surface:	ves

Elsewhere, **z:z**.

- A **d** is pronounced /J/ when it appears before a surface phoneme /i/. (N.B. This change occurs in the environment of any surface /i/, no matter what that surface /i/ may have been at the lexical level.) Elsewhere **d:d**.

Lexical:	disse	verdade	paredes
Surface:	Jis0i	verdaJi	pareJis

- A **t** is pronounced /C/ when it appears before a surface phoneme /i/. (N.B. This change occurs in the environment of any surface /i/, no matter what that surface /i/ may have been at the lexical level.) Elsewhere **t:t**.

Lexical:	tio	partes
Surface:	Ciu	parCis

- The vowels are **a, e, i, o, u, á, é, í, ó, ú, ã, õ, â, ê, ô, ü** and **à**. All lexical symbols map to themselves on the surface level by default.

Testing Portuguese Pronunciation Write a set of replace rules that performs the mappings indicated. As in the *kaNpat* example, the rules should be organized in a cascade, with the composition operator (\circ) between the rules. Be very careful about ordering your rules correctly; the rules cannot be expressed in exactly the same order as the facts listed just above.

If you write your grammar as a monolithic regular expression, then you can compile it by entering

```
xfst[0]: read regex < regex_filename
```

If you write the grammar as an **xfst** script (see Section 3.3.3), then you will run the script by entering

```
xfst[0]: source script_filename
```

It is probably preferable to write the grammar as a script.

Using **apply down**, you should be able to handle all the examples in Table 3.17, entering the lexical or upper-side string in each case and getting back the surface or lower-side string. The zeros representing the empty strings in examples above are not shown here and should not appear in your output. To facilitate the testing, you should type all the input (upper-side) words into a file, called something like *mydata*, with one word to a line, and tell **apply down** to read the various input strings from that file (review Section 3.3.4).

```
xfst[1]: apply down < mydata
```

If you have written your grammar as a script, you can even include this line at the end of the script. Be sure to test *all* the examples to ensure that your rules are really working as they should. Modify your rules, or the rule ordering, and re-test the input words until the grammar is working perfectly.

The Bambona Language Exercise

For students who need a challenge, our next exercise concerns vowel changes in the mythical Bambona language. This is a more difficult exercise, involving the definition of a lexicon relation and a rule-based relation that are then composed together.

Some Preliminaries Recall that in regular expressions, strings of characters written together such as *dog* are treated as single symbols called multicharacter symbols. By **Xerox** convention, we use multicharacter symbols such as *+Verb*, *+Noun*, *+Sg* (singular), and *+Pl* (plural) (or *[Noun]*, *[Verb]*, etc.) in our networks to convey part-of-speech and other featural information. The punctuation marks included by **Xerox** convention in the spelling of the multicharacter symbols make

disse	peru	pedaço
Jisi	piru	pedasu
livro	parte	parede
livru	parCi	pareJi
sabe	cada	simpático
sabi	kada	simpáCiku
verdade	casa	braço
verdaJi	kaza	brasu
chato	vermelho	gatinho
\$atu	vermeLu	gaCiNu
filhos	luz	case
fiLus	lus	kazi
braços	partes	paredes
brasus	parCis	pareJis
me	antes	ninhos
mi	anCis	niNus
rápido	carro	caro
Rápido	kaRu	karu
cantar	bicho	diferentes
kantar	bi\$u	JiferenCis

Table 3.17: Test Data for the Portuguese Pronunciation Exercise. Type the upper-side strings into a separate file, called something like `mydata`, with one word to a line, to facilitate repeated testing with **apply down**.

the output more readable, but because the plus sign and square brackets (and, indeed, almost all the punctuation characters) are special characters in regular expressions, we need to literalize them with a preceding percent sign, e.g. $\%+Noun$ or $\%[Noun\%]$, or by placing the entire multicharacter symbol in double quotes, e.g. $\"+Noun\"$ or $\"[Noun]\"$.

Recall also that the crossproduct operator in expressions such as $[Y .x. Z]$ designates the relation wherein Y is the upper-side language, Z is the lower-side language, and each string in each language is related to all the strings in the other language. Similarly, the colon operator as in $a:b$ denotes crossproduct, but it has higher precedence than $.x.$ and higher precedence than concatenation. The expression $[\%+Noun:0]$, which is equivalent to $[\%+Noun .x. 0]$ therefore denotes the relation that has the single symbol $+Noun$ on the upper side and the empty string on the lower side. The expression $[\%+Pl:\{i1\}]$, equivalent to $[\%+Pl .x. [i 1]]$, denotes the relation that has the single symbol $+Pl$ on the upper side and the concatenation $[i 1]$ on the lower side. You will need to use such notations in this exercise. Be aware that multicharacter symbols cannot appear inside curly braces.

The curly brace notation, as in $\{ing\}$ and $\%+PresPart:\{ing\}$, can contain only simple alphabetic symbols, not multicharacter symbols. If $\wedge U$ is a multicharacter symbol, do not try to write $\%+Tag:\{\wedge Uabc\}$; rather write $\%+Tag:[\wedge U a b c]$ or $[\%+Tag .x. \wedge U a b c]$.

The Facts Here are the linguistic facts of Bambona:

1. There are seven vowels in Bambona.

i	u
e	o
é	ó
a	

where e is a mid-high front vowel, \acute{e} is a mid-low front vowel, o is a mid-high back vowel and \acute{o} is a mid-low back vowel. There are no length distinctions.

2. We will limit this example to Bambona nouns. Nouns always start with a root, which is usually based on the pattern CVC or CVCC. E.g.

mad	"book"
nat	"house"
posk	"girl"

rip	"arm"
kuzm	"cooking pot"
karj	"cellular telephone"
zib	"enemy"
lér	"wild pig"
kóp	"nuclear reactor"
sob	"dentist"
mélk	"stone axe"
rut	"integrated circuit"

Hint: The noun-root sublanguage in your regular expression should simply be the union of all the noun roots listed above. Do not try to handle all possible Bambona roots in this exercise.

- The English glosses listed for the roots and for the suffixes below are provided for informational purposes only. Do not try to include the glosses in your network.
- After the root, nouns can continue with an optional suffix from the following list:

ak	"pejorative"
et	"diminutive"
ig	"augmentative"

A maximum of one of these three suffixes can appear in a word. In regular expressions, an expression is made optional by surrounding it with parentheses or by unioning it with the empty string: thus the expression (a) is equivalent to [a | []]. This new sublanguage should eventually be concatenated onto the end of the noun-root sublanguage. Use the multicharacter tags +Pej, +Dim and +Aug as appropriate on the lexical side.

Hint: One portion of your regular expression should look like this: [%+Pej:{ak} | %+Dim:{et} | %+Aug:{ig}].

- Next can come, optionally, a single suffix indicating the speaker's confidence from the following list:

izm	"obvious"
ubap	"probable"
ópot	"alleged"

Use the tags +Obv, +Prob and +All on the lexical side.

6. Next can come, optionally, a single suffix that indicates number.

il	"plural"
ejak	"paucal (a few)"

Overt number marking is always optional, even when the referent of the word is obviously plural or paucal. In fact, in the syntax, when the noun is preceded by another word that states a number explicitly, the number is never marked in the noun itself. There is no suffix to mark the singular (really unmarked number) reading explicitly, but construct the grammar so that each noun is marked either +Pl or +Pauc, or +NoNum (which is realized as zero/epsilon on the lower side); this effectively makes a number marking obligatory.

7. After the number marking must come one of the following case suffixes (or nothing for nominative/unmarked).

0	nominative/unmarked	
am	accusative	(marks direct objects)
ad	dative	(like English "to")
it	ablative	(like English "from")
ek	benefactive	(like English "for")
ozk	genitive	(like English "of")
ém	inessive	("in"/"at")
ót	elative	("from/out of")
ep	comitative	("with")

The following details are a bit tricky, so read carefully: When overt prepositions are used in the sentence, the noun may be unmarked for case. The accusative, dative, ablative and benefactive (and nominative) cases, if present, always end the word. The genitive *ozk* can end a word, or it can optionally be followed by the *on* suffix which denotes inalienable possession (like your own arms and legs, as opposed to a Sony Walkman or Saab 9000 or the severed limbs of an enemy). Remember that optionality in regular expression is indicated by surrounding an expression with parentheses. The inessive

ém and the elative *ót* can optionally be followed with a suffix *el*, a general intensifier in the language, resulting in the meanings “into” and “out of”, respectively. The comitative *ep* can be followed optionally with the suffix *eg*, denoting the negative, yielding the reading “without”. The suffix *eg* is also used in the verb system, not treated here. Use the following tags on the lexical side:

```
+Nom +Acc +Dat +Abl +Ben +Gen +Ine +Ela +Com
+Inalien +Int +Neg
```

Read the preceding description carefully and picture the possible suffix sequences before writing your regular expressions. Graphing sometimes helps. Published natural-language grammars are seldom this explicit in specifying the morphotactic possibilities, requiring you to read between the lines, study the examples, and, ideally, do some original fieldwork with native informants. Finite-state tools give you a way to encode the linguistic facts about your language, but they cannot do the linguistics for you.

8. There are two ways of going about writing a finite-state description of Bambona morphotactics: One way is to write it as a single regular expression (a “regular-expression file”) to be compiled with **read regex** < *filename*; the other is to write it as an **xfst** script (a “script file”) to be executed using the **source** command. Review the distinction between regular-expression files and scripts (Section 3.3.3) if this is not clear.

It is probably best to write the grammar as a script. Either way, the lexical (upper) side should contain alphabetic symbols and multicharacter tags, while the surface (lower) side contains only alphabetic symbols.

If you write the grammar as a monolithic regular expression in the file `bambona-lex.regex` you would compile it with the command

```
xfst[0]: read regex < bambona-lex.regex
xfst[1]: save stack bambona-lex.fst
```

Save the resulting network, using **save stack**, as the binary file `bambona-lex.fst`. If you write your grammar as a script file named `bambona-lex.xfst`, you would run the script with

```
xfst[0]: source bambona-lex.xfst
```


Lexical	nat+Pej+NoNum+Nom
Surface	natak
Lexical	nat+Dim+Obv+NoNum+Com+Neg
Surface	natetizmepeg
Lexical	sob+Dim+All+Pauc+Gen
Surface	sobetópotejakozk
Lexical	lér+Aug+All+Pauc+Ine+Int
Surface	lérigópotejakémel
Lexical	kóp+Aug+Pl+Ela
Surface	kópigilót
Lexical	rip+Dim+Pl+Gen+Inalien
Surface	ripetilozkon

Table 3.18: Lexical/Surface Pairs from the Lexicon Alone. Note that these surface forms are still just intermediate forms, awaiting correction via rules.

and the script itself should contain the command to save The Stack out to file `bambona-lex.fst`.

The lexicon network by itself is a transducer that should encode a relation that includes the ordered pairs of words shown in Table 3.18; test to make sure (using **apply up** and **apply down** as appropriate). Be warned that some of these lower-side forms from the lexicon are not quite right yet—this is still just the first step of the solution.

The surface forms in Table 3.18 are best thought of as intermediate steps on the way to a final solution. The final mapping of these intermediate forms to the real surface forms will be done later by rules. The words in Table 3.18 might be glossed as in Table 3.19, but do not try to make your transducer produce anything like a translation.

9. In actual surface words of Bambona, the consonants **p**, **t**, and **k** are never followed by the front vowels **i**, **e**, or **é** but always by their corresponding back vowels **u**, **o** and **ó** respectively: and the symbol pairs **i:u**, **e:o** and **é:ó** occur only after **p**, **t** or **k**. The vowel **a** is neutral in this phenomenon. Therefore the example “rip+Dim+Pl+Gen+Inalien” conceptually has three levels:

```
Lexical:      rip+Dim+Pl+Gen+Inalien
Intermediate: ripetilozkon
Surface:     ripotulozkon
```

where the intermediate **e** after **p** is realized in the surface string as **o**, and

natak	"awful house(s)"
natetizmepeg	"without (the) obvious little house(s)"
sobetópotejakozk	"of (the) few alleged dentists"
lérigópotejakémel	"into (the) few alleged wild-pigs"
kópigilót	"from (the) big nuclear reactors"
ripetilozkon	"of (the) little arms"

Table 3.19: Glosses for Some Intermediate Bambona Words. Again, these examples are intermediate words formed by simple concatenation or morphemes; they must subsequently be corrected via the application of suitable alternation rules.

the **i** after **t** is realized as **u**. Note that the mapping from the lexical level to the intermediate "ripetilozkon" is already performed by the lexicon transducer itself. Thus, for the lexicon grammar, the analysis string "rip+Dim+Pl+Gen+Inalien" is the lexical or upper side, and the string "ripetilozkon" is the surface or lower side.

```
Lexical (lexicon FST):  rip+Dim+Pl+Gen+Inalien
Surface (lexicon FST):  ripetilozkon
```

For the alternation rules, the string "ripetilozkon" will be the upper side, and the final form "ripotulozkon" will be the lower side. The rule transducer performs the mapping between intermediate, not yet correct, strings like "ripetilozkon" and final correct surface strings like "ripotulozkon".

```
Lexical (rule FST):    ripetilozkon
Surface (rule FST):    ripotulozkon
```

When the rule transducer is composed on the bottom of the lexicon transducer, the resulting transducer will map directly from the strings on the upper side of the lexicon transducer to the strings on the lower side of the rule transducer. The intermediate level simply disappears in the process of the composition. Don't proceed until you understand this. See Sections 1.5.3 and 2.3.1 for a review of composition.

```
Lexical (final FST):  rip+Dim+Pl+Gen+Inalien
Surface (final FST):  ripotulozkon
```

Intermediate	ripetilozkon
Surface	ripotulozkon
Intermediate	kópizmepeg
Surface	kópuzmepog
Intermediate	poskigizmilek
Surface	poskugizmilek
Intermediate	natetópotilótel
Surface	natotópotulótol

Table 3.20: Some Intermediate to Surface Mappings, Showing Vowel Alternations

In a separate file `bambona-rul.regex` write a regular expression consisting of replace rules that perform the vowel alternations described above. Compile the rules in **xfst** using the command `read regex < bambona-rul.regex` and save the compiled network to file as `bambona-rul.fst`.

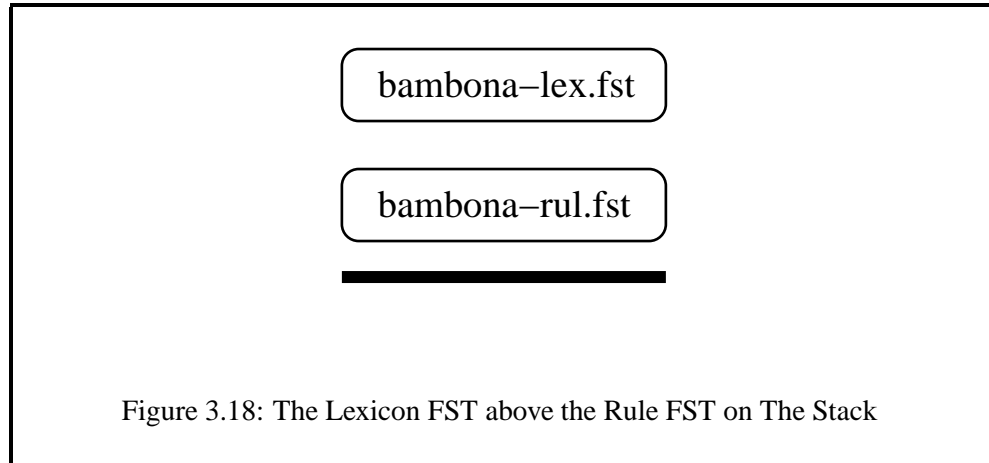
```
xfst[0]: clear stack
xfst[0]: read regex < bambona-rul.regex
xfst[1]: save stack bambona-rul.fst
```

This will still leave the rule transducer on The Stack. Test the rules separately by using **apply down**, entering strings from the lower side of the lexicon network (e.g. “ripetilozkon” and other lower-side examples from Table 3.18) to see if the rules produce the correct surface form. Use the additional examples in Table 3.20 to test your rules (perform **apply down** on the intermediate string, and you should get the surface string).

Compose the Rules under the Lexicon Having written, compiled and tested the lexicon and the rules separately, the next step is to compose the rules under the lexicon and test the result. There are several ways to do this.

Composition on The Stack Both `bambona-lex.fst` and `bambona-rul.fst` are saved to file, so they can be read back onto The Stack, in the correct order, and composed on The Stack. In order to compose the rule transducer under the lexicon transducer, it must appear under the lexicon transducer on The Stack; therefore it must be pushed onto The Stack first. The following commands will leave The Stack as shown in Figure 3.18.

```
xfst[0]: clear stack
xfst[0]: load stack bambona-rul.fst
```



```
xfst[1]: load stack bambona-lex.fst
```

There should be two networks on The Stack. The final network is then created by calling **compose net**.

```
xfst[2]: compose net
xfst[1]: save stack bambona.fst
```

This should leave one network on The Stack, ready for testing. Also save a copy to file `bambona.fst` as shown.

Composition in a Regular Expression The binary networks saved in the files `bambona-lex.fst` and `bambona-rul.fst` can also be composed using a regular expression. The regular-expression notation `@ "filename"` retrieves the value of the binary network stored in `filename`. One can therefore compose the two networks with the following commands:

```
xfst[0]: clear stack
xfst[0]: read regex @"bambona-lex.fst"
.o.
@"bambona-rul.fst" ;
xfst[1]: save stack bambona.fst
```

Once again, this should result in one network being left on The Stack, being the composition of the rules onto the bottom of the lexicon.

Test this lexical transducer with all the examples in Table 3.21, and make up some of your own. Use both **apply up** and **apply down** as appropriate. Note that the mapping between the upper-side and lower-side strings is now done in one step; the intermediate string levels have literally disappeared in the composition.

Lexical	nat+Pej+NoNum+Nom
Surface	natak
Lexical	nat+Dim+Obv+NoNum+Com+Neg
Surface	natotuzmepog
Lexical	sob+Dim+All+Pauc+Gen
Surface	sobetópotojakozk
Lexical	lér+Aug+All+Pauc+Ine+Int
Surface	lérigópotojakómel
Lexical	kóp+Aug+Pl+Ela
Surface	kópugilót
Lexical	rip+Dim+Pl+Gen+Inalien
Surface	ripotulozkon

Table 3.21: Lexical/Surface Pairs in the Final Bambona Lexical Transducer

Bambona Using a Single Script File

The Bambona exercise above involved writing two separate grammar files, compiling or sourcing them separately into networks, and composing them together. Using an **xfst** script, the whole grammar, plus final testing, can be written together in one file.⁶

In real applications, it is often necessary to edit your lexicon and rules many times before you get everything working correctly, and every time you change the regular-expression files you will need to re-execute all the **xfst** commands required to rebuild the final network and retest it. Retyping these commands manually can be tedious and error-prone. It is usually preferable to put all the necessary **xfst** commands in a script file (see Section 3.3.3) and then use the **source** command to re-execute the script after each change of the grammar.

Lexicon and rule grammars themselves can also be made much more readable by writing them in script files. For example, Bambona noun roots can be defined as a sublanguage using the **define** command, written in a script file like this:

```
define NROOT [ {mad} | {nat} | {posk} | {rip} | {kuzm} |
                {karj} | {zib} | {lér} | {kóp} |
                {sob} | {mélk} | {rut}
              ] ;
```

Recall that the regular expression `{mad}` is equivalent to `[m a d]`. Once defined in this way, the name **NROOT** can be used in subsequent regular expressions.

⁶The use of makefiles, see Appendix C, is an even more general and powerful technique for compiling and testing complex systems.

Similarly, the class of Bambona suffixes *ak/et/ig* can be defined in a script file with the following statement:

```
define SUFF1 [ %+Pej:{ak} | %+Dim:{et} | %+Aug:{ig} ] ;
```

After each of the morpheme classes has been defined and named in this manner, a final definition can concatenate the named sublexicons into the preliminary full lexicon:

```
define Lexicon NROOT (SUFF1) (SUFF2) SUFF3 SUFF4 ;
```

In the expression above, **SUFF1** and **SUFF2** are parenthesized because they are optional suffixes in Bambona. replace rules can also be defined, named, and composed with the lexicon via commands in the script file.

Now redo the Bambona exercise, writing everything in a single **xfst** script file that defines all the sublexicons and rules and then performs all the operations necessary to build the final Bambona network, test it, and save it to file as `bambona2.fst`. This `bambona2.fst` should be equivalent to the `bambona.fst` you built and saved previously using multiple source files.

To test the equivalence of two networks, load them onto an **xfst** stack and invoke the **test equivalent** command.

```
xfst[0]: clear stack
xfst[0]: load stack bambona.fst
xfst[1]: load stack bambona2.fst
xfst[2]: test equivalent
```

xfst will then indicate if the two networks are equivalent or not.

The Monish Language Exercise

The following exercise is based on the fictional Monish language.

1. Monish has eight vowels, divided into Front and Back groups, as shown in Table 3.22.
2. Monish exhibits a simple kind of vowel harmony: all of the vowels in a word must be either front vowels or back vowels. So one never finds front vowels and back vowels mixed in the same word.
3. Morphotactically, Monish words always begin with a root and always have suffixes; roots are BOUND morphemes, meaning that they are not valid words by themselves. We will concern ourselves with a subset of the Monish verb system.

Front	Back
i	u
e	o
é	ó
ä	a

Table 3.22: The Eight Vowels of Monish

ruuzod	“walk”
tsarlók	“drink”
ntonól	“gamble”
bunoots	“fish”
vésiimb	“invent facts for sociologists”
yääqin	“drink (alcoholic)”
fesééng	“steal/borrow”

Table 3.23: A Few Monish Roots

4. Table 3.23 shows a few Monish verb roots. The English glosses given here and below are for information only and should not be included in any way in your morphological analysis. Write your grammar so that it accepts only words that are based on the listed roots.
5. Monish suffixes in the lexicon contain underspecified vowels that are realized on the surface as front vowels or as back vowels depending on the overall harmony established by the vowels in the root. The following representation scheme is recommended, but you can use another if you design and document it well.

- Monish suffixes contain the following four underspecified vowel ARCHIPHONEMES or MORPHOPHONEMES:

\hat{U} \hat{O} \hat{O} \hat{A}

- \hat{U} is the underspecified high vowel; it must be realized on the surface as either **u**, in back-vowel words, or as **i** in front-vowel words.
 - Similarly, \hat{O} is the underspecified mid-high vowel; it must be realized on the surface as either **o**, in back-vowel words, or as **e** in front-vowel words.
 - The underspecified mid-low vowel \hat{O} is realized as either **é** (front) or **ó** (back).
 - The underspecified \hat{A} is realized as either **ä** (front) or **a** (back).
6. All Monish verbs begin with a root. After the root comes an optional intensifier that at the lexical level is spelled

$\hat{U}\hat{U}k$

consisting of two underspecified high vowels and a **k**. Use the multicharacter-symbol tag `+Int` at the lexical level for this optional intensifier morpheme.

Note that you cannot use the curly braces, as in `%+Pl:{ed}`, to map `+Int` to a string of symbols like “ $\hat{U}\hat{U}k$ ”; the problem is that multicharacter symbols cannot appear inside the curly-brace notation. Instead, use `[%+Int .x. [%^U %^U k]]` or `[%+Int : [%^U %^U k]]`. See Section 3.2.5, page 106.

7. Next comes a single required aspect marker, one of those shown in Table 3.24. Use the tags `+Perf`, `+Imperf`, and `+Opt` on the lexical side.

Lexical Tag	Morphophonemic Form	Semantics
+Perf	^On	“perfective”
+Imperf	^Ómb	“imperfective”
+Opt	^Udd	“optative”

Table 3.24: Monish Aspect Suffixes

Lexical Tag	Morphophonemic Form	Semantics
+True	^Ank	“true” (the speaker invites you to believe that he/she witnessed the event personally)
+Belief	^A^Av^Ot	“belief” (second-hand, from a witness judged reliable)
+Doubt	^U^Uz	“credulous” (second-hand, from rumor or an unreliable witness)
+False	^Óq	“false” (negative)

Table 3.25: Monish Confidence Suffixes

8. After the aspect marker can come an optional suffix, shown in Table 3.25, conveying the confidence of the speaker.
Use the tags +True, +Belief, +Doubt, and +False at the lexical level.
9. Next the seven suffixes shown in Table 3.26 convey person and number. Exactly one is required in each verb.
Use the tags +1P, +2P, +3P, +Sg, +Pl, +Incl, and +Excl on the lexical level.
10. Visualize the morphotactics of the words before writing a regular-expression grammar. Diagramming often helps.
11. Use comments in your source files to document the meaning of the tags. They may seem obvious now, but they won't be in a week. Comments in regular-expression files and in **xfst** scripts start with an exclamation mark and extend to the end of the line.

! This is a comment

Lexical Tags	Morphophonemic Form	Semantics
+1P+Sg	$\hat{A}\hat{A}b\hat{A}$	“I”
+2P+Sg	$\hat{O}m\hat{A}$	“you (sing)”
+3P+Sg	$\hat{U}v\hat{v}\hat{U}$	“he/she”
+1P+Pl+Excl	$\hat{A}\hat{A}b\hat{O}r\hat{A}$	“we (exclusive)”
+1P+Pl+Incl	$\hat{A}\hat{A}b\hat{U}g\hat{A}$	“we (inclusive)”
+2P+Pl	$\hat{O}m\hat{O}r\hat{A}$	“you (plural)”
+3P+Pl	$\hat{U}v\hat{v}\hat{O}r\hat{U}$	“they”

Table 3.26: Monish Person-Number Suffixes

12. You now have a couple of choices: either

- Create a regular-expression lexicon file called `monish-lex.regex`, compile it using `read regex < monish-lex.regex` and **save stack** the result as `monish-lex.fst`; or
- Define the lexicon using a script file and run it using the **source** command. This script file should save the final result as `monish-lex.fst`.

Either way, the lexicon file will describe the morphotactics of Monish words, and the resulting network should have baseforms and tags on the upper side, and baseforms followed by abstract suffixes on the lower side. Assuming that you choose to write your grammar as an **xfst** script, you will have definitions like the following:

```
define Root [
    r u u z o d      |
    t s a r l ó k   |
    n t o n ó l     |
    b u n o o t s   |
    v é s i i m b   |
    y ä ä q i n    |
    f e s é é n g
] ;

define Suff1 %+Int .x. [ %^U %^U k ] ;

define Suff2 [ %+Perf .x. [ %^O n ] ] |
[ %+Imperf .x. [ %^Ó m b ] ] |
[ %+Opt .x. [ %^U d d ] ] ;
```

Lexical	yääqin+Perf+2P+Pl
Surface	yääqinenémerä
Lexical	fesééng+Opt+False+1P+Pl+Incl
Surface	fesééngiddéqääbigä
Lexical	bunoots+Int+Perf+2P+Sg
Surface	bunootsuukonóma
Lexical	tsarlók+Opt+False+1P+Sg
Surface	tsarlókuddóqaaba
Lexical	ntonól+Imperf+1P+Pl+Excl
Surface	ntonólómbaabora

Table 3.27: Some Pairs of Words in the Final Monish Transducer

13. After writing your lexicon grammar, create a separate rule file called `monish-rul.regex` that describes the alternations required for Monish. Compile it, and save the result as `monish-rul.fst`. Alternatively, if you wrote the lexicon in a script, simply expand the script to define the rules as well. The trick is to write rules that realize each underspecified vowel as front or back depending on the frontness or backness of the vowels of the root.
14. Compose the rule network under the lexicon network, and save the result as `monish.fst`, i.e. perform the following by hand or add the commands to your script file.⁷

```

xfst[0]: clear stack
xfst[0]: read regex @"monish-lex.fst"
.0.
@"monish-rul.fst" ;
xfst[1]: save stack monish.fst

```
15. Test the resulting system. It should analyze and generate examples like those shown in Table 3.27.
16. Test for bad data as well. Your system should not be able to analyze input strings like “yääqinenémorä”, because it contains **o** in a front-harmony word; and it should similarly fail to analyze “tsarlókuddóqaabe” because it contains **e** in a back-harmony word.

⁷You can, of course, also push the two compiled networks on The Stack and compose them using **compose net**; in this case, remember that `monish-rul.fst` must be pushed onto The Stack first so that it is underneath `monish-lex.fst`.

The Monish Guesser

The Monish lexicon above contains only seven roots and can analyze only words that are based on these known roots. In a real development project, you will typically add tens of thousands of roots to the lexicon before you have a commercially viable system.

In some applications you may want to build an auxiliary analyzer or GUESSER that tries to analyze words with roots that are not in the dictionary. To do this, you need to define a regular expression that matches all possible roots in the language, and then use that in place of the seven unioned roots in the original example. This does not mean that the system should accept just anything as a root; we know, or at least can intelligently guess, some facts about Monish roots that should be respected in our guesser:

1. A Monish root must contain at least one vowel.
2. A Monish root cannot contain both a front vowel and a back vowel.
3. In addition, we can assume that the underspecified morphophonemes \hat{U} , \hat{O} , $\hat{Ó}$ and \hat{A} cannot appear in a root; they appear only in suffixes.

Make a copy of your script-based grammar of Monish, modify the definition of roots to cover all possible roots (which should of course cover all the known ones), rebuild the system, and save the resulting network as `monish-guesser.fst`. The trick is to define, using a regular expression, your roots as the set of all strings that either

1. Contain a back vowel plus any number of other symbols that are not front vowels and not morphophonemic vowels, or
2. Contain a front vowel plus any number of other symbols that are not back vowels and not morphophonemic vowels.

The containment operator $\$$ should come in handy for this exercise. Test the result by reanalyzing the good and bad surface examples given above on page 171; it should return analyses for the good words and not for the words that have illegal combinations of front and back vowels. In addition, analyze the following strings which cannot be analyzed by `monish.fst`.

viinémbivveri

monobuddóqaabora

minebiddéqääberä

In each case, your guesser should propose an analysis that identifies a potential root. The following examples should be analyzed to show two possible roots; confirm this and explain the results.

viiniikenänkämä

bunootsuukonóma

3.5.5 More Replace Rules

Replace rules are a very rich, and still growing,⁸ set of formalisms for specifying transducer networks. We will first look at the family of right-arrow rules, progressing later to left-arrow and double-arrow rules.

Right-Arrow Rules

Double-Vertical-Bar Rules The most commonly used form, the right-arrow, double-vertical-bar rules, were introduced above in Section 3.5.2. You will recall that these rules are based on the following template

$$A \rightarrow B \quad || \quad L _ R$$

where A , B , L and R are regular expressions, denoting languages (not relations), and L and R are optional. The overall rule denotes a relation. The left context L is extended by default on the left, and the right context R on the right, with $?^*$, the universal language. The $. \# .$ notation may be used to override this default and indicate the beginning and/or end of a word. These notational conventions will hold for the other subtypes of replace rules to be presented.

But before we move on to the more esoteric rules, it is important to explain the $||$ operator, which indicates, in a right-arrow rule, that the contexts L and R must both match on the upper-side of the relation. Relations and the transducers that encode them are of course bi-directional, but the upper side of a right-arrow rule is most naturally viewed as the “input” side; i.e. right-arrow rules are written with a notational bias toward generation. This semantics, where the sequence $L A R$ must appear in the upper side or input word for the rule to match and apply, corresponds to traditional rewrite rules. Some significant finite-state systems require no other subtypes of rules.

Double-Slash Rules Right-arrow, double-slash rules are built on the following template:

$$A \rightarrow B \quad // \quad L _ R$$

where A , B , L and R have the same restrictions and possibilities as for right-arrow, double-vertical-bar rules. The overall rule again denotes a relation, but here the $//$ operator indicates that the left context L must match on the lower or output side, while the right context R must match on the upper or input side.

⁸For the latest information, see <http://www.fsmbook.com>.

```

xfst[0]: define FrontVowel i | e | é | ä ;
xfst[0]: define Rules
%^U -> i, %^O -> e, %^Ó -> é, %^A -> ä || $[FrontVowel] _
.o.
%^U -> u
.o.
%^O -> o
.o.
%^Ó -> ó
.o.
%^A -> a ;

```

Figure 3.19: Monish Alternation Mapping Front Vowels First

For some examples, // rules appear to “generate their own left context” and have been found useful for modeling vowel harmony, which conceptually moves left-to-right inside a word. As we saw in the Monish exercise (Section 3.5.4), the simplest kind of vowel harmony requires that all the vowels in a word be either front or back. The root, which itself contains only front or back vowels, sets the harmony for the entire word.

In real life, languages with vowel harmony are often more complicated. They may allow compounding of roots, which means that front-vowel and back-vowel roots may occur together in the same word. In such cases, the harmony may change several times from left-to-right as roots are added, with the finally compounded root dictating the harmony for any following suffixes. And whereas Monish suffixes contained only underspecified morphophonemic vowels, some languages contain suffixes with surfacy fully-specified vowels that can attach to any word, regardless of the previous harmony setting, and that potentially reset the harmony for the remainder of the word (or until some other suffix re-sets the harmony again).

Monish suffixes contain the underspecified vowels \hat{U} , \hat{O} , $\hat{Ó}$ and \hat{A} , and the simple rules in Figure 3.19 suffice to realize them correctly. That is, in such a simple vowel-harmony language, where a whole word contains only front or back vowels, the underspecified vowels are realized as front vowels if and only if the left context contains one or more front vowels. Otherwise they are realized as back vowels. The rules could just as well be written the other way, of course, first realizing the underspecified vowels as back vowels after a root containing back vowels, and then realizing any leftovers as front vowels, as shown in Figure 3.20.

In a more complicated vowel harmony language where the harmony may change as you move from left to right, the left context may contain both front and back vowels. Any underspecified vowels must be realized according to the most recently set harmony feature, which is exemplified by the nearest vowel to the left *on*

```

xfst[0]: define BackVowel u | o | ó | a ;
xfst[0]: define Rules
%^U -> u, %^O -> o, %^Ó -> ó, %^A -> a || $[BackVowel] _
.o.
%^U -> i
.o.
%^O -> e
.o.
%^Ó -> é
.o.
%^A -> ä ;

```

Figure 3.20: Monish Alternation Mapping Back Vowels First

the surface side. The following example, using a double-slash rule, achieves the desired mapping:

```

xfst[0]: define FrontVowel i | e | é | ä ;
xfst[0]: define Cons b|c|d|f|g|h|j|k|l|m|n|p|q|
r|s|t|v|w|y|z ;
xfst[0]: read regex
%^U -> i, %^O -> e, %^Ó -> é, %^A -> ä // FrontVowel Cons* _
.o.
%^U -> u
.o.
%^O -> o
.o.
%^Ó -> ó
.o.
%^A -> a ;

```

Let us suppose, for example, that a language much like Monish, but with compounding and fully specified vowels in certain suffixes, constructs lexical strings like the following:

Abstract word: totutike^UUm^{Aqr}O^{On}otun^{Uq}

Morphemes:

totu	root
tike	root
^U Um	suffix
^{Aqr}	suffix
^O On	suffix
otun	suffix (N.B. the fully-specified vowels)

$\hat{U}q$ suffix

In this example, the root *totu* starts the word in back harmony, but the compounded stem *tike* then changes it to front harmony. The following $\hat{U}Um$, $\hat{A}qr$ and $\hat{O}On$ suffixes contain only underspecified vowels and so must agree with the current (front) harmony. Then the *otun* suffix comes along, resetting the harmony to back with its fully-specified back vowels, and then the suffix $\hat{U}q$ must be realized according to the currently active back harmony. The mapping should be as follows (spaces line up the symbols here for comparison of the two levels but do not really appear in the surface string)

```
Lexical: totutike $\hat{U}Um\hat{A}qr\hat{O}Onotun\hat{U}q$ 
Surface: totutike i im äqr e enotun uq
```

Note that each underspecified vowel must be realized in accordance with the harmony of the preceding vowel on the left as it appears on the surface side. The `//` rule, which matches the left context on the surface side, is therefore perfectly suited for capturing this kind of vowel harmony.

Type in the rules above, generate using the lexical string

```
totutike $\hat{U}Um\hat{A}qr\hat{O}Onotun\hat{U}q$ 
```

as input, and satisfy yourself that the proper surface form is generated. Then generate from “*tikewalowo $\hat{U}Um\hat{A}qr\hat{O}Onetiq\hat{U}q\hat{A}mm\hat{A}$* ” and satisfy yourself that the result is correct.

To further compare `||` and `//` rules, try the following:

```
xfst[0]: clear stack
xfst[0]: read regex b -> a || a _ ;
xfst[1]: down abbbbb
aabbbb
```

Note in this example that the left-context **a** must match on the upper (input) side of the relation, and that only the first **b** has an **a** as the upper left context, so the output is “aabbbb”. Now try the following variant with a double-slash operator.

```
xfst[0]: clear stack
xfst[0]: read regex b -> a // a _ ;
xfst[1]: down abbbbb
aaaaaa
```

The output here is “aaaaaa” because the first **a** in the input maps (by default) to itself on the surface, the first **b** is mapped to an **a** on the surface because it has a surface left-context of **a**, and then the remaining **bs** also map to **a**, with the rule appearing to generate its own left context. In reality, of course, the rule denotes a relation rather than some kind of algorithm, and the relation simply maps a lexical “abbbbb” to a surface “aaaaaa”.

Double-Backslash Rules The right-arrow, double-backslash rules are built on the following template:

$$A \rightarrow B \ \backslash\backslash \ L _ R$$

where A, B, L and R have the same restrictions and possibilities as in the previous rules. Such a rule indicates that the left context L is to be matched on the upper (input) side of the relation and the right context R is to be matched on the lower or output side of the relation. The double-backslash rule is therefore the inverse of the double-slash rule as far as context matching is concerned.

As you might expect, the double-backslash rule can appear to generate its own surface context from right to left, and it can be useful for modeling the phonological process of umlaut, which is the opposite of vowel harmony.

To compare `||` and `\backslash` rules, try the following:

```
xfst[0]: clear stack
xfst[0]: read regex b -> a || _ a ;
xfst[1]: down bbbba
bbbaa
```

Note in this example that the right context **a** must match on the upper (input) side of the relation, and that only the last **b** has an **a** as the upper right context, so the output is “bbbaa”. Now try the following variant with a double-backslash operator.

```
xfst[0]: clear stack
xfst[0]: read regex b -> a \backslash _ a ;
xfst[1]: down bbbba
aaaaa
```

The output here is “aaaaa” because the last **a** in the input maps (by default) to itself on the surface, the last **b** is mapped to an **a** on the surface because it has a surface right context of **a**; and then the remaining **bs** also map to **a**, with the rule appearing to generate its own right context. In reality, of course, the rule denotes a relation rather than some kind of algorithm, and the relation simply maps a lexical “bbbaa” to a surface “aaaaa”.

Longest Match Right-arrow, left-to-right, longest-match rules are built on the following template

$$A \ @\rightarrow B \ || \ L _ R$$

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The new arrow operator is `@->`, consisting of an at-sign, a minus sign and

a right angle-bracket, written without any intervening spaces. Where the expression A can match the input in multiple ways, the @-> rule conceptually matches left-to-right and replaces only the longest match at each step.

Right-arrow, right-to-left, longest-match rules are built on the following template

$$A \rightarrow@ B \quad | \quad | \quad L _ R$$

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The new arrow operator is ->@, consisting of a minus sign, a right angle-bracket and an at-sign, written without any intervening spaces. Where the expression A can match the input in multiple ways, the ->@ rule conceptually matches right-to-left and replaces only the longest match at each step.

Longest-match rules are often useful in noun-phrase identification and other kinds of syntactic “chunking”.

Shortest Match Right-arrow, left-to-right, shortest-match rules are built on the following template

$$A @> B \quad | \quad | \quad L _ R$$

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The new arrow operator is @>, consisting of an at-sign and a right angle-bracket, written without any intervening spaces. Where the expression A can match the input in multiple ways, the @> rule conceptually matches left-to-right and replaces only the shortest match at each step.

Right-arrow, right-to-left, shortest-match rules are built on the following template

$$A >@ B \quad | \quad | \quad L _ R$$

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The new arrow operator is >@, consisting of a right angle-bracket and an at-sign, written without any intervening spaces. Where the expression A can match the input in multiple ways, the >@ rule conceptually matches right-to-left and replaces only the shortest match at each step.

Shortest-match rules are not widely used.

Backslash-Slash Rules The right-arrow backslash-slash rules are built on the following template:

$$A \rightarrow B \setminus / L _ R$$

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The \ / operator indicates that both contexts L and R must match on the lower (output) side of the relation.

Backslash-slash rules have not been widely used.

```
[...] -> p || m _ k
```

Figure 3.21: A Simple Epenthesis Rule that Maps “pumkin” to “pumpkin”, Inserting a Single **p** between **m** and **k**

```
[] -> p || m _ k
```

Figure 3.22: A Bad Epenthesis Rule that will Try to Insert an Infinite Number of **ps** between **m** and **k**

Epenthesis Rules Epenthesis rules insert new symbols *ex nihilo* into strings, mapping the empty string into non-empty strings. Such rules are built on the template:

```
[...] -> A || L _ R
```

where **A**, **L** and **R** have the same restrictions and possibilities as in other rule subtypes. A simple epenthesis rule that inserts the symbol **p** between a left context **m** and a right context **k** is shown in Figure 3.21. If the network corresponding to this rule is applied in a downward direction to the string “pumkin”, the output is “pumpkin”.

```
xfst[0]: clear stack
xfst[0]: read regex [...] -> p || m _ k ;
xfst[1]: apply down pumkin
pumpkin
```

The rule in Figure 3.21 illustrates the DOTTED BRACKETS [**.** and **.**], which are required in such epenthesis rules. The need for the dotted brackets is best explained by illustrating what will happen if one tries to write the rule using ordinary square brackets as in Figure 3.22, or with the equivalent **0** notation as in **0 -> p || m _ k**. The notation [**]** or **0** denotes the empty string, and the rule would appear at first glance to be correct, mapping the empty string into **p** between **m** and **k**. However, it must be understood that there are an infinite number of empty strings (which take up no space) between the **m** and the **k** in a word like “pumkin”, and therefore the rule in Figure 3.22 will dutifully try to insert an infinite number of **ps**. In practice, this results in an error or strange output.

```
xfst[0]: clear stack
```

```
[.a*.] @-> p || m _ k
```

Figure 3.23: Dotted Brackets Around the Expression a^* , which can Match the Empty String

```
[a|e|i|o|u] -> %[ ... %]
```

Figure 3.24: A Bracketing Rule to Surround Vowels in the Output with Literal Square Bracket Symbols

```
xfst[0]: read regex [] -> p || m _ k ;
xfst[1]: apply down pumkin
Network has an epsilon loop on the input side.
pumkin
pumpkin
```

Here **xfst** warns that the network has an epsilon loop, corresponding to the infinite number of epsilons (empty strings) between **m** and **k**, and the output is not what we expected. In some cases, an attempt to apply such rules will even result in a segmentation fault and cause **xfst** to crash.

The useful effect of the new `[. and .]` dotted brackets is to cause the rule to be compiled without an epsilon loop, so that it will treat the input as having only one empty string before and after each input symbol. In addition to the notation `[. .]`, indicating a single empty string, dotted brackets are also necessary in examples like the rule in Figure 3.23, where the expression being replaced is `[. a* .]`. It should by now be understood that the regular expression a^* matches all strings containing zero or more **as**, and that this includes the empty string. The bad epenthesis rule $a^* @-> p || m _ k$ will therefore try, in cases where there are no **as** in the context, to insert an infinite number of **ps** between **m** and **k**. So in any replace rule, where the input expression can match the empty string, the input expression should be enclosed in dotted brackets.

Bracketing or Markup Rules It is often useful to have rules that match certain patterns in input strings and surround them in the output with some kind of bracketing. This bracketing might consist of punctuation symbols like `[` and `]`, XML tags like `<latin>` and `</latin>`, or whatever is meaningful and useful in the current application. For example, the rule in Figure 3.24 surrounds vowels with literal square brackets. The three dots `...`, written without intervening spaces, are

in fact a special finite-state operator that refers to the matched symbols. The rule indicates that the matched symbols are to be mapped without change to the output and surrounded with square brackets.

```
xfst[0]: clear stack
xfst[0]: read regex [a|e|i|o|u] -> %[ ... %] ;
xfst[1]: apply down unnecessarily
[u]nn[e]c[e]ss[a]r[i]lly
```

Bracketing rules perform a kind of double epenthesis around the matched substring. The vowel-bracketing rule could be rewritten, in a less readable and harder to maintain form, using a pair of parallel epenthesis rules.

```
xfst[0]: clear stack
xfst[0]: read regex [ [..] -> %[ | | _ [a|e|i|o|u] , ,
[..] -> % | | [a|e|i|o|u] _ ];
xfst[1]: apply down unnecessarily
[u]nn[e]c[e]ss[a]r[i]lly
```

The bracketing expressions can include strings of symbols, for example XML tags.

```
xfst[0]: clear stack
xfst[0]: read regex "ad hoc" -> %<latin%> ... %</latin%> ;
xfst[1]: apply down avoid writing ad hoc code
avoid writing <latin>ad hoc</latin> code
```

Bracketing rules can also be conditioned with contexts in the usual ways. The following rule surrounds “ad hoc” with the tags <latin> and </latin> unless it is already surrounded with such tags.

```
xfst[0]: clear stack
xfst[0]: read regex "ad hoc" -> %<latin%> ... %</latin%> | |
.#. ~[ ?* %<latin%> " "*" ] _ ~[ " "*" %</latin%> ?* ] .#. ;
xfst[1]: apply down avoid writing ad hoc code
avoid writing <latin>ad hoc</latin> code
xfst[1]: apply down avoid writing <latin>ad hoc</latin> code
avoid writing <latin>ad hoc</latin> code
```

Bracketing rules using the longest-match operator are often useful for bracketing maximally long sequences where the matching expression could match in multiple ways. As a simple example, assume that we wanted to bracket not just single vowels but sequences of vowels. The following rule can match the input in multiple ways and so generates multiple outputs.

```
xfst[0]: clear stack
xfst[0]: read regex [a|e|i|o|u]+ -> %[ ... %] ;
xfst[1]: apply down feeling
f[e][e]l[l]i[ng]
```

```
f[ee]l[i]ng
xfst[1]: apply down poolcleaning
p[o][o]lcl[e][a]n[i]ng
p[o][o]lcl[ea]n[i]ng
p[oo]lcl[e][a]n[i]ng
p[oo]lcl[ea]n[i]ng
```

To favor the longest matches, the @-> operator can be used:

```
xfst[0]: clear stack
xfst[0]: read regex [a|e|i|o|u]+ @-> %[ ... %] ;
xfst[1]: apply down feeling
f[ee]l[i]ng
xfst[1]: apply down poolcleaning
p[oo]lcl[ea]n[i]ng
```

This longest-match behavior is often useful in syntactic “chunking” applications, such as bracketing noun phrases in text. Let’s assume that the words in our input sentences have already been morphologically analyzed and reduced to part-of-speech tags such as **d** for determiner, **a** for adjective, **n** for nouns, **p** for prepositions and **v** for verbs. The sentence originally reading “The quick brown fox jumped over the lazy dogs.” would then be reduced to “daanvpdan”. Similarly, “Boy scouts do good deeds for the senior citizens living on the poor side of town.” would reduce to the string of symbols “nnvanpdanvpdanpn”. A very loose characterization of English noun phrases, appropriate for this simple example, is that they start with an optional determiner (d), followed by any number of adjectives a*, and end with one or more nouns n+. The following rule brackets all possible noun phrases:

```
xfst[0]: clear stack
xfst[0]: read regex (d) a* n+ -> %[ ... %] ;
xfst[0]: apply down daanvpdan
[daan]vp[dan]
[daan]vpd[an]
[daan]vpda[n]
d[aan]vp[dan]
d[aan]vpd[an]
d[aan]vpda[n]
da[an]vp[dan]
da[an]vpd[an]
da[an]vpda[n]
daa[n]vp[dan]
daa[n]vpd[an]
daa[n]vpda[n]
xfst[0]: apply down nnvanpdanvpdanpn
```

```

[n][n]v[an]p[dan]vp[dan]p[n]
[n][n]v[an]p[dan]vpd[an]p[n]
[n][n]v[an]p[dan]vpda[n]p[n]
[n][n]v[an]pd[an]vp[dan]p[n]
[n][n]v[an]pd[an]vpd[an]p[n]
[n][n]v[an]pd[an]vpda[n]p[n]
[n][n]v[an]pda[n]vp[dan]p[n]
[n][n]v[an]pda[n]vpd[an]p[n]
[n][n]v[an]pda[n]vpda[n]p[n]
[n][n]va[n]p[dan]vp[dan]p[n]
[n][n]va[n]p[dan]vpd[an]p[n]
[n][n]va[n]p[dan]vpda[n]p[n]
[n][n]va[n]pd[an]vp[dan]p[n]
[n][n]va[n]pd[an]vpd[an]p[n]
[n][n]va[n]pd[an]vpda[n]p[n]
[n][n]va[n]pda[n]vp[dan]p[n]
[n][n]va[n]pda[n]vpd[an]p[n]
[n][n]va[n]pda[n]vpda[n]p[n]
[nn]v[an]p[dan]vp[dan]p[n]
[nn]v[an]p[dan]vpd[an]p[n]
[nn]v[an]p[dan]vpda[n]p[n]
[nn]v[an]pd[an]vp[dan]p[n]
[nn]v[an]pd[an]vpd[an]p[n]
[nn]v[an]pd[an]vpda[n]p[n]
[nn]v[an]pda[n]vp[dan]p[n]
[nn]v[an]pda[n]vpd[an]p[n]
[nn]v[an]pda[n]vpda[n]p[n]
[nn]va[n]p[dan]vp[dan]p[n]
[nn]va[n]p[dan]vpd[an]p[n]
[nn]va[n]p[dan]vpda[n]p[n]
[nn]va[n]pd[an]vp[dan]p[n]
[nn]va[n]pd[an]vpd[an]p[n]
[nn]va[n]pd[an]vpda[n]p[n]
[nn]va[n]pda[n]vp[dan]p[n]
[nn]va[n]pda[n]vpd[an]p[n]
[nn]va[n]pda[n]vpda[n]p[n]

```

The reason for all the outputs is that a string like “the lazy dogs” is a noun phrase, but so are “lazy dogs” and just “dogs”. What we want, in practice, is to bracket whole noun phrases, preferring the longest possible match in each case where multiple matches are possible. Again, the @-> operator saves the day, returning a single output for each input string.

```
xfst[0]: clear stack
```

```

xfst[0]: read regex (d) a* n+ @-> %[ ... %] ;
xfst[0]: apply down daanvpdan
[daan]vp[dan]
xfst[0]: apply down nnvanpdanvpdanpn
[nn]v[an]p[dan]vp[dan]p[n]

```

Left-Arrow Rules

While all replace rules compile into transducers, and so are bi-directional, right-arrow rules are written with a certain notational bias towards generation. The input side of a right-arrow rule is most naturally visualized as the upper side. Right-arrow rules are very frequently used in phonology and morphology to map from abstract lexical strings, on the upper side, to surface or more surface strings on the lower side; networks compiled from right-arrow rules are typically composed on the bottom of a lexicon network.

The family of left-arrow rules, in contrast, has a notational bias towards upward or analysis-like application. Left-arrow rules are often composed on the upper side of a lexicon network, mapping from lexical strings upwards to modified lexical strings.

The simplest left-arrow rules are built on the following template

```
B <- A
```

where A and B are regular expressions denoting languages (not relations). The overall $B \leftarrow A$ rule compiles into a transducer that is the inversion of $A \rightarrow B$.

In morphological analyzers at **Xerox**, left-arrow replace rules are often used to modify morphological tags on the upper side of a lexical transducer. Consider the case where an English-speaking linguist has written a morphological analyzer of Portuguese that contains pairs of strings like the following, where +Noun, +Masc and +Pl are multicharacter symbols.

```

Lexical: livro+Noun+Masc+Pl
Surface: livros

```

A Portuguese-speaking user might prefer to see a tag like +Subst, for *substantivo*, in place of the English-oriented +Noun tag. Changing such a tag on the upper side is a trivial matter using left-arrow rules. Assuming that the original lexical transducer is stored in a binary file named `lexicon.fst`, the command

```

xfst[0]: clear stack
xfst[0]: read regex [%+Subst <- %+Noun] .o. @"lexicon.fst" ;
xfst[1]: save stack lexicon-port.fst

```

will produce a new version of the lexical transducer with pairs of strings like the following


```
Lexical: livro+Subst+Masc+Pl
Surface: livros
```

As far as anyone can tell from looking at the result `lexicon-port.fst`, the tag `+Subst` was there from the beginning.

Of course, it can easily be shown that left-arrow rules are not formally necessary. One could achieve the same result by

1. Inverting `lexicon.fst`,
2. Composing the right-arrow rule `[%+Noun -> %+Subst]` on the bottom of the inverted lexicon, and
3. Inverting the result.

as in the following commands.

```
xfst[0]: clear stack
xfst[0]: read regex
[["@lexicon.fst"].i .x. [ %+Noun -> %+Subst ] ].i ;
xfst[1]: save stack lexicon-port.fst
```

However, such multiple inversions, and the writing of upside-down rules, are not at all natural to most developers.

Left-arrow rules can also have contexts

```
B <- A || L _ R
```

and the `||` operator here indicates that both contexts must match on the default input side, which in a left-arrow rule is the *lower* side.

The `//` and `\\` operators are also available in left-arrow rules, but their interpretation requires some comment. In left-arrow double-slash rules such as

```
B <- A // L _ R
```

the `//` operator indicates that the left context must be matched on the *upper* side, i.e. the output side, and that the right context must be matched on the *lower* or input side. Conversely, in a rule such as

```
B <- A \\ L _ R
```

the left context is matched on the lower or input side, and the right context is matched on the upper or output side.

The left-arrow backslash-slash rules are built on the following template.

```
A <- B \/ L _ R
```

In a left-arrow rule, the `\/` operator indicates that both contexts `L` and `R` must match on the upper (output) side of the relation.

Optional left-arrow mapping is also possible using the `(<-)` operator, which is just the left arrow with parentheses around it. A rule like `[b (<-) a]` maps each lower-side `a` both to `b` and to `a`, resulting in behavior like the following:

```
xfst[0]: clear stack
xfst[0]: read regex b (<-) a ;
xfst[1]: apply up abba
bbbb
bbba
abbb
abba
```

The development of new subtypes of replace rules is a continuing project, and `xfst` does not currently implement the left-arrow operators `<-@`, `<@`, `@<-` or `@<`, although they would be welcome additions.

Double-Arrow Rules

Finally, `xfst` provides double-arrow rules built on the following templates:

```
A <-> B
A <-> B || L _ R
```

where `A`, `B`, `L` and `R` must denote languages, not relations. Such a rule is compiled like the simultaneous pair of rules

```
[ A -> B || L _ R , , A <- B || L _ R ]
```

where in the first rule, `A -> B || L _ R`, the contexts must match on the upper side of the relation, and in the second rule, `A <- B || L _ R` they must match on the lower side. Double-arrow rules are also possible with the `//` and `\\` operators, but in these cases the understanding of where the contexts must match transcends normal human comprehension. Double-arrow rules have not been widely used in practice.

3.6 Examining Networks

Once a network has been compiled and pushed onto The Stack, one can always test it using the **apply up** and **apply down** commands. However, there exist other helpful commands that reveal the nature and contents of your networks.

3.6.1 Boolean Testing of Networks

xfst does not include control structures such as **if-then**, **while** or **for** that respond to boolean tests, but certain tests are available during manual interaction.

N-ary Boolean Tests

- The **test-equivalent** command returns 1 (TRUE) if and only if all the networks on the stack are equivalent.
- The **test overlap** command returns 1 (TRUE) if and only if all the networks on the stack have a non-empty intersection.
- The **test sublanguage** returns 1 (TRUE) if and only if the language of the *i*-th network is a sublanguage of the next (*i*+1-th) network on The Stack. The comparison starts at the topmost network (*i*=0) and works down.

Unary Boolean Testing

The unary boolean tests are always applied to the top network on The Stack.

- The **test null** command returns 1 (TRUE) if and only if the top network on The Stack encodes the null language.
- The **test non-null** command returns 1 (TRUE) if and only if the top network on The Stack encodes a non-null language or relation.
- The **test upper-bounded** command returns 1 (TRUE) if and only if the upper side of the network on the top of The Stack has no epsilon cycles.
- The **test upper-universal** command returns 1 (TRUE) if and only if the upper side of the network on the top of The Stack contains the universal language.
- The **test lower-bounded** command returns 1 (TRUE) if and only if the lower side of the network on the top of The Stack has no epsilon cycles.
- The **test lower-universal** command returns 1 (TRUE) if and only if the lower side of the network on the top of The Stack contains the universal language.

3.6.2 Printing Words and Paths

Print Commands We have already introduced the **print words** utility, which might better be termed *print paths*. When the network on the top of The Stack encodes a language, as in Figure 3.25, the output is just a list of the words in the language.

```

xfst[0]: clear stack
xfst[0]: read regex [ {dog} | {cat} | {horse} ] [s|0] ;
xfst[1]: print words
dog
dogs
cat
cats
horse
horses

```

Figure 3.25: **print words** Enumerates a Language

When the network on the top of The Stack encodes a relation, as in Figure 3.26, **print words** outputs each path, using the notation $\langle \mathbf{u:l} \rangle$ when an arc label has **u** on the upper side and **l** on the lower side.

For any transducer, i.e. any network denoting a relation, use **print upper-words** to see an enumeration of the upper language; similarly, use **print lower-words** to see an enumeration of the lower language.

As networks grow larger, precluding practical enumeration to the terminal of the entire language or relation, you can resort to using **print random-words**, which, as its name suggests, prints a random selection of the paths. **xfst** also provides **print random-upper** and **print random-lower** to display random strings from the upper or lower language, respectively.

Finally, it is sometimes useful to identify the longest string in a network, or its length, and the commands **print longest-string** and **print longest-string-size** are provided.

Variables Affecting Print Commands By default, the **print** commands return valid “words”, i.e. strings to which the network could be successfully applied. These words do not show which sequences of letters are treated as multicharacter symbols.

xfst provides the interface variable **print-space**, set to **OFF** by default, that can be set **ON** to force the various **print** commands to print spaces between symbols. The effect can be seen in the following example:

```

xfst[0]: read regex [ {dog} | {cat} ]
%+Noun:0
[ %+Pl:s | %+Sg:0 ] ;
364 bytes. 8 states, 9 arcs, 4 paths.
xfst[1]: up dog
dog+Noun+Sg
xfst[1]: up cats
cat+Noun+Pl

```

```

xfst[0]: clear stack
xfst[0]: read regex
[ {dog} | {cat} | {horse} ] %+Noun:0 [ %+Pl:s | %+Sg:0 ]
| {ox} %+Noun:0 [ %+Sg:0 | %+Pl:{en} ]
| [ {sheep} | {deer} ] %+Noun:0 [ %+Sg:0 | %+Pl:0 ]
;
xfst[1]: print words
cat<+Noun:0><+Sg:0>
cat<+Noun:0><+Pl:s>
horse<+Noun:0><+Sg:0>
horse<+Noun:0><+Pl:s>
ox<+Noun:0><+Sg:0>
ox<+Noun:0><+Pl:e><0:n>
sheep<+Noun:0><+Pl:0>
sheep<+Noun:0><+Sg:0>
dog<+Noun:0><+Sg:0>
dog<+Noun:0><+Pl:s>
deer<+Noun:0><+Pl:0>
deer<+Noun:0><+Sg:0>

```

Figure 3.26: **print words** Enumerates a Relation

```

xfst[1]: set print-space ON
variable print-space = ON
xfst[1]: up dog
d o g +Noun +Sg
xfst[1]: up cat
c a t +Noun +Sg
xfst[1]: up cats
c a t +Noun +Pl

```

Note that the output, when **print-space=ON**, shows clearly that +Noun, +Sg and +Pl are being treated as multicharacter symbols in the network.

The effect of a similar variable, **show-flags**, will be demonstrated in Chapter 8.

3.6.3 Alphabets of a Network

Printing Alphabets

Each network has two alphabets which are usually distinct: the LABEL ALPHABET and the SIGMA ALPHABET. The label alphabet is the collection of labels that actually appear on arcs in the network. Labels of the form **u:l** overly signal that the network is a transducer, and single-character labels such as **a** are effectively treated like **a:a** in the **Xerox** encoding of networks.

print words print upper-words print lower-words
print random-words print random-upper print random-lower
print longest-string print longest-string-size
print sigma print labels print sigma-tally print label-tally

Figure 3.27: Some Useful Print Commands

The sigma alphabet is the set of individual symbols known to the network; the sigma alphabet never contains labels of the form **u:l** but only the individual symbols **u** and **l**. The example in Figure 3.28 illustrates the use of the **print labels** command to display the label alphabet and the **print sigma** command to display the sigma alphabet. As usual, these commands refer by default to the network on the top of The Stack. If `Myvar` is a defined variable set to a network value, then **print labels Myvar** and **print sigma Myvar** will print the alphabets for that network. The rarely used **print label-tally** and **print sigma-tally** quantify the frequencies of labels and symbols, respectively.

In some cases (see Section 2.3.4), symbols in the sigma alphabet may not actually appear on any arc and so will not appear in the label alphabet. The simplest and most obvious examples involve networks compiled from a regular expression that contains the symbol-complement operator `\` as in Figure 3.29. The network for `[\a]` is shown in Figure 3.30. Note that the symbol **a** occurs in the sigma alphabet but not in the label alphabet. The question mark in the sigma alphabet represents UNKNOWN (or “OTHER”) symbols, and the presence of **a** in the sigma alphabet indicates that **a** is known (and therefore not included in the UNKNOWN symbols).

The question mark (denoting UNKNOWN symbols) must always be interpreted relative to the symbols that are known, i.e. to the concrete symbols that are in the sigma alphabet. For this reason, networks are always associated with a sigma alphabet. In finite-state operations like union and composition that join two networks, the resolution of the respective sigma alphabets is a non-trivial problem that is taken care of by the underlying algorithms.

```

xfst[0]: clear stack
xfst[0]: read regex
[ {dog} | {cat} | {horse} ] %+Noun:0 [ %+Pl:s | %+Sg:0 ]
| {ox} %+Noun:0 [ %+Sg:0 | %+Pl:{en} ]
| [ {sheep} | {deer} ] %+Noun:0 [ %+Sg:0 | %+Pl:0 ]
;
xfst[1]: print labels
a c d e g h o p r s t x <0:n> <+Noun:0> <+Pl:e>
<+Pl:s> <+Pl:0> <+Sg:0>
Size: 18
xfst[1]: print sigma
a c d e g h n o p r s t x +Noun +Pl +Sg
Size: 16

```

Figure 3.28: Displaying the Label and Sigma Alphabet of a Network

```

xfst[0]: clear stack
xfst[0]: read regex \a ;
xfst[1]: print labels
?
xfst[1]: print sigma
? a

```

Figure 3.29: Label Alphabet vs. Sigma Alphabet

Symbol Tokenization of Input Strings

When a network is applied to an input string using **apply up** or **apply down**, the input string is first tokenized into individual symbols (see Section 2.3.6). This tokenization includes the identification of multicharacter symbols, and it must be performed relative to the sigma alphabet of the network being applied. In the following example, the network's sigma includes the multicharacter symbols +Noun, +Pl and +Sg; when the network is applied in a downward direction to the string “dog+Noun+Pl”, the sigma alphabet is consulted and the string is symbol-tokenized into

```
d o g +Noun +Pl
```

before the symbols are matched against upper-side labels in the network.

```
xfst[0]: read regex [ {dog} | {cat} ]
```

Figure 3.30: The Network Encoding the Language $[\backslash a]$

```
%+Noun:0
[ %+Pl:s | %+Sg:0 ] ;
364 bytes. 8 states, 9 arcs, 4 paths.
xfst[1]: sigma
a c d g o s t +Noun +Pl +Sg
Size: 10
xfst[1]: apply down dog+Noun+Pl
dogs
```

Where an input string might be tokenized in multiple ways, the apply routines resolve the ambiguity by processing the input string left-to-right, always selecting at each point the longest possible match. If the sigma includes **+**, **P**, **I**, and **+PI**, the multicharacter symbol will always be chosen over the single symbols. If the sigma includes multiple multicharacter symbols whose spellings start the same way, e.g. **+PI**, **+Plur** and **+Plural**, the longest multicharacter symbol will always have precedence.

The deterministic tokenization of input strings is one of the several reasons to avoid creating multicharacter symbols like **ing**, which are visually indistinguishable from concatenations of single alphabetic symbols. If it looks like a network should match an input string, but doesn't, then you should suspect a problem with multicharacter symbols and tokenization. Use **print sigma** to see what the network's symbols really are.

3.6.4 Printing Information about Networks

When dealing with non-trivial networks, size often becomes an issue. The **print size** command displays the size of a network in terms of states, arcs and paths, the same information that is displayed when a network is compiled using **read regex**. If the network contains a loop, and so contains an infinite number of paths, **print size** indicates that the network is "Circular".

```
xfst[0]: clear stack
xfst[0]: read regex {dog} | {cat} ;
6 states, 6 arcs, 2 paths.
xfst[1]: print size
```



```

6 states, 6 arcs, 2 paths.
xfst[1]: clear stack
xfst[0]: read regex a b* (c) d+ [e|f] ;
5 states, 8 arcs, Circular.
xfst[1]: print size
5 states, 8 arcs, Circular.

```

The output of **print size** can also be directed to a file using **print-size > filename**.

By default, **print size** displays information about the top network on The Stack. If **MyNet** is a variable defined to have a network value, then `print size MyNet` will display the size of that network.

```

xfst[0]: define MyNet [ {dog} | {cat} ] [ s | 0 ] ;
7 states, 7 arcs, 4 paths.
xfst[0]: print size MyNet
7 states, 7 arcs, 4 paths.

```

The **print stack** command, previously introduced in Section 3.2.4, displays size information about all the networks on The Stack. The output of **print stack** can also be directed to a file using **print-stack > filename**. Similarly, use **print defined** and **print-defined > filename** to display information about the set of networks stored in defined variables.

The **print net** command, with the variants **print net defined-variable** and **print net > filename** displays detailed information about a network, including the sigma alphabet, the size (in paths), and other features including the ARITY, where an arity of 1 indicates a simple automaton encoding a language and an arity of 2 indicates a transducer. Finally, **print net** lists each state in the network, followed by notations describing the arcs leading from that state.

```

xfst[0]: clear stack
xfst[0]: read regex [ {dog} | {cat} ] [ s | 0 ] ;
7 states, 7 arcs, 4 paths.
xfst[0]: print net
Sigma: a c d g o s t
Size: 7
Net: EE2D8
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0:   c -> s1, d -> s2.
s1:   a -> s3.
s2:   o -> s4.
s3:   t -> fs5.
s4:   g -> fs5.

```

```
fs5:  s -> fs6.
fs6:  (no arcs)
```

In the listing of states and arcs, notations such as **s0**, **s1** and **s2** indicate non-final states. The state numbered 0 is the start state, and the other numbering is more or less arbitrary though used consistently in the output. The line

```
s0:  c -> s1, d -> s2.
```

indicates that state 0 has two arcs leading from it: one is labeled **c** and leads to state 1; and the other is labeled **d** and leads to state 2. The notations **fs5** and **fs6** indicate final states.

3.6.5 Inspection of Networks

The **print net** command, which identifies all the states and arcs of a network, is useful for visualizing fairly small nets. However, a full-sized net for natural language processing frequently contain hundreds of thousands of states and arcs, and the output from **print net** would be overwhelming. To explore a larger network, traversing it state by state, following the arcs, use the **inspect net** command. As **inspect net** is primarily intended for expert users, and particularly for **Xerox** developers debugging new finite-state algorithms, we will not go into the details here. If you want to experiment with network inspection, see **help inspect net** and the command summaries displayed when you invoke **inspect net**.

3.7 Miscellaneous Operations on Networks

3.7.1 Substitution

Substitution Commands

The various **substitute** commands are a powerful method for modifying networks.

- The **substitute label** command is built on the following template

```
substitute label list-of-labels for label
```

It replaces the top network on The Stack with a network derived by replacing every arc with the given label by an arc or arcs, each labeled with one of the substitute labels. If the list consists of the keyword **NOTHING**, all paths containing the given label are removed. The label and the substitutes may be single symbols or a symbol pairs.

The following example would replace all instances of the label **V** with the labels **a**, **e**, **i**, **o** and **u**.

```
xfst[1]: substitute label a e i o u for V
```

Note that this command, as written, will not affect any labels such as **V:x** or **y:V**, where **V** is not the whole label but only a symbol on one side of an upper:lower label.

- The **substitute symbol** command is built on the following template

```
substitute symbol list-of-symbols for symbol
```

It replaces the top network on The Stack with a network derived by replacing every arc whose label contains the given symbol by an arc or arcs whose label instead contains one of the substitute symbols. If the list consists of the keyword NOTHING, all paths containing the given symbol are removed. The symbol and the substitutes must be single symbols, not symbol pairs.

The following example would replace all instances of the symbol **V** with the symbols **a, e, i, o** and **u**.

```
xfst[1]: substitute symbol a e i o u for V
```

Note that this command, in contrast to the **substitute label** command, will affect labels such as **V:x** and **y:V** where **V** is a symbol in a label. It will also affect the label **V**, which, in the **Xerox** implementation of networks, encodes the relation **V:V**.

The **substitute symbol** command can also be indicated in regular expressions as

```
`[ [A], S, L ]
```

where **A** is the network to be affected, **S** is the symbol to be replaced, and **L** is the list of replacement symbols, e.g.

```
`[ [ @"MyNetwork.fst" ], V, a e i o u ]
```

The regular-expression syntax is not much used.

- The **substitute defined** command is built on the following template

```
substitute defined defined-variable for label
```

It replaces the top network on the stack with a network derived by splicing in the network associated with the given defined variable for each arc that has the indicated label. If the network is a transducer, the label must not be a symbol that occurs in a (non-identity) symbol pair.

The **substitute** commands provide powerful, and probably under-appreciated, ways to modify networks. The **substitute defined** command, in particular, allows networks to be built with single placeholder labels that can later be substituted with whole networks (see Section 10.5.4). Different versions of a common system could be produced by building a master network with placeholder labels, and then substituting in different networks for the placeholder labels.

Substitution Applications

Shuffling The **shuffle net** function, which is invoked by the binary operator `<>` in regular expressions, was developed some years ago to shuffle the individual letters of the strings of one language with the individual letters of the strings of a second language. For example,

```
xfst[0]: read regex [ a b <> c d ] ;
9 states, 12 arcs, 6 paths.
xfst[1]: print words
acdb
acbd
abcd
cadb
cabd
cdab
```

However, shuffling found little practical use until we looked at some American Indian languages and saw suffix classes that could appear in free relative order. When **shuffle** is used with individual symbols, the result is a language of strings that contains all permutations of those symbols, i.e. they appear in free relative order.

```
xfst[0]: read regex [ Suff1 <> Suff2 ] <> Suff3 ;
8 states, 12 arcs, 6 paths.
xfst[1]: words
Suff1Suff3Suff2
Suff1Suff2Suff3
Suff2Suff3Suff1
Suff2Suff1Suff3
Suff3Suff1Suff2
Suff3Suff2Suff1
```

Once such permutations are encoded in a network with individual multicharacter labels like **Suff1**, **Suff2** and **Suff3**, and if the network is left on the top of the stack, one can then use **substitute defined** to replace them with networks of arbitrary complexity representing the corresponding suffix classes.

```

xfst[1]: define S1 [ k o n | d i i | f o t ] ;
xfst[1]: substitute defined S1 for Suff1
xfst[1]: define S2 [ l l a t | b o o | t e e ] ;
xfst[1]: substitute defined S2 for Suff2
xfst[1]: define S3 [ x o n | l m a n | k o z ] ;
xfst[1]: substitute defined S3 for Suff3

```

In this example, the resulting network contains 162 paths, representing the possible permutations of all the individual suffixes in the three classes.

Adding Roots to a Network The following script defines a network that accepts a language of words built on two roots, *raam* and *cazard*.

```

! lexicon script

clear stack
define Roots {raam} | {cazard} ;
define Suff1 {ze} | {azuu} ;
define Suff2 {lazam} | {mule} | {kiba} ;
read regex Roots Suff1 Suff2 ;
save stack Lex.fst

```

To add the two new roots *zalam* and *gexar*, you usually have to edit the script and change the `define Roots` line, e.g.

```
define Roots {raam} | {cazard} | {zalam} | {gexar} ;
```

and then recompile everything from scratch.

In some simple cases, like this one, it's possible to add new roots to a compiled lexicon without recompilation. The trick is to introduce a placeholder multicharacter symbol such as `!*ROOTPLACEHOLDER!*`, as in the following script.

```

! lexicon script

clear stack
define Roots {raam} | {cazard} | !*ROOTPLACEHOLDER!* ;
define Suff1 {ze} | {azuu} ;
define Suff2 {lazam} | {mule} | {kiba} ;
read regex Roots Suff1 Suff2 ;
save stack Lex.fst

```

This will add meaningless strings like “`*ROOTPLACEHOLDER*zamazam`” to the language, but if the placeholder symbol is given a bizarre name like this one, preferably with some literal punctuation symbols thrown in, the bizarre strings won't match any normal input. Now we can add new roots to the lexicon via substitution, e.g.

```

xfst[0]: define New {zalam} | {gexar} |
%*ROOTPLACEHOLDER%* ;
xfst[0]: load stack Lex.fst
xfst[1]: substitute defined New for %*ROOTPLACEHOLDER%*

```

The resulting network now contains four roots plus a new copy of the placeholder, in case some more new roots need to be added later. Test these examples, using **print words** to see the language before and after the **substitute** command.

It should be noted that this substitution trick, avoiding a recompilation, works only if the roots are unaffected by subsequently composed alternation rules or other similar processing; the general ability to add new roots to a complex lexical transducer is a non-trivial problem. Nevertheless, the substitution trick shown here has proved useful in certain applications.

3.7.2 Network Properties

It is often useful to attach documentation to a network, including a version number, the date of creation, the names of the authors, etc. This is especially important for commercial products and any other networks that are delivered for use in other projects. **xfst** therefore allows an arbitrary number of feature:value pairs called **PROPERTIES** to be associated with a network, where the feature and value are strings chosen by the developer. Properties do not affect the functioning of a network in any way.

The **read properties** < *filename* command reads a set of feature:value pairs from a text file and stores them in the top network on The Stack. If you don't specify a filename, **xfst** will expect you to enter feature:value pairs at the terminal. The input should be of the format

```

FEATURE1: VALUE1
FEATURE2: VALUE2
FEATURE3: VALUE3
etc.

```

where **FEATURE_n** and **VALUE_n** are strings, and they must be enclosed by double quotes if they contain spaces. Each feature:value pair must reside on a single line. Such files are typically created with a text editor like **xemacs** or **vi**.

The following is a slightly edited version of a real property file used to mark a network that performed morphological analysis for Italian.

```

LANGUAGE: ITALIAN
CHARENCODING: ISO8859/1
VERSIONSTRING: "1.2"
COPYRIGHT1: "Copyright (c) 1994 1995 Xerox Corporation."
COPYRIGHT2: "All rights reserved."
TIMESTAMP: "29 November 1995; 16:30 GMT"

```

```
AUTHOR1: "Antonio Romano"
AUTHOR2: "Cristina Barbero"
AUTHOR3: "Dario Sestero"
AUTHOR4: "Kenneth Beesley"
```

The **read properties** command overwrites any previously set properties.

The **add properties** command is like **read properties**, reading feature:value pairs from a file or the terminal, but the new pairs are added to any existing pairs rather than overwriting them.

The **edit properties** command initiates a dialog that allows the developer to edit the property list manually. The dialogs are a bit awkward, so most developers prefer to edit feature:value pairs in a text file and call **read properties** or **add properties**.

Finally, **write properties** > *filename* writes the properties of the top network on The Stack out to the designated file in the format required by **read properties**. If you do not specify a filename, the output is displayed on the terminal. Use **write properties** *defined-variable* to write the properties of a network stored in a defined variable.

3.7.3 Housekeeping Operations

The housekeeping operations listed in Table 3.28 are rarely used by developers. For example, after a call to an algorithm like **union net**, the result is automatically pruned, epsilon-removed, determinized and minimized before being pushed onto The Stack. See the **help** messages for more information.

cleanup net
complete net
determinize net
epsilon-remove net
minimize net
prune net
sort net
compact sigma

Table 3.28: Housekeeping Operations

3.7.4 Generation of Derived Networks

The operations listed in Table 3.29 produce various kinds of derived networks from the top network on The Stack. See the **help** messages for more information.

label net name net sigma net substring net

Table 3.29: Commands to Build Derived Networks

3.8 Advanced xfst

3.8.1 Word-Number Mapping

Any loop-free network contains a finite number n of paths, and such finite-state networks can assign a unique integer i , where $0 \leq i < n$, to each word. With appropriate word-number mapping routines, the network maps from words to numbers and numbers to words. This capability can be used to implement perfect hashing.

The **print nth-upper** and **print nth-lower** commands map from integers to numbers. The upper-lower distinction is necessary for transducers; with simple automata that encode languages, the commands are equivalent.

```

xfst[0]: clear stack
xfst[0]: read regex {cant}
[ o | a s | a t | a m u s | a t i s | a n t ] ;
6 states, 10 arcs, 6 paths.
xfst[1]: print nth-lower 0
canto
xfst[1]: print nth-lower 1
cantas
xfst[1]: print nth-lower 2
cantamus
xfst[1]: print nth-lower 3
cantant
xfst[1]: print nth-lower 4
cantat
xfst[1]: print nth-lower 5
cantatis
xfst[1]: print nth-lower 6
Argument too large: 6.

```

Note in this example that the network includes 6 paths and that the numbered words, which happen to look like words of Latin, range from 0 to 5.

The **print num-upper** and **print num-lower** routines map from words to numbers.


```

xfst[0]: clear stack
xfst[0]: read regex {cant}
[ o | a s | a t | a m u s | a t i s | a n t ] ;
6 states, 10 arcs, 6 paths.
xfst[1]: print num-lower cantant
3
xfst[1]: print num-lower cantatis
5
xfst[1]: print num-lower cantamus
2
xfst[1]: print num-lower cantat
4
xfst[1]: print num-lower cantas
1
xfst[1]: print num-lower canto
0
xfst[1]: print num-lower cantabimus

```

For the final input shown, “cantabimus”, there is no output because the string is not included in the language of the network.

Starting from a regular expression, there is no general way to predict or force which numbers will be assigned to which words, but obviously the word-number mapping will be consistent in both directions. If you build a network from an alphabetically pre-sorted wordfile using **read text** (see Section 3.3.1) then the word-number mapping will match the order of the words in the sorted file, with the first word in the file numbered 0.

3.8.2 Modifying the Behavior of xfst

Quite separate from the defined variables that hold network values, there is a fixed predefined set of internal interface variables that control the behavior of **xfst**. You can see a list of the internal variables, and the commands that query and reset them, by entering **apropos variable**.

```

xfst[0]: apropos variable
set                : sets a new value to the variable
show               : show the value of the variable
assert            : OFF : quit if a test fails and quit-on-fail is ON
flag-is-special   : OFF : treat flags as epsilons in composition
minimal           : ON  : minimize the result of calculus operations
obey-flags        : ON  : enforce flag diacritics
name-nets         : OFF : use regular expressions as network names
print-pairs       : OFF : show both sides of labels in apply
print-sigma       : ON  : show the sigma when a network is printed
print-space       : ON  : insert a space between symbols in printing words
recursive-define  : OFF : allow self-reference in definitions

```

```

recursive-apply : OFF : work depth-first in 'compose-apply'
quit-on-fail    : OFF : quit abruptly on any error
show-flags     : OFF : show flag diacritics when printing
quote-special  : OFF : print special characters in double quotes
random-seed    :      : seed of the random number generator.
retokenize     : ON  : retokenize regular expressions in 'compile-
replace'
sort-arcs      : ON  : sort the arcs before printing a network
verbose        : ON  : print messages

```

We have already seen **print-space** in Section 3.6.2. We will eventually learn about some of the others, including **obey-flags**, **show-flags**, **retokenize** and **flag-is-special**, in coming chapters.

To display short documentation about a particular internal variable, use **help** as usual, e.g.

```

xfst[0]: help obey-flags
variable obey-flags == ON|OFF

```

when ON, the constraints expressed as flag diacritics are taken into account by 'print [upper|lower]-words' and 'print random-[upper|lower]' commands. When OFF, flag diacritics are treated as ordinary symbols in listing the contents of a network. Default is ON. Current value is ON.

To show the current value of a variable, use **show variable-name**, e.g.

```

xfst[0]: show obey-flags
variable obey-flags = ON

```

To reset the value of an internal variable, use **set variable-name new-value**, e.g.

```

xfst[0]: set obey-flags OFF
variable obey-flags = OFF

```

As shown, **xfst** will reset and reflect the new value.

3.8.3 Command Abbreviations

You will have noted that **xfst** commands consist of two or even three separate words. However, almost all the commands can be entered as a single word following the examples shown in Table 3.30. Notice that the abbreviated command consists of the content word of the full command, dropping the qualifying words such as **apply**, **print**, and **net**.

Full Command	Abbreviation
apply up	up
apply down	down
clear stack	clear
load stack	load
save stack	save
rotate stack	rotate
print words	words
print upper-words	upper-words
print lower-words	lower-words
print random-words	random-words
print random-upper	random-upper
print random-lower	random-lower
print labels	labels
print sigma	sigma
test equivalent	equivalent
compose net	compose
concatenate net	concatenate
intersect net	intersect
invert net	invert
negate net	negate
reverse net	reverse
union net	union

Table 3.30: Some Common Command Abbreviations

3.8.4 Command Aliases

xfst includes a simple kind of macro capability called ALIASES. An alias definition consists of the keyword **alias** followed by a user-chosen alias name, a newline, a series of **xfst** commands, and then *END*; to terminate. After entering the alias name and newline, a special prompt is displayed to remind you that you are defining an alias. The alias name should consist of plain alphabetic letters.

```
xfst[0]: alias myaliasname
alias> load lexicon.fst
alias> print sigma
alias> END;
xfst[0];
```

The alias name can then be entered like any other **xfst** command, e.g.

```
xfst[0]: myaliasname
```

An alias is thus like a script file that can be invoked without the **source** command.

An alternative definition format puts the commands on the same line as the **alias** command, and the alias is then terminated by the newline. In this single-line format, multiple commands must be separated by semicolons.

```
xfst[0]: alias myaliasname load lexicon.fst; print sigma
xfst[0]:
```

To see which aliases have already been defined, type **print aliases**. The defined aliases can also be output to file using **print aliases > filename**.

3.8.5 xfst Command-Line Options

If you invoke **xfst -h** from the operating-system command line, you will see the following list of command-line options.

```
unix> xfst -h
usage: xfst [-pipe] [-flush] [-q] [-v] [-V]
[-f scriptfile] [-l startscript]
[-o output_buffer_size] [-s fsm-file] [-e commands] [-stop]
```

To print out the version number, use **-v**. The **-V** flag is obsolete.

```
unix> xfst -v
```

To invoke **xfst** and run a startup script, use the **-l** flag. **xfst** will perform the script and then go into normal interactive mode. Multiple startup scripts can be indicated using multiple **-l** flags.

```
unix> xfst -l startup_script
xfst[0]:
```

To run an **xfst** script file and then exit immediately back to the operating system, use `-f`.

```
unix> xfst -f script
```

This is equivalent to

```
unix> xfst
xfst[0]: source script
xfst[0]: exit
```

To indicate an **xfst** command on the operating-system command-line itself, use the `-e` flag. The commands themselves must be enclosed in double quotes if they contain spaces or other symbols that could confuse the operating system. Multiple commands can be specified with multiple `-e` flags. Use `-stop` if you want **xfst** to exit back to the operating system immediately after performing the indicated commands.

```
unix> xfst -e "regex d o g | c a t | m o u s e ;" \
-e "print words" -e "apply up mouse" -stop
10 states, 11 arcs, 3 paths.
dog
cat
mouse
mouse
unix>
```

Use `-q` to indicate “quiet mode”, suppressing interactive messages. This is particularly appropriate when running **xfst** as a background process. The `-stop` flag tells **xfst** to terminate the background process, e.g.

```
unix> xfst -q -e "load myfile.fst" -e "apply up dog" \
-stop > outputfile
```

The backslash in this example indicates to the Unix-like operating system that the command continues on the next line. The command could also be typed on a single line, in which case no backslash should be used.

Here’s another example that invokes **xfst**, loads a network, analyzes a whole list of words from a tokenized file, and then stops.

```
unix> xfst -q -e "load myfile.fst" \
-e "apply up < myInput" -stop > outputfile
```

More exotic flags include `-pipe` which causes **xfst** to take commands as usual, but without printing command prompts. The flag `-flush` flushes the output buffer after each print command without waiting for an end-of-line character.

:	High-precedence crossproduct operator
~\ \$ \$? \$.	Complement, symbol complement, containment
+ * ^ .1 .2 .u .l .i .r	Kleene plus and star, iteration, upper-lower, invert and reverse
/	Ignore
	Concatenation (no overt operator)
> <	Precede and follow
& -	Union, intersect, minus
=> -> (->) @-> etc.	Rule operators
.x. .o.	Crossproduct and compose

Table 3.31: Precedence of Operators from High to Low

3.9 Operator Precedence

In regular expressions as in arithmetic expressions, operators have to be assigned a precedence. For example, in the unbracketed arithmetic expression $3 + 4 * 2$, multiplication conventionally has precedence over addition, resulting in a value of 11. It is obvious that the precedence is important because if the addition were perversely performed first, the answer would be 14. If in fact the intent of the author is to have the addition performed first, then that interpretation would need to be forced by parenthesizing the expression as $(3 + 4) * 2$.

The **xfst** regular-expression operators have the precedence shown in Table 3.31, which descends from high precedence at the top to low precedence at the bottom. In regular expressions, to force particular subexpressions to be performed before others, they can be grouped using square brackets []. Remember that parentheses in **xfst** regular expressions denote optionality. Note also that there are two crossproduct operators in **xfst** regular expressions, the colon (:) which has high precedence, and the .x. , which has low precedence.

3.10 Conclusion

xfst is a large and powerful tool for finite-state development. In addition to stack-based operations, it includes a compiler for regular expressions in the **Xerox** format, which includes powerful abbreviated notations such as replace rules. You should now be comfortable with regular expressions and the basic commands in **xfst**; and you should know how to use **help** and **apropos** to explore the rich and somewhat overwhelming set of options. Mastery of **xfst** will come only after considerable practice.

In practical development, **xfst** is usually used together with **lexc** (Chapter 4) to build complete systems.

Part II

Finite-State Compilers

Chapter 4

The Finite-State Lexicon Compiler: `lexc`

Contents

4.1	Introduction	212
4.2	Defining Simple Automata	213
4.2.1	Invoking <code>lexc</code>	213
4.2.2	LEXICON Root	216
4.2.3	<code>lexc</code> and Finite-State Networks	218
4.2.4	Continuation Classes	221
4.2.5	Multiple Classes of Words	225
4.2.6	Optionality and Loops	225
4.2.7	Alternation	227
4.2.8	Exercises	228
	Esperanto Nouns	228
	A Little History	228
	The Facts	230
	The Task	231
	Testing	232
	Reduce the Task to Lexicography	233
	Esperanto Adjectives	234
	Background	234
	The Facts	234
	The Task	235
	Testing	235
	Reducing the Task to Lexicography	236
	Nouns and Adjectives	236
4.3	Defining Lexical Transducers	238

4.3.1	<i>Upper:Lower</i> Entries	238
4.3.2	Epsilons	239
4.3.3	Regular-Expression Entries	240
4.3.4	Multicharacter Symbols	242
4.3.5	Exercises	244
	Esperanto Nouns and Adjectives with Upper-Side Tags	244
	Esperanto Verbs	245
	The Facts	245
	The Task	246
	Esperanto Verbalization	247
	Bambona	247
	Simplified Germanic Compounding	248
4.4	The lexc Interface	251
4.4.1	Introduction	251
4.4.2	lexc Registers	251
4.4.3	Command Menu	252
4.4.4	Individual lexc Commands	253
	Input-Output Commands	253
	The Source Register	253
	The Rules Register	253
	The Result Register	253
	Properties	254
	Operations among Registers	254
	Composition	254
	Checking	255
	Switches	255
	Scripts	257
	Display	257
	Miscellaneous	257
	Help	257
	Quit/Exit	258
	quit	258
4.5	Useful lexc Strategies	258
4.5.1	Compiling lexc Files Directly from xfst	258
4.5.2	The Art and Craft of lexc	258
4.5.3	Expert Morphotactics	258
	Defining Sublexicons	258
	All Morphemes are Organized into LEXICONS	258
	A LEXICON Should be Coherent	259
	A LEXICON is a Potential Continuation Target	260

	Refining LEXICON Targets	260
	Choosing and Using <code>Multichar_Symbols</code>	261
	Symbol Tags	261
	Some General Principles for Tag-Name Choice	264
	Separated Dependencies	264
	The Problem of Separated Dependencies	264
	Constraining Separated Dependencies with Fil-	
	ters	267
	replace rules as Filters	272
	Exercises	273
4.5.4	Properties	274
4.5.5	lexc Strategy Summary	275
4.5.6	Common lexc Errors	275
	Error: Putting a Semicolon after the LEXICON Name	275
	Error: Forgetting to Put a Semicolon after an Entry	276
	Error: Failure to Declare a Multicharacter Symbol	276
	Error: Trying to do Concatenation in an Entry Line	277
4.6	Integration and Testing	278
4.6.1	Mind Tuning for lexc Integration	278
	Thinking about Languages	278
	Thinking about Transducers	280
	Thinking about Rules	283
	Thinking about Composition	284
	Intermediate Languages	284
	Comparing Composition with UNIX Pipes	285
	Composing Transducers in xfst	286
	Composing Lexicons and Rules in lexc	289
	The Intersecting-Composition Algorithm	289
4.6.2	The check-all Utility	291
	Introduction	291
	What Happens when Rules are Composed	292
	Output from check-all	293
	Switch Settings	294
	Using check-all after Composition in xfst	294
	Finding Ambiguities and/or Duplicates in a Single FST	295
4.6.3	Exercises	296
	Irish Gaelic Lenition	296
	Background	296
	The Data	296
	The Task	297

4.7	lexc Summary and Usage Notes	298
4.7.1	Mindtuning	298
	Input	298
	Output	298
	lexc Interface	298
	Avoiding the lexc Interface	299
	Right-Recursive Phrase-Structure Grammar	299
4.7.2	lexc Syntax	299
	Multichar_Symbols Declaration	299
	Comments	299
	LEXICONS	299
	LEXICON Root	299
	User-Named LEXICONS	300
	Entries	300
	Entry Fields	300
	Data Formats	300
4.7.3	Special Attention	303
	Special Characters	303
	Explosion vs. Multicharacter Symbols	304
	Failure to Declare Multicharacter Symbols	305

4.1 Introduction

lexc, for **Lexicon Compiler**, is a high-level declarative programming language and associated compiler for defining finite-state automata and transducers; as its name indicates, it is particularly well suited for defining natural-language lexicons. In a traditional **Xerox** morphological analyzer, **lexc** is used to describe the morphotactics of a language while **twolc** (Chapter 5) is used to describe the phonological and orthographical alternation rules. More recently, **Xerox** linguists have increasingly adopted **xfst** replace rules (see Sections 2.4.2, 3.5.2 and 3.5.5) as an alternative to **twolc** rules.

Formally, the **lexc** language is a kind of right-recursive phrase-structure grammar. The **lexc** compiler was written in C at **Xerox PARC** in 1992-93 by Lauri Karttunen and Todd Yampol (Karttunen, 1993). It is based on Karttunen's 1990 Common Lisp implementation of a similar utility.

The input to **lexc** is a text file in a format similar to that accepted by Kimmo Koskenniemi's *TwoL* program (Koskenniemi, 1983; Koskenniemi, 1984) and Evan Antworth's (Antworth, 1990) PC-KIMMO (version 1), but with several novel features. One innovation is that individual lexical entries can now specify a two-level relation between forms. Suppletions (gross irregularities) and idiosyncratic mappings such as the *i-a* alternation of *swim* to *swam* can now be coded directly in the

lexicon whereas regular and productive alternations, such as the loss of the stem-final *e* from *dine* to *dining* may be handled by general rules. Another innovation allows **lexc** entries to define finite-state subnetworks using the regular-expression notation.

A **lexc** description compiles into a standard **Xerox** finite-state network, either a simple automaton or a transducer. Inside the **lexc** interface, this network can be subsequently composed with rule transducers, compiled with **twolc** or **xfst**, that perform alternations, filter the morphotactics or in other ways modify the initial **lexc** result. The networks compiled from **lexc** can also be saved to file, loaded into **xfst**, and manipulated like any other networks; **xfst** can even compile **lexc** files directly, using the **read lexc** command.

The presentation proceeds as follows:

- Section 2, entitled *Defining Simple Automata*, is a brief tutorial on using **lexc** to create basic finite-state lexicons that encode languages.
- Section 3, entitled *Defining Lexical Transducers*, completes the presentation of **lexc** syntax and leads to more complicated examples and exercises.
- Section 4, entitled *The lexc Interface*, looks in more detail at the various finite-state algorithms available from the **lexc** interface and how they can be used to test and manipulate your networks.
- Section 5, entitled *Useful lexc Strategies*, explains strategies for more sophisticated usage of **lexc**, including the principled division of data into sublexicons, the choice and consistent usage of multicharacter-symbol tags, and the proper use of lexical overgeneration and filters.
- Section 6, entitled *Integration and Testing*, examines how **lexc** networks fit into larger systems and how they can be tested using finite-state techniques.
- Section 7, entitled *lexc Summary and Usage Notes*, is intended as a summary for those who need to switch back and forth between **xfst** and **lexc** and need a quick review of **lexc** syntax, semantics, and special characters.

In most practical systems, **lexc** and **xfst** are used together build a final LEXICAL TRANSDUCER, and this chapter will discuss both languages as appropriate.

4.2 Defining Simple Automata

4.2.1 Invoking lexc

The best way to learn **lexc** is by working with it. The **lexc** interface is invoked from the command line by entering **lexc**.

```
unix> lexc
```

As shown in Figure 4.1, the **lexc** interface responds with a welcome banner, a menu of **COMMANDS** that can be invoked, and a **lexc** prompt. It then waits for you to enter a command, terminated with a carriage return, at the prompt line. The commands are grouped under three general headings: Input/Output, Operations, and Display. Each group is further divided into subsections of related commands for easier reference. Don't be overwhelmed by all the choices; only a few of them will be required for our first few examples and exercises. You can enter a question mark (?) at any time to have the command menu redisplayed, and you can also enter **help** *commandname* at any time to see short documentation of what a particular command does.

For example, entering

```
lexc> help compile-source
```

causes the following terse but useful summary to be displayed:

```
'compile-source <textfile>' reads the lexc source file contained in
<textfile>, compiles it, and stores the resulting network in the
SOURCE register. You can save this network to file with the command
'save-source'. To read a pre-compiled network from file into the
SOURCE register, use the command 'read-source'. To compose the SOURCE
network with a set of rule transducers, previously read in by
'read-rules', use the command 'compose-result'.
```

The utilities that you will need to start are

- **compile-source**: Use this command to read a textfile, containing a **lexc** description, and compile it into a finite-state network. The network will be placed in a buffer known as the **SOURCE REGISTER**. For example, if you have edited a **lexc** description and saved it in file *mylanguage-lex.txt*, you compile it in **lexc** with the command:

```
lexc> compile-source mylanguage-lex.txt
```

If you do not specify a filename, **lexc** will prompt you for one. The **compile-source** command can also be passed multiple source files, which it will combine and treat as a single **lexc** source description, producing one network as the result.

```
lexc> compile-source part1-lex.txt part2-lex.txt part3-
lex.txt
```

It is a common convention at **Xerox** to give names ending -*lex.txt* or -*lexc.txt* to **lexc** source files, e.g. *french-lex.txt* or *afrikaans-lex.txt*.

```

*****
*   Finite-State Lexicon Compiler 3.2.1 (7.9.1)   *
*                   created by                   *
*                   Lauri Karttunen and Todd Yampol *
*   Copyright © 1993-2002 by the Xerox Corporation. *
*                   All Rights Reserved.         *
*****

Input/Output -----
  Source:          compile-source, merge-source, read-
source,
                  result-to-source, save-source.
  Rules:           read-rules.
  Result:          merge-result, read-result, save-result,
                  source-to-result.
  Properties:     add-props, reset-props, save-props.
Operations -----
  Composition:    compose-result, extract-surface-forms,
                  invert-source, invert-result.
  Checking:       check-all, lookdown, lookup, random,
                  random-lex, random-surf.
  Switches:       ambiguities, duplicates, failures,
                  obey-flags, print-space, quit-on-fail,
                  show-flags, singles, time.
  Scripts:        begin-script, end-script, run-script.
Display -----
  Misc:           banner, labels, props, status, storage.
  Help:           completion, help, history, ?.
Type 'quit' to exit.

lexc>

```

Figure 4.1: The Welcome Banner and Command Menu of **lexc**

- **lookup**: Once you have successfully compiled a source file, and the result is stored in the source buffer, you can test it with the **lookup** utility. This **lookup** works like **apply up** in **xfst**; it attempts to look up or *analyze* the input string using the network in the SOURCE register.

```
lexc> lookup elephant
```

- **lookdown**: Similarly, **lookdown** works like **apply down** in **xfst**; it attempts to look down or *generate* the input string using the network in the SOURCE register.

```
lexc> lookdown cantar+Verb+PresInd+1P+Sg
```

- **save-source**: Save the binary network in the source buffer to file with **save-source**.

```
lexc> save-source mylanguage-lex.fst
```

If you do not supply a filename, the interface will prompt you for one.

The usual **Xerox** convention is to give the binary output file a name ending *-lex.fst*, e.g. *french-lex.fst*. You do not, of course, have to use **Xerox** file-naming conventions, but consistency and the use of mnemonic names are highly encouraged.

- **quit**: Use **quit** to exit from **lexc**.

```
lexc> quit
```

These five commands are all that we will need to get started. We'll introduce others as we need them.

4.2.2 LEXICON Root

Let's start off with a minimal example. You are encouraged to type in the examples yourself to help you get the feel of the **lexc** syntax and interface. Use your favorite text editor to create a source file, called something like *ex1-lex.txt*, that looks like Figure 4.2; then invoke **lexc** itself, and compile the source file using the **compile-source** command.

The complete **lexc** program shown in Figure 4.2 consists of a single LEXICON, named *Root*, three ENTRIES, and a comment. Every **lexc** program should


```
! ex1-lex.txt

LEXICON Root
dog      # ;
cat      # ;
bird     # ;
```

Figure 4.2: A Simple **lexc** Program

have a `LEXICON Root`; it marks the start state of the network to be compiled. In general, there may be any number of LEXICONS, but exactly one should have the reserved name `Root`.

There are several things to note about the entries:

- Each entry consists of two parts, a `FORM` and a `CONTINUATION CLASS`. In Figure 4.2, the form in the first entry is `dog` and the continuation class is `#`, a reserved **lexc** symbol indicating the end of word. We'll have more to say about continuation classes later.
- Each entry is terminated with a semicolon (`;`), but note that there is *not* a semicolon after the LEXICON name.
- **lexc** by default interprets the three entry forms as consisting of separate symbols, i.e. `dog` is interpreted like the regular expression `[d o g]`, `cat` as `[c a t]`, and `bird` as `[b i r d]`. In other words, **lexc** “explodes” each string entry by default into separate symbols, quite unlike the strings in regular expressions compiled by **xfst**.
- Comments in **lexc** are introduced with an exclamation mark (`!`) and extend to the end of the line. You are encouraged to use lots of them.

lexc syntax is fairly free-form, and multiple entries can be placed on the same line as in Figure 4.3, although this seldom improves readability. The reserved word `END` can optionally be placed at the end of the file as in Figure 4.4. Any text after the `END` command is ignored. Because `END` is a reserved word in **lexc**, you cannot use it for any other purpose, e.g. as the name of a LEXICON. The pound sign (`#`) and the semicolon (`;`) are both special characters for **lexc**; to indicate a literal pound sign or semicolon, you must precede them with the literalizing percent sign (`%#` and `%;`). To indicate a literal percent sign, use `%%`.

```
! ex2-lex.txt (equivalent to ex1-lex.txt)

LEXICON Root
dog      # ;   cat      # ;   bird      # ;
```

Figure 4.3: Multiple Entries on a Line

```
! ex3-lex.txt

LEXICON Root
dog      # ;
cat      # ;
bird     # ;

END

this text will be ignored
```

Figure 4.4: Optional **END**. Any text after the **END** keyword is ignored.

4.2.3 **lexc** and Finite-State Networks

The result of a **lexc** compilation is a finite-state network in **Xerox** standard format that is completely compatible with networks built in **xfst** with the **read regex** utility (see Section 3.2.2). In fact, **lexc** is just another formalism for specifying finite-state networks.

Consider again our first **lexc** example, reprinted here as Figure 4.6. The result of **compile-source** is an automaton that looks like Figure 4.7. Type this example into a file, using a text editor, compile it using **compile-source**, and perform the following experiments:

1. Perform **lookup** on the word “dog”. It should succeed, returning the same string “dog” as the result.

```
lexc> lookup dog
```

Lookup should also succeed for the words “cat” and “bird”. Also try **lookdown**. The results should be the same.

```

! ex4-lex.txt

LEXICON Root
dog      # ;
cat      # ;
bird     # ;
%#       # ; ! a literal pound-sign entry
%;foo    # ; ! contains a literal semicolon
20%%     # ; ! contains a literal percent sign

```

Figure 4.5: Notating Literal Pound Signs, Semicolons, and Percent Signs

```

! ex1-lex.txt

LEXICON Root
dog      # ;
cat      # ;
bird     # ;

```

Figure 4.6: A Simple **lexc** Program

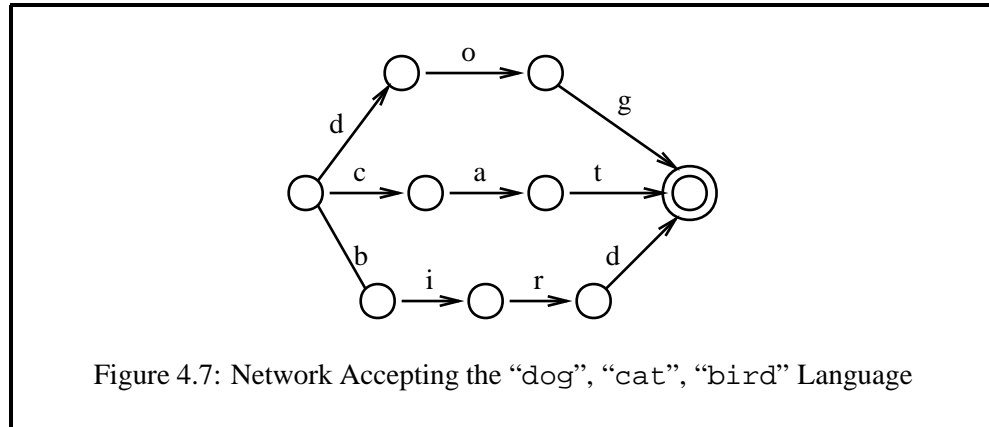
2. Perform **lookup** on the word “elephant”. It should fail, returning a string of asterisks.
3. Invoke the **random** utility inside **lexc**. It should print out a random list of words from the network, including “dog”, “cat” and “bird”.

```
lexc> random
```

Save the network to file using **save-source** and quit **lexc** (using the **quit** command).

```
lexc> save-source ex1-lex.fst
lexc> exit
```

4. Enter **xfst** and use **load stack** to read the same network from the file onto the **xfst** stack. Test it using **apply up** and **apply down**. The behavior of the network should be the same in **xfst** and **lexc**. Quit from **xfst**.



```
lexc> load stack ex1-lex.fst
```

5. Use your text editor to open a new source file and define the same automaton with a regular expression. It should look something like

```
d o g | c a t | b i r d ;
```

Remember that in **Xerox** regular expressions, symbols must be separated by white space, and the entire regular expression must be terminated with a semicolon and a carriage return, even when the regular expression is typed into a file. Invoke **xfst** again and compile the regular expression using the **read regex <** facility (Section 3.3.1). Test it using **apply up** and **apply down**. Then call **load stack** to read in the binary file created by **lexc**. You should now have two networks on the stack. Invoke **test equivalent** (see Section 3.5.4) to see if the two networks are indeed equivalent; they should be.

6. Re-edit the **lexc** source file and add the following words: *elephant, horse, hippopotamus, aardvark, snake, donkey, monkey, deer, mastodon, tiger, lion, cow, baboon, pigeon, fish, whale, shark, rat, bat, gazelle, gorilla, stingray, earthworm* and *butterfly*. Recompile it in **lexc** using **compile-source** and test the result.
7. Re-edit the regular-expression source file and add the same new words. Remember that in regular expressions you need to put spaces between all the separate symbols or to surround the strings with curly braces, e.g. {elephant}. Compile the regular expression, using **read regex <** inside **xfst**, and test the result.

Even though you can define completely compatible and equivalent networks using either **xfst** or **lexc**, you may have found that adding just two dozen words

to a regular expression (including all the required spaces between characters, or the curly braces around whole words) was rather tedious, and the resulting regular-expression file is less readable than the **lexc** file. The **lexc** notation is designed especially for use by lexicographers who may need to add tens of thousands of words to the source file. The basic assumption of **lexc**, that entry strings should be “exploded” apart into separate symbols, generally facilitates lexical work. However, the linguist must be careful, when moving back and forth between **lexc** and **xfst** (and, as we shall see, between **lexc** and **twolc**) to keep the different assumptions and notational conventions straight.

lexc, in exploding strings by default into separate symbols, is the exception to all the other **Xerox** finite-state tools. The assumptions of **lexc** are intended to facilitate large-scale lexicography.

4.2.4 Continuation Classes

Let’s look at another simple **lexc** grammar.

```
LEXICON Root
walk    # ;
walks   # ;
walked  # ;
walking # ;

talk    # ;
talks   # ;
talked  # ;
talking # ;

pack    # ;
packs   # ;
packed  # ;
packing # ;
```

Notice that this grammar will successfully compile into a network that accepts all the words, but linguistically it’s not very satisfying or perspicuous: some regularities are being missed. And to add a new verb lemma, such as “quack”, to the grammar, we will have to add four new strings: “quack”, “quacks”, “quacked” and “quacking”. To capture such regularities and save a lot of unnecessary typing, **lexc** allows us to break words up into parts, typically morpheme parts, to put similar morphemes together in separate LEXICONS, and to show the connections or word-formation rules via CONTINUATION CLASSES. The study of

such word formation is formally called MORPHOTACTICS or MORPHOSYNTAX. Here's an equivalent grammar written in a more perspicuous way.

```

LEXICON Root
walk      V ;
talk      V ;
pack      V ;

LEXICON V
s         # ;
ed        # ;
ing       # ;
          # ; ! <- an empty-string entry

```

Note that the common suffixes of the first grammar have been identified, “factored out” and placed in a second LEXICON named *V* (a name chosen by the linguist to suggest “verb”). Now, to add the whole paradigm for *quack* the lexicographer has to add only a single new entry under LEXICON *Root*, assigning to it the *V* continuation class.

```

LEXICON Root
walk      V ;
talk      V ;
pack      V ;
quack     V ; ! <- new entry here

LEXICON V
s         # ;
ed        # ;
ing       # ;
          # ; ! <- an empty-string entry

```

Compile this grammar in **lexc** and satisfy yourself that sixteen separate words are defined. Add “kick” and “fail” to LEXICON *Root* and confirm that eight new words have been added.

You should note the following syntactic constructions and features:

- The linguist can define as many LEXICONS as he or she finds convenient and useful. In practice, there may be dozens or hundreds of them.
- An entry can have as its continuation class the name of *any* LEXICON in the same **lexc** source file.
- The choice of lexicon names, except for *Root*, which is reserved, is entirely left up to the linguist.

- Lexicon names do not become part of the resulting network; they exist only in the **lexc** source file.
- Case is significant everywhere.
- The form of an entry, as in the last entry of LEXICON V, can be empty, representing the empty string.
- Each entry is terminated by a semicolon.
- LEXICON names are *not* terminated by a semicolon.
- Exclamation marks introduce comments that continue to the end of line. You are encouraged to use lots of them.
- As always, there should be exactly one LEXICON Root in every **lexc** source file; it corresponds to the start state of the resulting network. If you neglect to specify a LEXICON Root, **lexc** will treat the first LEXICON in the source file as the root.
- The path of continuation classes must eventually lead to a special continuation class #, indicating the end of word. Just as the LEXICON Root corresponds to the start state in the resulting network, the special # continuation class corresponds to a final state.

Continuation classes, inherited from Koskenniemi's Two-Level Morphology notation, are the basic mechanism for describing morphotactics in **lexc**. Note that the mechanism of continuation classes translates into CONCATENATION in regular expressions, and most languages do indeed build words primarily by concatenating morphemes together. However, continuation classes have come under fire, quite rightly, for being inadequate for describing morphotactic phenomena that occur in some languages, including:

1. Separated dependencies
2. Interdigitation
3. Infixation
4. Reduplication

As we shall see in good time (Chapter 9), the full arsenal of the Finite-State Calculus offers us ways to overcome the limitations of the continuation classes of **lexc**.

```
LEXICON Root
dog      N ;
cat      N ;
bird     N ;
walk     V ;
talk     V ;
pack     V ;
old      # ;
big      # ;
blue     # ;
and      # ;
or       # ;

LEXICON N
# ;      ! an empty entry for single noun
s       # ;      ! add 's' for plural nouns

LEXICON V
s       # ;      ! 's' suffix for 3rd-person sing. verbs
ed      # ;      ! 'ed' suffix for past tense
ing     # ;      ! 'ing' for the present progressive
# ;      ! an empty entry for bare verbs
```

Figure 4.8: Mixing Morphemes with Different Continuation Classes in LEXICON Root

4.2.5 Multiple Classes of Words

With appropriately defined continuation classes and LEXICONS, a single **lexc** file can accommodate multiple classes and subclasses of words. The example in Figure 4.8 mixes various kinds of roots together in the LEXICON `Root`, giving each entry an appropriate continuation class.

Another style of **lexc** programming seeks to make each LEXICON as coherent as possible, grouping together morphemes having a common continuation class. The example in Figure 4.9, equivalent to the one in Figure 4.8, shows how this can be done. Note as always that the order of LEXICONS in the source file is not significant.

If you compile the program in Figure 4.9, **lexc** outputs the following messages

```
lexc> compile-source temp-lex.txt
Opening 'temp-lex.txt'...
Root...4, Nouns...3, N...2, Verbs...3, V...4,
Adjectives...3, Conjunctions...2
Building lexicon...Minimizing...Done!
SOURCE: 23 states, 34 arcs, 23 paths.
```

showing the number of entries in each LEXICON (there may be tens of thousands in a commercial description) and showing the size of the resulting network in terms of states, arcs and paths.

4.2.6 Optionality and Loops

The **lexc** LEXICONS and continuation classes allow some slightly clumsy ways to indicate optionality of morpheme classes and loops. Consider the case of the Esperanto verb, which optionally places the aspectual morpheme *ad*, indicating repetition or continuous action, between the root and the required verbal suffix. Figure 4.10 shows one method of making *ad* optional, by including an empty entry in the same LEXICON. Satisfy yourself that the resulting network can accept both *kantis* (“sang”) and *kantadis* (“sang repeatedly”).

Another ultimately equivalent way to make *ad* optional is to create an intermediate LEXICON, such as LEXICON `V` in Figure 4.11, that leads either to LEXICON `AD` or bypasses it completely, continuing directly to the verb suffixes. The choice between the two methods is largely one of programming style.

Compile the grammars in Figure 4.10 and Figure 4.11, save the two results to file, and verify, using **test equivalent** in **xfst** (see Section 3.5.4), that they are indeed equivalent.

Continuation classes also give us a way to model iteration via loops. Let us consider the case of the fictional language Zing that optionally adds the prefix *mega* to any noun to form the augmentative (“big”) form, and this *mega* prefix can theoretically be repeated any number of times. We can model this behavior with a **lexc** loop as in Figure 4.12.

```
LEXICON Root      ! empty branches to the various
                  ! LEXICONS that can start a word
                  Nouns ;
                  Verbs ;
                  Adjectives ;
                  Conjunctions ;
                  ! add more classes here as needed

LEXICON Nouns
dog      N ;
cat      N ;
bird     N ;

LEXICON N
        # ;
s       # ;

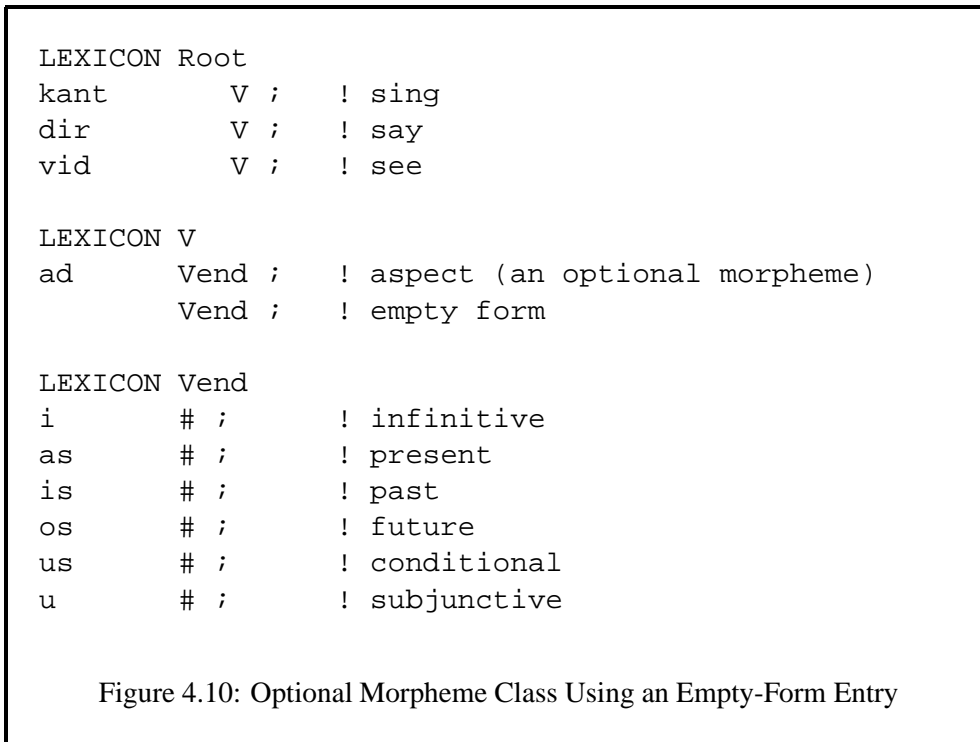
LEXICON Verbs
walk    V ;
talk    V ;
pack    V ;

LEXICON V
s       # ;
ed      # ;
ing     # ;
        # ;

LEXICON Adjectives
old     # ;
big     # ;
blue    # ;

LEXICON Conjunctions
and     # ;
or      # ;
```

Figure 4.9: A Programming Style that Groups Morphemes into Coherent Classes that Share the Same Continuation Class



Compile this grammar and verify that it will accept *warj*, *megawarj*, *megamegawarj*, etc. Note that a **lexc** loop, as seen in LEXICON MEGA, must always have an escape continuation or the word will never reach #, a final state.

4.2.7 Alternation

Many programmers with experience in traditional procedural programming languages like C have a natural but inappropriate tendency to read **lexc** programs procedurally. For example, they might try to interpret the LEXICON in Figure 4.13—and we stress that this is *not* correct—as 1) first try to match the *eg* morpheme, 2) else try to match the *et* morpheme, 3) else try to match the *ort* morpheme, else just continue on to LEXICON EVO.

It cannot be emphasized enough that this is not what happens. The entries indicated in a LEXICON simply indicate all the possible paths that a valid word might take, and in finite-state lookup and generation, all of those paths will ultimately be explored, regardless of the order in which they are declared. The order of entries in a LEXICON is not significant, and so the LEXICON in Figure 4.14 is completely equivalent to the one in Figure 4.13.

lexc source files are statements of fact about the morphotactics of a language, and they are compiled into finite-state networks, which are data structures rather than procedural algorithms. These networks are applied to input strings in the manner illustrated in Section 1.3.2.

```

LEXICON Root
kant      V ;    ! sing
dir       V ;    ! say
vid       V ;    ! see

LEXICON V
          AD ;    ! empty entry--go to AD
          Vend ;  ! empty entry--go to Vend

LEXICON AD
ad        Vend ;  ! aspect (an optional morpheme)

LEXICON Vend
i         # ;     ! infinitive
as        # ;     ! present
is        # ;     ! past
os        # ;     ! future
us        # ;     ! conditional
u         # ;     ! subjunctive

```

Figure 4.11: Optionality via Bypassing

Entries in a *lexc* LEXICON are just OR-ed alternatives. The order in which they are written is not significant.

4.2.8 Exercises

Esperanto Nouns

A Little History Esperanto was invented or *constructed* by Dr. L. L. Zamenhof, a Polish oculist, and was first described in a Russian-language publication in 1887. Another constructed language called Volapük was already enjoying some popularity, but most of its speakers deserted to join the new Esperanto movement, which showed impressive vitality up until World War I, in which an estimated 50 percent of all Esperanto speakers died. The movement has never really recovered, but it continues today among a speaking population estimated at anywhere from one to eight million speakers, the figure being hotly contested. In over a century of activity, most of the world's literary classics have been translated into Esperanto, and it

```

LEXICON Root
    MEGA ;      ! go to LEXICON MEGA first

LEXICON MEGA
mega    MEGA ;      ! this is the looping point
        Nouns ;    ! the escape path to Nouns

LEXICON Nouns
warj    # ;      ! buffalo
kanjum  # ;      ! paper clip
slarp   # ;      ! kangaroo

```

Figure 4.12: Iteration Loop in LEXICON MEGA

```

! an INCORRECT way to read a LEXICON

LEXICON Foo
eg      ARD ;      ! first try -eg; if found, go to ARD
et      # ;        ! else try -et; if found, end.
ort     AKK ;      ! else try -ort; if found, go to AKK
        EVO ;      ! else go to LEXICON EVO

```

Figure 4.13: Do NOT Try to Read **lexc** Programs Procedurally. The order of the entries within a LEXICON is not significant. A **lexc** description is compiled into a finite-state network, not a procedural program.

```

! reordering the entries in a LEXICON has no effect

LEXICON Foo
        EVO ;
eg      ARD ;
ort     AKK ;
et      # ;

```

Figure 4.14: The Order of Entries in a LEXICON is Not Significant

also boasts a considerable original literature.

While Esperanto is often charged with being too strongly based on Romance languages, it has also helped inspire Latine Sine Flexione, Interlingua and several other languages whose inventors believe strongly that Esperanto isn't Romance enough. Despite predictions that non-Indo-Europeans would never accept it, China and Japan are relative Esperanto strongholds. It has been accused of being "too inflected" although in formal linguistic terms it is not an inflecting language at all but is almost purely *agglutinating*. Language construction is definitely one of those activities where you can't please everyone.

In any case, whatever anyone thinks of it as a viable human language, Esperanto's simplicity and regularity make a perfect subject for initial exercises in **lexc**; we can model much of the language using only **lexc**, without any **twolc** or replace rules at all.

In this exercise, you will use **lexc** to model a portion of the Esperanto noun system. The facts of the noun sublanguage have been limited to keep the exercise easily manageable. If you know some Esperanto, don't worry right now about the gaps and overgeneration—some of them will be fixed in future exercises.

The goals of this exercise are 1) to reinforce the basic **lexc** syntax, and 2) to reinforce the **lexc** interface commands for compiling and saving files.

The Facts

1. Esperanto nouns usually begin with a root such as the following:

hund	("dog")
kat	("cat")
bird	("bird")
elefant	("elephant")

The English glosses are provided for information only and should not be included in the grammar, except perhaps in comments.

2. A bare root, like *hund* is not a valid word by itself.
3. All Esperanto nouns must have an *o* suffix, e.g. *hundo* and *elefanto* are valid words.
4. A word like *hundo* is masculine or perhaps better viewed as unmarked for gender. To mark it as feminine, the *in* suffix is placed between the root and the *o* suffix. E.g. *hundino*, meaning "bitch", is a valid word.
5. To mark a word as diminutive, Esperanto places an *et* suffix between the root and the *o* suffix, e.g. *elefanteto* ("little elephant") is a valid word. To mark a word as augmentative, there is a parallel *eg* suffix, e.g. *hundego* ("big dog") is a valid word.

6. The augmentative, diminutive and feminine suffixes can co-occur, and it is not clear if there are any rules to limit their mutual order or co-occurrence. For this exercise, use a **lexc** loop to allow any number of these three suffixes, in any order, between the root and the *o* suffix. Do not worry about the overgeneration for now.
7. To mark a noun as plural, add the *j* suffix, which, if present, must appear immediately after the *o* suffix. E.g. *birdoj* means “birds”. Nouns not marked overtly as plural are singular.
8. To mark a noun as accusative, the *n* suffix appears just after the *j* plural suffix (if present) or else just after the *o* suffix (which is always present). No other morpheme can occur after the accusative *n*.

The Task Based on the simplified facts listed above, write a grammar of Esperanto nouns using **lexc**. Put your grammar in a file named something like `esp-nouns-lex.txt`. We suggest that you start the grammar this way:

```
LEXICON Root
      Nouns ;
```

```
LEXICON Nouns
bird   Nmf ;
hund   Nmf ;
kat    Nmf ;
elefant Nmf ;
```

where `Nmf` is intended to mark noun roots that denote animals that can be either male or female.

After creating the source file, invoke **lexc** with the **lexc** command from the command line:

```
unix> lexc
```

At the **lexc** prompt, enter

```
lexc> compile-source esp-nouns-lex.txt
```

This will read your source file, invoke the **lexc** compiler, and put the result in the `SOURCE` register. If the **lexc** compiler finds any errors, it will print suitable messages that you should read carefully. Fix any errors in your source file using your text editor, and recompile the source file until it compiles cleanly. One solution to this exercise is shown on page 637.

Testing Test the compiled network in **lexc** using the **lookup** and **lookdown** utilities. It should successfully **lookup** the following words:

```
birdo
birdoj
birdon
birdeton
birdetinegojn
elefantegegegoj
```

Re-edit and recompile the source file until the network behaves correctly, then save the network to file as follows:

```
lexc> save-source esp-nouns-lex.fst
```

Then quit (exit) from **lexc**.

```
lexc> quit
```

To test your new **FST** file more easily, without having to type **lookup** or **lookdown** for each word, load the binary file onto an **xfst** stack and invoke **apply up** with no argument.

```
unix> xfst
xfst[0]: load stack esp-nouns-lex.fst
xfst[1]: apply up
apply up>
```

This will put **xfst** into an interactive **apply-up** mode in which you simply enter a string to be analyzed on each line, without having to retype a command like **apply up** each time. To escape from **apply-up** mode and return to the normal **xfst** command line, enter **END;**, including the semicolon.¹

```
unix> xfst
xfst[0]: load stack esp-nouns-lex.fst
xfst[1]: apply up
apply up> input_word1
apply up> input_word2
apply up> END;
xfst[1]:
```

Continue your testing. The network should accept all the following words.

¹On a Unix system, you can also escape the **apply up** mode by entering **^D**, i.e. by holding down the **Control** key and typing **D**. In Windows, the **^Z** command has the same effect.


```

hundo
hundinoj
elefantinetojn
katego
birdinego

```

You can facilitate repeated testing by typing the words to be analyzed into a separate file, e.g. `testwords.txt`, with one word to a line. Then invoke

```
xfst[1]: apply up < testwords.txt
```

to have **apply up** operate on all the strings in the file.

Negative testing can also be useful. The grammar should reject the following ill-formed strings

```

hund
hundjo
katoego
elefantonj

```

Like positive testing, negative testing can be automated by typing ill-formed strings into another file, e.g. `badwords.txt`, and entering the command `apply up < badwords.txt`.

Now go back to the **lexc** source file and add the following new noun roots, with appropriate continuation classes.

```

tigr          ("tiger")
best         ("animal")
leon         ("lion")

```

Again, the English glosses are provided for your information only and should not be included in the grammar except perhaps in comments. Recompile the grammar and test the new entries using **lookup** and **lookdown** in **lexc**, or using **apply up** and **apply down** in **xfst**.

Reduce the Task to Lexicography One main task for the computational linguist is to discover what morphotactic structures exist in the language, and then to model those structures using appropriate **lexc** LEXICONS and continuation classes. In most languages, alternation rules will also have to be written using **twolc** or replace rules. Once all the classes (and subclasses) of words have been successfully modeled, the task is then reduced to lexicography, the careful adding of new roots with appropriate continuation classes to the **lexc** source file. There may in fact be dozens of noun subclasses to distinguish—but if they are well defined and documented, the lexicographical work can often be continued by an educated native speaker who may have no background in **lexc** or computer programming in general.

Always try to reduce the task to lexicography. If you do your work well, the project can be expanded by good lexicographers.

Esperanto Adjectives

Background Esperanto adjectives are similar to nouns, but not identical. As with nouns, the task is to capture the system (or systems) of adjectives in general so that the task can eventually be reduced to good lexicography. Wherever possible and appropriate, try to design your adjective analysis so that it is parallel to the noun analysis. In a future exercise you will be asked to combine your noun and adjective analyses together into a single grammar.

The Facts The facts to be modeled have again been limited and simplified to make a manageable exercise.

1. Esperanto adjectives can begin optionally with one of the following prefixes:

ne	(negation of the root)
mal	(opposite of the root)

2. Every adjective must contain a root, and an adjective can begin with the root.

bon	("good")
long	("long")
alt	("tall/high")
grav	("important")
jun	("young")

A root like *long* is not a valid word by itself. A prefix by itself is not a valid word, and neither is a prefix plus a root.

3. All Esperanto adjectives must have an *a* suffix, e.g. *bona* and *grava*, which are valid words.
4. Like nouns, adjectives can be optionally marked as augmentative or diminutive:

eg	("augmentative")
et	("diminutive")

As with nouns, the *eg* or *et* must come after the root but before the *a* suffix. The augmentative and diminutive can co-occur, and it is not clear if there are any rules to limit their mutual order or co-occurrence. For this exercise, allow any number of these suffixes to appear inside an adjective.

5. Unlike nouns like *hundo/hundino*, adjectives cannot be marked feminine with the *in* suffix.
6. To mark an adjective as plural, add the *j* suffix, which, if present, must appear immediately after the *a* suffix. E.g. *longaj*. Adjectives not overtly marked as plural are singular.
7. To mark an adjective as accusative, the *n* suffix appears just after the *j* plural suffix (if present) or just after the *a* suffix (which is always present). No suffix can appear after the accusative *n*.

The Task First, try to visualize the adjective system graphically. Then based on the facts listed above, write a grammar of Esperanto adjectives using **lexc**. Edit the grammar in a source file named something like `esp-adjs-lex.txt`. One way to start is the following:

```
! esp-adjs-lex.txt

LEXICON Root
    Adjectives ;
    AdjPrefix ;

LEXICON AdjPrefix
mal    Adjectives ;
ne     Adjectives ;

LEXICON Adjectives
bon    Adj ;
long   Adj ;
alt    Adj ;
grav   Adj ;
jun    Adj ;
```

Complete the grammar, compile the file using **lexc**, and save the result as `esp-adjs-lex.fst`. One solution to this problem is shown on page 638.

Testing To test your new network, use **apply up** in **xfst**. The grammar should accept strings like the following:

bona
 bona j
 bona jn
 juneta j
 malboneta jn
 nealta j
 gravegeta

The grammar should reject the following strings:

mal
 bon
 bonja
 boneg
 gravaeg
 altanj

Reducing the Task to Lexicography Now go back into your source file and add the following new adjective roots:

ideal	("ideal")
luks	("luxurious")
blank	("white")
nigr	("black")
liber	("free")
evident	("obvious")
grandioz	("sublime/magnificent")
oportun	("convenient")

The English glosses are shown for your information only and should not appear in the grammar, except perhaps in comments. Recompile the grammar, save the network to file, and test the new entries using **lookup** in **lexc** or **apply up** in **xfst**.

Nouns and Adjectives

In Esperanto, as in many languages, nouns and adjectives have much in common.

1. Create a single **lexc** grammar that handles both adjectives and nouns. Typically such a grammar will begin like this.

```
LEXICON Root
      Nouns ;
      Adjectives ;
```

Cut and paste sections from your existing noun and adjective grammars to make the new comprehensive grammar. Retest it to make sure that it covers all the same phenomena as before.

2. It is now time to enhance the grammar a bit. In Esperanto, noun roots, and noun roots concatenated with augmentative, diminutive and feminine suffixes, can continue to take adjective suffixes, as in *hunda* (“canine” or “dog-like”), *hundina* (“bitch-like”), *elefantega* (“big-elephantine”), *elefantegajn* (“big-elephantine [PL, ACC]”), etc. Fix your grammar to handle such words.
3. Esperanto noun roots denoting animals with masculine and feminine sexes can take the *ge* prefix denoting “male and female”. Fix your grammar to accept words such as *gehundoj* (“male and female dogs”), *geelegantoj* (“male and female big elephants”), and *gekatetoj* (“male and female kittens”). Do not worry about overgeneration at this stage. In reality, when *ge* is present, the plural suffix *j* is perhaps required and the *in* feminine suffix is forbidden. We’ll fix this overgeneration in later exercises.
4. Adjectives can be nominalized with the *ec* suffix, which then continues on to take nominal suffixes, e.g. *boneco* (“goodness”), *alteco* (“height”), *juneco* (“youth”), *malbonegecon* (“extreme evil [ACC]”), etc.
5. Test your enhanced grammar (one possible solution is shown on page 639). It should accept

```
hundo hundino hundetoj hundeginojn gehundoj
hunda hundaj hundinajn
bona bonaj bonan boneco bonecojn
malbona malbonecoj malbonega neboneco
elefanto elefantinojn geelefantoj
elefanta elefantaj elefantajn
```

6. Your grammar should reject

```
bon elephant alt
bono geelefant hundeco gealtaj
```

In fact, our grammar is still far from complete, and some of the rejected words, especially *hundeco*, may in fact be valid words of the real Esperanto language. Your challenge as a linguist is to try to make your formal language, described in your **lexc** grammar, as much like the target language as possible, minimizing both undergeneration and overgeneration. This always requires some good old-fashioned linguistic research because you can never get the whole story from the textbooks or from your own intuition.

```

LEXICON Root
swim          # ;
swim:swam    # ;      ! <- an upper:lower entry

```

Figure 4.15: The *upper:lower* Notation of **lexc**

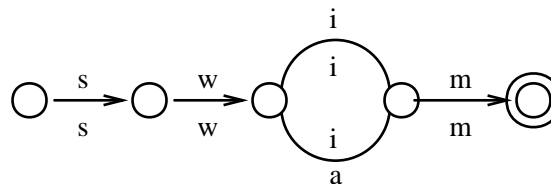
Linguistic tools such as **xfst** and **lexc** give you powerful ways to encode linguistic facts, but they cannot tell you what the facts are or otherwise do the linguistics for you.

4.3 Defining Lexical Transducers

4.3.1 *Upper:Lower* Entries

So far we have used **lexc** to make simple one-level automata, but **lexc** has been enhanced to allow the direct specification of useful transducers as well. The most straightforward mechanism is the *upper:lower* form in lexical entries.

In Figure 4.15, the *swim:swam* entry specifies a transducer that has [s w i m] on the upper-side and [s w a m] on the lower-side. As with simple entries, the upper and lower strings are exploded into separate symbols by default, and the continuation classes work the same as before. In Figure 4.15, the simpler entry *swim* is also interpreted in two levels, just as if it had been written *swim:swim*. The network compiled from this grammar is shown in Figure 4.16.

Figure 4.16: The Transducer for *swim:swam* and *swim:swim*

Compile the grammar and confirm that when you **lookup** “swam” or “swim” the response is “swim”. When you **lookdown** (generate) from “swim” the response should consist of two strings, “swim” and “swam”, both lower-side strings

that relate to the input on the upper side.

In **lexc** the colon (:) is a special character, and the *upper:lower* notation can have a string of characters on each side of the colon, as in *swim:swam*. The strings “swim” and “swam” are exploded into individual characters according to the normal **lexc** assumptions.

4.3.2 Epsilons

Add some more two-level paths to the lexicon, including *think:thought*, *win:won*, *go:went*. Note that when the lengths of the strings differ, **lexc** automatically pads out the shorter one with zeros (epsilons) on the end as in Figure 4.17.

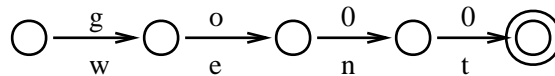


Figure 4.17: The Transducer for *go:went*

Note that the two strings in an *upper:lower* entry are always lined up symbol-by-symbol, starting at the left side of both strings. In the *go:went* example, upper-side *g* pairs with lower-side *w*, upper-side *o* pairs with lower-side *e*, and the remaining two lower-side symbols are paired with filler epsilons (shown here as zeros) on the upper side.

These zeros (epsilons) are not real characters but rather represent the empty string. When matching input against the arcs of a network, a zero arc is a free match and free transition that consumes no input characters. When strings are output from a network, a zero arc outputs nothing (the empty string).

In **lexc** *upper:lower* entries you can control where the zeros occur by typing them explicitly. The entry *fight:fought* would normally be lined up as

```

upper:  f i g h t 0
lower:  f o u g h t
  
```

with a zero filling out the upper string on the end, but you may want to minimize the number of mixed character pairs by padding out the shorter string with a zero

in the middle, either `fi0ght:fought`, which would line up

```
upper:  f i 0 g h t
lower:  f o u g h t
```

or `f0ight:fought`, which would line up as

```
upper:  f 0 i g h t
lower:  f o u g h t
```

The practical results will be the same in all three lineups.²

4.3.3 Regular-Expression Entries

In addition to simple `string` and `upper:lower` entries, there is a third and final type of **lexc** entry that allows you to use the power of regular expressions inside **lexc**.

```
LEXICON Root
go          # ;          ! simple entry (same as go:go)
go:went    # ;          ! upper:lower entry
< d o:i 0:d > # ;      ! regular-expression entry
                !          (same as do:did)
< a b c* (d) e+ > # ; ! another regular-expression
                !          entry
```

In **lexc**, regular-expression forms are written inside angle brackets (`<>`), using the notation and assumptions of regular expressions. These facts deserve some repetition and expansion because they account for much confusion among **lexc** users.

- Inside **lexc** angle brackets, most of the notation of regular expressions is available, including Kleene stars (*), Kleene pluses (+), optionality (shown with parentheses), union (shown with vertical bars), complex iteration, complementation, intersection, etc.³ Outside the angle brackets, **lexc** does not offer such a powerful formalism.
- Along with the power of regular expressions comes the corollary of *full assumptions* of regular expressions. Asterisks and plus signs, circumflexes and commas, vertical bars and other characters that are normally *not* special characters in **lexc** do become special when used inside angle brackets. All notations inside angle brackets are compiled like the regular expressions described in the **xfst** chapter.

²The manual specification of the position of the zero, which minimized the number of mixed-character labels in the resulting network, used to be a Good Thing. However, it is now seldom necessary to worry about the number of labels.

³Inside **lexc**, it is not possible to define and reuse **xfst** variables.

- Recall that the default assumption for compiling **xfst** regular expressions is that any string of characters written together, without spaces between the characters, is interpreted as a single multicharacter symbol. Thus, the regular expression `cat` denotes a language consisting of a single string “`cat`” that consists of a single symbol spelled **cat**. In complete contrast, the default convention of simple *string* and *upper:lower lexc* entries is that strings of characters written together are to be exploded apart into separate symbols. Thus the **lexc** grammar

```
LEXICON Root
cat      # ;
```

denotes a language consisting of a single string, “`cat`”, composed of three separate concatenated symbols spelled **c**, **a** and **t**. However, inside angle brackets inside a **lexc** program, *all* the conventions of regular expressions apply, including the one that interprets and compiles strings of characters written together as a single symbol (with a multicharacter print name).

```
LEXICON Root
cat      # ;      ! cat will be exploded into
                !      three separate symbols, as
                !      per normal lexc assumptions
< dog >  # ;      ! inside angle brackets, dog
                !      is not exploded but is
                !      compiled, as in all regular
                !      expressions, as a single
                !      symbol
< c a t > # ;      ! denotes the string "cat" as three
                !      concatenated symbols; here the
                !      explosion was done manually
                !      by the linguist
```

- Because angle brackets are used inside **lexc** to delimit regular expressions, angle brackets are special characters for **lexc** in general. To unspecialize them (i.e. to denote the literal angle bracket characters), precede them with percent signs: `%<` and `%>`.

The special characters of **lexc** are the semicolon (;), the pound sign (#), the colon (:), and the angle brackets (< and >). Inside the angle brackets, the compiler reverts to all the assumptions of **xfst** regular expressions, where almost all the punctuation marks are special.

There are good practical reasons for **lexc** to explode strings by default, except, of course, when they appear inside angle brackets. There are also good reasons for the very different conventions used when compiling regular expressions. It is up to the linguist to keep these differing conventions straight when working with **lexc** and other tools, and when working inside and outside of angle brackets in a **lexc** file. Failure to understand the differing conventions is the source of many compilation errors and runtime bugs.

4.3.4 Multicharacter Symbols

We have stressed that **lexc** explodes strings into separate symbols *by default*, but what do you do when you want to override that default and use multicharacter symbols such as +Noun, +Verb, +1P, and +Sg in a **lexc** grammar? The answer is that you have to declare them explicitly as `Multichar_Symbols` at the beginning of the **lexc** source file. Figure 4.18 is a partial grammar of some Latin verbs (*-are* class) in the present-indicative active conjugation.

In **lexc**, multicharacter symbols must be explicitly declared in the `Multichar_Symbols` section at the top of the **lexc** source file.

This grammar defines a relation that contains pairs of strings that look like the following. As usual, we follow the **Xerox** convention of showing the upper-side (or lexical) string above the lower-side (or surface) string.

```
Lexical:      canto+Verb+PresInd+Act+1P+Pl
Surface:      cantamus
```

```
Lexical:      canto+Verb+PresInd+Act+2P+Sg
Surface:      cantas
```

The grammar is written so that the lexical strings start with a form like *canto* because this form (“I sing”) is used by convention as the headword in printed Latin dictionaries. Defined multicharacter symbols such as +PresInd and +Verb are manipulated in **lexc**, and by any subsequent rules, like any other single symbols. Linguists can define any multicharacter symbols that they find useful, and they can spell them any way they want; however, by a useful **Xerox** convention, the symbol tags intended to convey morphological or syntactic features back to the user include a plus sign or other non-alphabet character or characters. Note that in **lexc** the plus sign is not special (except inside angle brackets) and needs no literalizing percent sign in the example in Figure 4.18.

```

Multichar_Symbols +Verb +PresInd +Act
                  +1P +2P +3P +Sg +Pl

LEXICON Root
      AreVerbs ;

LEXICON AreVerbs
cant      AreV ;      ! to sing
laud      AreV ;      ! to praise
am        AreV ;      ! to love

LEXICON AreV
o+Verb:0  AreConj ;   ! N.B. two symbols,
                  ! o and +Verb on the
                  ! lexical side

LEXICON AreConj
      ArePresIndicAct ;
!      ArePresIndicPass ;      ! add later

LEXICON ArePresIndicAct
+PresInd+Act:0      ArePresIndicEnds ;

LEXICON ArePresIndicEnds
+1P+Sg:o      # ;
+2P+Sg:as     # ;
+3P+Sg:at     # ;

+1P+Pl:amus   # ;
+2P+Pl:atis   # ;
+3P+Pl:ant    # ;

```

Figure 4.18: A Small Fragment of Latin Verb Conjugation

The **lexc** default is to explode written strings into separate, concatenated symbols. You override the **lexc** explosion assumption by declaring `Multichar_Symbols`. In contrast, the assumption in **xfst** regular expressions is that strings of contiguous characters are interpreted as single symbols. When separate symbols are desired in **xfst** regular expressions, and inside **lexc** angle-bracket entries, the grammar writer must separate the symbols manually, e.g. [c a t], or surround them with curly braces as in {cat}.

4.3.5 Exercises

Esperanto Nouns and Adjectives with Upper-Side Tags

Rewrite your Esperanto noun-and-adjective grammar so that the lexical strings (i.e. the upper-side strings) consist of roots and multicharacter-symbol tags. Follow the general plan shown in the treatment of Latin *-are* verbs in Figure 4.18.

1. Each noun root should be marked on the upper side with the `+Noun` tag; each adjective root should be marked with the `+Adj` tag. Declare these and all other tags as `Multichar_Symbols`.
2. In addition, use `+NSuff` to mark the *o* suffix, `+ASuf` to mark the *a* suffix, and `+Nize` to mark the nominalizing *ec* suffix.
3. Declare and use the following tags as well:

<code>+Pl</code>	for plural (j)
<code>+Sg</code>	for singular (i.e. not plural)
<code>+Acc</code>	for accusative (n)
<code>MF+</code>	for male-female (ge)
<code>+Aug</code>	for augmentative (eg)
<code>+Dim</code>	for diminutive (et)
<code>+Fem</code>	for feminine (in)
<code>Op+</code>	for opposite (mal)
<code>Neg+</code>	for negative (ne)

The tags with plus-signs after them are used for marking prefixes.

4. Your resulting network should accept the following string pairs; i.e. if you **lookup** the lower string, you should get back the upper string, and if you **lookdown** the upper string, you should get back the lower string. Test carefully, fixing your grammar as necessary to handle the strings as shown here:

Lexical:	hund+Noun+NSuff+Sg
Surface:	hundo
Lexical:	hund+Noun+ASuff+Pl
Surface:	hundaĵ
Lexical:	MF+hund+Noun+NSuff+Pl+Acc
Surface:	gehundojn
Lexical:	elefant+Noun+Aug+Fem+NSuff+Pl
Surface:	elefanteginoj
Lexical:	Op+bon+Adj+Nize+NSuff+Sg
Surface:	malboneco

A solution to this exercise is shown on page 640.

Esperanto Verbs

It's now time to add Esperanto verbs to your grammar. As usual, the facts have been limited for this exercise. Your root lexicon should look something like

```
LEXICON Root
  Nouns ;
  Adjectives ;
  Verbs ;
```

As usual, the ordering of entries in a LEXICON and the ordering of LEXICONs in a **lexc** source file are not significant.

The Facts

1. Esperanto verbs can start with a verb root or with one of the following prefixes. The glosses shown below are for your information only and should not be included inside your grammar.

```
mal          ("opposite")
ne           ("negation")
```

2. Every verb requires a verb root, but the roots are not valid words by themselves.

```
don          ("give")
est          ("be")
```

```
pens          ("think")
dir           ("say")
fal          ("fall")
```

3. Directly after the verb root, there can appear an optional aspect marker *ad* that denotes continuous or repeated action.

```
ad           ("continuous/repeated")
```

4. After the root (or after the root plus *ad*) the verb must have exactly one of the following tense/mood suffixes:

```
i           ("infinitive")
as          ("present")
is          ("past")
os          ("future")
us          ("conditional")
u           ("subjunctive")
```

The Task

- Visualize the verbal system graphically, and plan out an analysis with the goal of relating pairs of lexical/surface strings like the following:

```
Lexical:      don+Verb+Cont+Past
Surface:      donadis
```

- Declare and use the tags +Verb, +Past, +Pres, +Fut, +Subj, +Inf, +Cond (defined at the top of the **lexc** file in the `Multichar.Symbols` statement). Use the +Cont tag for the *ad* suffix.
- Test by looking up the following examples.

```
donis      donos      donadus
fali       falu       faladi
pensas     pensadas
```

A solution to this exercise is shown on page 641.

- Now add the following new verb roots to the lexicon.

```

ir           ( "go" )
ven          ( "come" )
vetur       ( "travel" )
parol       ( "speak" )
far         ( "do" )
raz         ( "shave" )
kant        ( "sing" )

```

- Add the following new noun roots to the lexicon. Nouns that do not refer to male/female animals should not allow the *in* suffix; create appropriate new continuation classes as necessary.

```

dom          ( "house" )
mond        ( "world" )
infan       ( "child" )
tag         ( "day" )
monat       ( "month" )
jar         ( "year" )
odor        ( "odor" / "smell" )
urb         ( "city" )
urs         ( "bear" )

```

- Add the following adjective roots.

```

verd        ( "green" )
plat        ( "flat" )
san         ( "healthy" )

```

- Save your noun-adjective-verb description in `esperanto-lex.txt`, compile it with **lexc**, and test it as thoroughly as you can.

Esperanto Verbalization

Consult any grammar of Esperanto and research the behavior of the suffixes *-ig* and *-iĝ*, which verbalize nouns and adjectives. Augment your Esperanto grammar to handle these verbalizing suffixes.

Bambona

Redo the Bambona exercise on page 155, this time using **lexc** to build the lexicon, but using replace rules to perform the vowel changes as before. You should first compile the **lexc** file and save the resulting network (using **save-source**) to a file, say `bambona-lex.fst`. Then exit **lexc** and enter **xfst**. In **xfst**, first compile the rules, and then compose the rule network underneath the lexicon network (see Section 3.5.4, page 163).

Simplified Germanic Compounding

Assume that we are dealing with a Simplified Germanic language that looks a lot like English. For this example we will deal only with simple adjectives, simple nouns, and compound words containing adjective and noun roots. Our Simplified Germanic example will exhibit free compounding: A compound word consists of any two or more adjective or noun roots, concatenated together, and the overall part of speech of the compound word is determined by the final compound element. The final compound element has the normal adjective or noun inflections as appropriate, and non-final compound elements should consist only of bare noun and adjective roots.

Warning: this exercise will overgenerate wildly, accepting and producing compound words that make little or no sense. The language will in fact cover an infinite number of words. Because it is impossible to predict which compound words people will make up, such overgeneration is practically unavoidable when modeling free-compounding languages.

Here are the facts:

1. Assume we have the following noun roots:

```
LEXICON Nouns
dog      N ;
horse   N ;
cow      N ;
food     N ;
bicycle N ;
tree     N ;
sky      N ;
house    N ;
```

2. Nouns can be singular or plural, with the plural expressed as *s* and singular expressed as nothing. Write LEXICON N so that you get string-to-string mappings like the following (zeros are used to mark epsilons)

```
Lexical:   horse+Noun+Sg      horse+Noun+Pl
Surface:   horse0      0      horse0      s
```

with the lexical- and surface-level symbols lined up as shown.

3. Adjective roots include the following:


```

LEXICON Adjectives
small  Adj ;
red    Adj ;
big    Adj ;
black  Adj ;
blue   Adj ;
tall   Adj ;
brown  Adj ;

```

4. Adjectives can be plain or comparative (+**Comp:er**) or superlative (+**Sup:est**). Write the continuation lexicons to support the following examples, which show the zeros overtly (don't worry about spellings like "biger" and "skys"—they are OK in Simplified Germanic).

```

Lexical:    small+Adj
Surface:    small0

```

```

Lexical:    small+Adj+Comp
Surface:    smalle  r

```

```

Lexical:    small+Adj+Sup0
Surface:    smalle  s  t

```

5. Now add productive compounding, by allowing all bare noun and adjective roots to loop back to pick up more noun and/or adjective roots. Mark non-final noun elements with +NC (noun-compound element) and non-final adjective elements as +AC. Also, place [^]CB (compound boundary) as a feature between compound elements. As usual, you should declare [^]CB, +AC, +NC and other tags in the `Multichar.Symbols` section. The final element in a compound, whether adjective or noun, should receive the normal adjective or noun tags as appropriate. Strive to handle the following string pairs (zeros are not shown here, and there is no attempt to align the lexical and surface characters).

```

Lexical:    dog+NC^CBhouse+Noun+Sg
Surface:    doghouse

```

```

Lexical:    dog+NC^CBhouse+Noun+Pl
Surface:    doghouses

```

```

Lexical:    small+AC^CBhorse+Noun+Sg
Surface:    smallhorse

```

Lexical:	blue+AC^CBsky+Noun+Sg
Surface:	bluesky
Lexical:	sky+NC^CBblue+Adj
Surface:	skyblue
Lexical:	dog+NC^CBfood+NC^CBhouse+Noun+Sg
Surface:	dogfoodhouse
Lexical:	dog+NC^CBhouse+NC^CBfood+Noun+Pl
Surface:	doghousefoods
Lexical:	sky+NC^CBblack+Adj+Comp
Surface:	skyblacker

6. It's fairly easy to imagine scenarios in which the compound words just cited might be useful and would make sense. But it is almost impossible to inject a notion of what makes sense into a system that handles productive compounding. Your system should also analyze the following unlikely compounds.

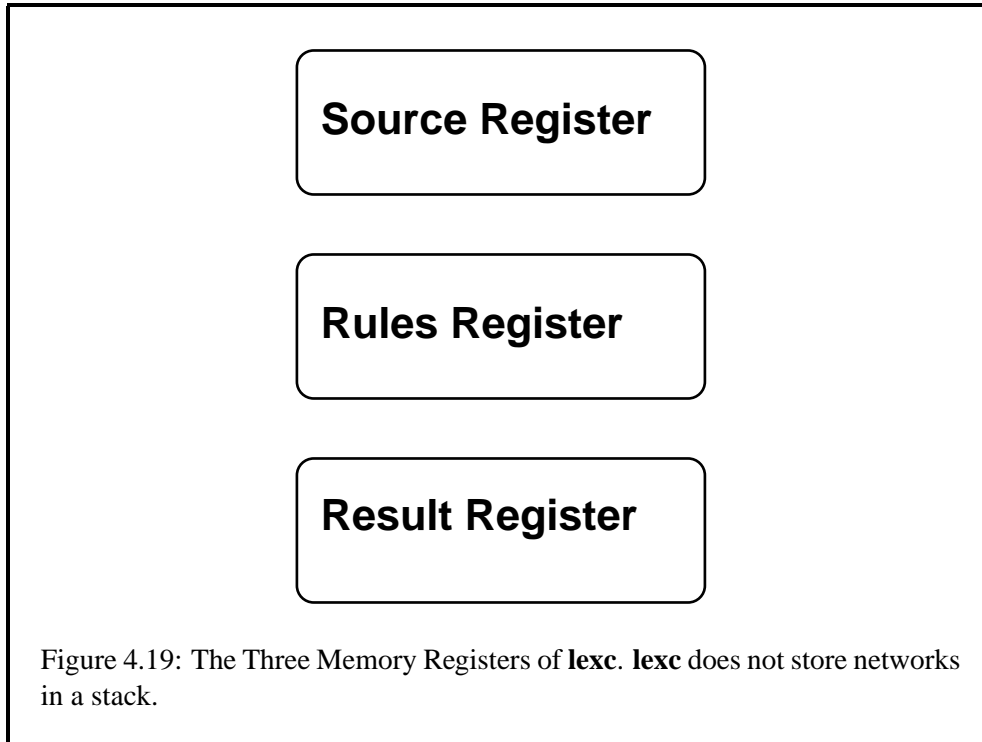
dogbicycle
 cowsky
 bluecows
 skybluedogfoodhouses
 dogsmallest
 horsehorsehorsehorse
 redsmallbigblack

The price of handling productive compounding is always overgeneration and overrecognition to some extent; any attempt to prevent such meaningless compounds will also prevent the analysis of real compounds that real speakers produce spontaneously every day. Note, however, that the system should not accept just anything; it should, according to the rules cited above, refuse to analyze words like the following:

dogshouse
 smallerhouses
 dogssmallest
 biggesthouse

7. Your compounding system should still handle non-compound words like

houses



```
dog
horses
small
smaller
smallest
```

4.4 The lexc Interface

4.4.1 Introduction

Historically, **lexc** and **twolc** were sometimes licensed without an interface like **xfst** that gives access to all the algorithms of the Finite-State Calculus. The **lexc** interface was therefore enriched to provide a number of the most essential algorithms for manipulating finite-state networks and combining multiple networks into Lexical Transducers. Unfortunately, these algorithms sometimes hide under idiosyncratic names.

4.4.2 lexc Registers

The **lexc** memory structure is composed of just three registers: the SOURCE REGISTER, the RULES REGISTER, and the RESULT REGISTER. They are best visualized as shown in Figure 4.19. It is important to keep this memory structure in

mind as you work with the **lexc** interface; many of the **lexc** commands have names that refer to the registers, and all the operations take their data from, or write their results to, one or more of the registers.

lexc organizes its working memory into three named registers: SOURCE, RULES, and RESULT. In contrast, the **xfst** interface organizes its memory as a last-in first-out stack.

The SOURCE register and the RESULT register can hold only a single network at a time. The RULES register may contain one network or a set of networks, typically resulting from compiling a set of **twolc** rules (see Section 5.2.4).

4.4.3 Command Menu

Let's look again at the commands available in **lexc**:

```

Input/Output -----
  Source:      compile-source, merge-source, read-source,
               result-to-source, save-source.
  Rules:      read-rules.
  Result:     merge-result, read-result, save-result,
               source-to-result.
  Properties: add-props, reset-props, save-props.
Operations -----
  Composition: compose-result, extract-surface-forms,
               invert-source, invert-result.
  Checking:   check-all, lookdown, lookup, random,
               random-lex, random-surf.
  Switches:  ambiguities, duplicates, failures,
               obey-flags, print-space, quit-on-fail,
               show-flags, singles, time.
  Scripts:   begin-script, end-script, run-script.
Display -----
  Misc:      banner, labels, props, status, storage.
  Help:     completion, help, history, ?.
Type 'quit' to exit.

```

This menu is displayed when you first invoke **lexc**, and you can cause it to be redisplayed at any time by entering **help** or the question mark (?) at the command line. You can also enter **help** followed by the name of any command to see a short summary of its operation. You can also enter the command **status** to see which registers are filled and to see the current setting of the various switches that affect how **lexc** operates.

4.4.4 Individual lexc Commands

Keep the three **lexc** registers in mind as we review the various utilities. In each case, the online help message is shown, followed in some cases by additional comments.

Input-Output Commands

The Source Register

compile-source 'compile-source *textfile*' reads the lexc source file contained in *textfile*, compiles it, and stores the resulting network in the SOURCE register. You can save this network to file with the command 'save-source'. To read a pre-compiled network from file into the SOURCE register, use the command 'read-source'. To compose the SOURCE network with a set of rule transducers, previously read in by 'read-rules', use the command 'compose-result'.

merge-source 'merge-source *fsmfile*' load the pre-compiled network in *fsmfile* and merges (unions) it with the network currently stored in the SOURCE register (if any), storing the result in the SOURCE register. This allows the SOURCE network to be constructed as a union of any number of saved networks.

read-source 'read-source *fsmfile*' loads the pre-compiled network in *fsmfile* and stores it in the SOURCE register. Any previous SOURCE network is overwritten.

result-to-source 'result-to-source' moves the network in the RESULT register to the SOURCE register, clearing the RESULT register and overwriting any previous SOURCE network.

save-source 'save-source *filename*' stores the current SOURCE network to file in a binary format. Use 'read-source' or 'read-result' to read in files created in this way.

The Rules Register

read-rules 'read-rules *fsmfile*' loads the transducer network(s) in *fsmfile* and stores them in the RULES register, overwriting any previous RULES network(s). The files read by 'read-rules' should be binary files saved from *xfst* or *twolc*.

The Result Register

merge-result 'merge-result *fsmfile*' load the pre-compiled network in *fsmfile* and merges (unions) it with the network currently stored in the RESULT register (if any), storing the result in the RESULT register. This allows the RESULT network to be constructed as a union of any number of saved networks.

read-result 'read-result *fsmfile*' loads the pre-compiled network in *fsmfile* and stores it in the RESULT register. Any previous RESULT network is overwritten.

save-result 'save-result *filename*' stores the current RESULT network to file in a binary format. Use 'read-result' or 'read-source' to read in files created in this way.

source-to-result 'source-to-result' copies the network in the SOURCE register to the RESULT register, overwriting any previous RESULT network.

Properties

add-props 'add-props' reads a set of attribute-value pairs from a text file and appends them to the property list of the SOURCE or the RESULT network.

reset-props 'reset-props' reads a set of attribute-value pairs from a text file and resets the property list of the SOURCE or the RESULT network, overwriting any previously stored properties.

save-props 'save-props' write the property list of the SOURCE or the RESULT network to a text file in the format expected by 'add-props' and 'reset-props'.

Operations among Registers

Composition

compose-result 'compose-result' composes the RULES network(s) under the SOURCE network, storing the result in the RESULT register. This command requires that the SOURCE and RULES registers contain networks. 'compose-result' uses the intersecting-composition algorithm. The RULES register may contain a single FST or a set of transducers produced by compiling a set of twolc rules and saving them without intersecting; in such cases, the intersecting-composition algorithm performs the necessary intersection and composition. Any Flag Diacritics on the lower-side of the SOURCE network are treated as epsilons in the composition and are preserved on the lower-side of the RESULT network. The RULES may not contain Flag Diacritics.

extract-surface-forms 'extract-surface-forms' replaces the RESULT network with a simple FSM that contains only the surface forms from the original RESULT network.

invert-source 'invert-source' inverts the two sides of the SOURCE network. If the SOURCE network is a transducer, the upper side becomes the lower side and vice versa. If the network is not a transducer, the invert command has no effect.

invert-result 'invert-result' inverts the two sides of the RESULT network. If the RESULT network is a transducer, the upper side becomes the lower side and vice versa. If the network is not a transducer, the invert command has no effect.

Checking

check-all 'check-all' helps you verify the RESULT network produced by using 'compose-result' to compose the SOURCE network with the RULES network(s). 'check-all' compares the upper side of the SOURCE network to the RESULT network and tabulates and displays the number of upper-side SOURCE strings that have one, none or several realizations in the RESULT net. The enumeration of examples by 'check-all' is controlled by a set of switches: If the 'singles' switch is ON, all upper-side SOURCE strings with one output in the RESULT are printed alongside the result. If the 'failures' switch is ON, every upper-side SOURCE string that is no longer in the RESULT is printed, followed by a row of asterisks. If the 'duplicates' switch is ON, each upper-side SOURCE string with multiple outputs in the RESULT is printed with all of its output forms. If the 'ambiguities' switch is ON, each lower-side RESULT string that corresponds to two or more lexical strings is printed out with all of the related lexical strings. The default settings are: 'failures' = ON, 'duplicates' = ON, 'singles' = OFF, and 'ambiguities' = OFF. Use the 'status' command to see the settings of the switches.

lookdown 'lookdown *string*' searches the SOURCE network or the RESULT network for *string* by matching *string* against the upper side of the transducer, and printing out all related strings on the lower side of the transducer.

lookup 'lookup *string*' searches the SOURCE network or the RESULT network for *string* by matching *string* against the lower side of the transducer, and printing out all related strings on the upper side of the transducer.

random 'random' prints 15 random paths (words) from either the SOURCE network or the RESULT network, depending on the user's choice. Each path is printed as a sequence of symbol pairs.

random-lex 'random-lex' prints 15 random words from the upper-side language of either the SOURCE network or the RESULT network, depending on the user's choice. Only the lexical or upper side words are printed.

random-surf 'random-surf' prints 15 random words from the lower-side language of either the SOURCE network or the RESULT network, depending on the user's choice. Only the surface or lower side words are printed.

Switches

ambiguities 'ambiguities' is a switch that affects the output of the 'check-all' command. If the 'ambiguities' switch is ON, 'check-all' prints all the surface forms in RESULT that relate to two or more lexical forms. For example, it can detect that the French word 'suis' belongs to two paradigms: the copula 'tre' and the verb 'suivre'. If the 'ambiguities' switch is OFF, 'check-all' does not print such forms. The default setting is OFF.

duplicates 'duplicates' is a switch that affects the output of the 'check-all' command. If the 'duplicates' switch is ON, 'check-all' prints all lexical forms in the RESULT network with multiple surface outputs. If the 'duplicates' switch is OFF, 'check-all' does not print such forms. The default setting is ON.

failures 'failures' is a switch that affects the output of the 'check-all' command. If the 'failures' switch is ON, 'check-all' prints all lexical forms from the SOURCE network that yield no output in the RESULT network; each lexical form is paired with a string of asterisks to indicate the failure. If the 'failures' switch is OFF, 'check-all' does not print such forms. The effect of 'failures'=ON is to display all the forms that were lost in the process of composing the RULES network(s). This is usually helpful, and the default setting is ON.

obey-flags 'obey-flags' is an ON/OFF or toggled switch. If it is ON, Flag-Diacritic constraints are enforced and Flag Diacritics are not displayed. If it is OFF, Flag-Diacritic constraints are not enforced and Flag Diacritics are visible; i.e. Flag Diacritics are treated as normal multicharacter symbols. The default setting is ON.

print-space 'print-space' is an ON/OFF switch. If it is ON, 'random', 'random-lex', and 'random-surf' print a space between output symbols. Default is OFF.

quit-on-fail 'quit-on-fail' is an ON/OFF or toggled switch. If it is ON, lexc quits with an error message if a command is incorrectly spelled or cannot be executed because of a missing file or some other problem. This is useful when lexc is run by a shell script. The default setting is OFF.

show-flags 'show-flags' is an ON/OFF switch. If it is ON, 'random', 'random-lex', and 'random-surf' display flag diacritics. Default is OFF.

singles 'singles' is an ON/OFF or toggled switch that affects the output of the 'check-all' command. If the 'singles' switch is ON, 'check-all' prints all upper-side strings from the SOURCE network that yield exactly one surface output in the RESULT network. If the 'singles' switch is OFF, 'check-all' does not print such forms. Usually the listing of such 'singles' solutions is overwhelming and not interesting, so the default setting is OFF.

time 'time' is an ON/OFF or toggled switch that provides timing information for some commands. The default setting is OFF.

Scripts

begin-script 'begin-script *file*' records all subsequently typed lexc commands in *file* until 'end-script'. Such a script of commands can then be run using 'run-script'.

end-script 'end-script' stops recording lexc commands into a script file. Begin recording a new script with 'begin-script'; run an existing script with 'run-script'.

run-script 'run-script *textfile*' executes the script of lexc commands previously stored in *textfile*. Record a new script with 'begin-script' and 'end-script'. Scripts can also be created in a text editor.

Display

Miscellaneous

banner 'banner' displays information about the version and the authors of the Xerox Finite-State Lexicon Compiler.

labels 'labels' displays the label set of the SOURCE, RESULT, or RULES network. The lower side SOURCE network labels should match the upper side of the RULES network labels. If a symbol on the lower side of some source label does not appear on the upper side of at least one rule label, every string that contains that source label will disappear in the composition.

props 'props' displays the property list of the SOURCE or the RESULT network.

status 'status' displays information about the current SOURCE, RULES, and RESULT. It also displays the current setting of the switches.

storage 'storage' displays statistics about current memory usage.

Help

completion 'completion' is toggled switch that turns OFF/ON the automatic command completion feature and activates (or deactivates) the history mechanism. The default setting is OFF.

help 'help *command*' displays short documentation on what *command* does. 'help all' prints out all help messages. For more detailed documentation, consult The Book.

history 'history' displays the history list.

? '?' displays a menu of available **lexc** commands.

Quit/Exit

quit 'quit' exits **lexc**.

4.5 Useful **lexc** Strategies

4.5.1 Compiling **lexc** Files Directly from **xfst**

Many developers use the **lexc** interface only for compiling a **lexc** source file and saving the result to file; then they close the **lexc** interface, open **xfst**, read the compiled result from file, and continue working within **xfst**. For such users, it is now possible to compile **lexc** files directly from within **xfst** and avoid using the **lexc** interface altogether. The **xfst** command is **read lexc**, followed by a left angle-bracket and the name of the source file, e.g. `my-lex.txt`.

```
xfst[0]: read lexc < my-lex.txt
xfst[1]:
```

If the compilation is successful, the new network will be pushed onto The Stack. The **read lexc** command is parallel to **read regex**.

4.5.2 The Art and Craft of **lexc**

As with any programming language, there is an art (or at least a craft) to writing good **lexc** code. First and foremost, there is no substitute for starting with a clear plan, a coherent linguistic model to be formalized; you can do good linguistics without computers, but no amount of computerizing will make bad linguistics into good linguistics (or bad linguists into good linguists). Second, there are some useful conventions, techniques and idioms for using **lexc** to best advantage; and while these tricks of the trade are not always obvious to the beginner, they can be illustrated and taught. Third, there are some notorious **lexc** traps waiting to catch the unwary. This section is dedicated to transmitting some of the accumulated wisdom of experienced **lexc** programmers.

4.5.3 Expert Morphotactics

Defining Sublexicons

All Morphemes are Organized into LEXICONS The various morphemes (prefixes, roots, suffixes, and perhaps patterns, infixes, tonality markers and other featural archiphonemes) that are used to build words in your language all have to be organized into LEXICONS in a **lexc** program. There must be at least one LEXICON, the obligatory LEXICON `Root`, and in practice the linguist may need to

```
LEXICON Root
  Nouns ;
  Verbs ;
  Adjectives ;
  Prepositions ;
  Pronouns ;
  FunctionWords ;
  Numbers ;
  Dates ;
  Punctuation ;
```

Figure 4.20: A Typical LEXICON Root for a Natural Language

define dozens or even hundreds of others. What are the criteria for grouping and dividing morphemes? While there is no single answer, and while even two good linguists will never do things in quite the same way, there are some general principles to follow.

A LEXICON Should be Coherent Coherence is an ideal and a challenge to the linguist, always threatened by the idiosyncrasies of real language. The morphemes grouped into a LEXICON should generally be linguistically coherent; that is, they should share the same (or at least a very similar) morphotactic distribution. Thus a linguist usually starts by defining a LEXICON for verb roots, another for noun roots, and third for adjective roots, and perhaps a dozen more to handle major categories. A typical **lexc** program starts with a LEXICON Root as shown in Figure 4.20.

Major classes may be further divided according to the taste or needs of the linguist. You may, for example, find it useful to subdivide the nouns into several distinct subclasses, or the verbs into dozens of verb subclasses, especially if you are modifying existing lexicons or morphological analyzers that already make such distinctions. You can always group the subclasses under an umbrella LEXICON, if convenient, as shown in Figure 4.21.

All else being equal, it is usually better to preserve distinctions at the lower levels of **lexc** than to collapse them. It's always easy to collapse distinctions later; but distinctions erased at the lower levels are difficult to restore.

Other valid LEXICON groupings generally conform to the notion of affix classes such as noun suffixes, adjective prefixes, adjective suffixes, verb suffixes, etc.

```

LEXICON Nouns
  CommonNouns ;
  SingularOnlyNouns ;
  PluralOnlyNouns ;

```

Figure 4.21: An Umbrella LEXICON for Three Noun Subclasses

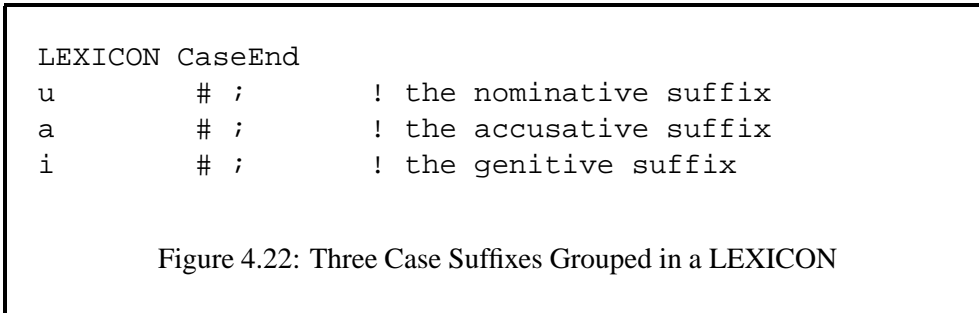
A LEXICON is a Potential Continuation Target Beginners in **lexc** sometimes assume that LEXICON coherence means that all the entries in a LEXICON could (or should) have the same continuation class, that all the entries should point to the same target. This is *not* necessarily the case. While the entries in a LEXICON should usually appear in similar morphotactic frames, it is important to understand that

- Each entry in a LEXICON has its own continuation class, so a LEXICON containing five entries could conceivably branch and continue in five different directions.
- Morphemes (and this is a key notion) are properly grouped under a single LEXICON because they form a coherent *target* for continuations from other morphemes. In other words, think of a LEXICON as something coherent to branch *to*, not necessarily to branch *from*.

That being said, a significant school of **lexc** users prefer to build maximally coherent LEXICONS, with each entry sharing the same continuation class. **lexc** certainly does not force this programming style, but it may facilitate maintenance and visualization of the morphotactic description.

Refining LEXICON Targets A good linguist puts some serious thought into planning an analysis before trying to formalize it in **lexc**; but nobody is perfect, and the rigorous coding of **lexc** will often open your eyes to regularities or subtle distinctions that you missed before. For example, it is often the case that the linguist will start an analysis by grouping a fairly obvious subclass of morphemes under a single sublexicon. Imagine a language that has three case suffixes for nouns as in Figure 4.22.

Thousands of noun roots such as *kirit* may share `CaseEnd` as a continuation class, and the grammar would recognize strings such as *kiritu*, *kiriti* and *kirita*. However, you may later find a small subclass of noun roots, or perhaps just a single noun root, that simply never occurs in the genitive case (perhaps because the resulting form is phonologically awkward or because it sounds like a different, vulgar word). Let's assume that the noun root *wup* is of this type, allowing only



the nominative and accusative suffixes. One solution to this problem, *not* generally recommended, is to copy the two allowable case suffixes to a new LEXICON (let's call it `NomAcc`) and point `wup` to the more limited lexicon as in Figure 4.23.

While this solution will work, the copying of identical morphemes to multiple places in the grammar is inherently dangerous, especially when forms and their ramifications become more complex. Maintaining multiple copies of the same morpheme is generally a Bad Thing; if you edit one of them, you may well forget to edit the other copies in parallel.

The better solution is to create the new sublexicon as before, extracting out the subset of suffixes from `CaseEnd` and creating an empty continuation from the old `CaseEnd` LEXICON to the new subset, as shown in Figure 4.24. Note that this solution provides a valid target for `wup` while still maintaining `CaseEnd` as a valid target for the thousands of more regular nouns in the language. You can, and may need to, carry such subdivisions down to lower levels as you find new idiosyncratic noun roots. If you do it carefully, you can avoid re-editing thousands of entries that point to the original `CaseEnd` lexicon. Your final result may look something like Figure 4.25.

Choosing and Using `MultiChar_Symbols`

Symbol Tags Xerox finite-state morphology systems store feature information including category or part-of-speech, aspect, mood, person, number, gender, etc. in the form of multicharacter-symbol TAGS such as `+Noun`, `+PresInd`, `+1P`, `+Sg`, and `+Masc`. These tags are in fact just symbols, manipulated exactly like the alphabetic symbols **a**, **b**, and **c**, but they have multicharacter print names. There is nothing magic about `+Noun`, `+PresInd` and `+Sg`; they must all be chosen and declared by the linguist in the `MultiChar_Symbols` section at the top of the `lexc` source file.

Because languages differ so much in the distinctions they make and in the terminology traditionally used to describe those distinctions, imposing rigid rules for defining multicharacter-symbol tags is both impossible and undesirable. However, there is at least some benefit to be had if reasonably similar languages adopt, where appropriate, the same tags to mark the same phenomena. If nothing else, this facilitates maintenance of the systems by those who may have to move back and forth

```

LEXICON Nouns
kirit      CaseEnd ;
wadil      CaseEnd ;
ridok      CaseEnd ;
faarum     CaseEnd ;
! and thousands of others

wup        NomAcc ; ! the idiosyncratic exception

LEXICON CaseEnd
u          # ;      ! the nominative suffix
a          # ;      ! the accusative suffix
i          # ;      ! the genitive suffix

LEXICON NomAcc
u          # ;      ! the nominative suffix (copied)
a          # ;      ! the accusative suffix (copied)
           # ;      ! N.B. no genitive here

```

Figure 4.23: A Suboptimal Solution for Idiosyncratic Nouns

```

LEXICON CaseEnd
           NomAcc ; ! an empty continuation to NomAcc
i          # ;      ! the genitive suffix

LEXICON NomAcc
u          # ;      ! the nominative suffix
a          # ;      ! the accusative suffix

```

Figure 4.24: Factoring Out an Idiosyncratic Subset of Suffixes

```
LEXICON CaseEnd      ! for normal productive nouns
                    ! that take all three case suffixes
                    Nom ;
                    Acc ;
                    Gen ;

LEXICON NomAcc       ! for a few nouns that take
                    ! only the nom. and the acc.
                    Nom ;
                    Acc ;

LEXICON NomGen       ! for a few nouns that take
                    ! only the nom. and the gen.
                    Nom ;
                    Gen ;

LEXICON Nom
u                    # ;

LEXICON Acc
a                    # ;

LEXICON Gen
i                    # ;
```

Figure 4.25: Avoid Multiple Copies of Morpheme Entries

from one language to another.

The following guidelines and examples are offered as helpful suggestions, and you are encouraged to follow them, unless of course you have any good reason for doing otherwise. Above all, do not get overly wrought about tag names; remember that

- Most end users will probably never see them;
- Tags don't really mean anything, except as you define them and contrast them with other tags in your system—don't get hung up on how tags are spelled;
- For your own convenience, during development and testing, you want to choose tag names that are reasonably mnemonic and yet are not too long to type;
- It is absolutely trivial to change tag names later, using replace rules and the composition operation.

Some General Principles for Tag-Name Choice

1. **Prime Directive:** Use morphological-analysis tags that are appropriate for indicating the distinctions in the language being analyzed. Do not try to force your language into a descriptive framework that is foreign to it. If there is an established and well-motivated linguistic vocabulary for describing your language, consider defining tags that evoke that vocabulary.
2. **Secondary Directive:** Where a set of words act the same syntactically, i.e. where a set of words fit into the same syntactic frames, then they should probably be analyzed with the same tags. Where two words act differently in the syntax, they should probably be analyzed with distinct strings of tags. Use tags, and strings of tags, consistently.
3. **Tertiary Directive:** Use the tags and tag orders which have already been used in grammars for related languages, unless this violates the Prime Directive or Secondary Directive.

A list of tags previously-used in **Xerox** systems is found in the Recommended-Tags Appendix on page 585. If you can select from them, consistent with the directives just listed, it will somewhat facilitate future maintenance.

Separated Dependencies

The Problem of Separated Dependencies Most languages build words by concatenating morphemes together, and such concatenation is pretty well modeled by the continuation classes provided in *lexc*. Other rarer but well-attested morphotactic phenomena such as infixation, reduplication and interdigitation are not handled

well at all by continuation classes alone and will be discussed in Chapter 9. But even for languages with concatenative morphotactics, continuation classes fail to capture what are known as SEPARATED DEPENDENCIES, DISCONTIGUOUS DEPENDENCIES or LONG-DISTANCE DEPENDENCIES. Using finite-state filters, and the Flag Diacritics to be presented in Chapter 8, we can remedy the limitations of **lexc** and similar languages that model morphotactics with continuation classes.

In **lexc**, each morpheme is assigned a continuation class, which is either the name of a LEXICON or #, a special continuation class indicating the end of word. Continuation classes control which morphemes can appear next in the word, and the linguist can define as many continuation classes as are necessary to limit which morphemes can appear next. The contiguous dependencies of the language, i.e. the dependencies between a morpheme and the morphemes that can appear next, directly after it, can therefore be captured perfectly well. The key word here is *next*.

However, in Arabic and many other languages, even in Esperanto, there are dependencies or restrictions involving the co-occurrence of morphemes that are not contiguous in the word. Continuation classes can control which morphemes appear *next*, but they cannot in general control what other morphemes appear in the overall word. To focus on the issue of separated dependencies, ignoring some other complication of Arabic, we introduce a simplified language that we will call Pseudo-Arabic. Assume that Pseudo-Arabic has a concatenative morphology with the following characteristics:

1. The noun roots include *kital*, *larum* and *faalin*.
2. There are three definite case suffixes, *u* for definite nominative, *a* for definite accusative, and *i* for definite genitive. The word *larumu* is therefore definite and nominative, which we will indicate on the lexical side using two multi-character tags +Def and +Nom. Accusative case will be marked +Acc, and genitive +Gen.
3. There are also three indefinite case suffixes, which we will notate here as *uN* for indefinite nominative, *aN* for indefinite accusative, and *iN* for indefinite genitive. Therefore *faalinaN* is indefinite accusative, which we will mark on the lexical side as +Indef+Acc.

A **lexc** grammar that captures these partial facts is shown in Figure 4.26. Type this grammar into a file, compile it, and test it against the stated facts. Ask yourself how else the same language might be defined and experiment with the alternative codings.

All of the strings generated so far by the grammar are correct and should be maintained as we continue to expand and refine the example. It's now time to introduce the complicating facts of Pseudo-Arabic (which also hold for Modern Standard Arabic).

```
Multichar_Symbols +Noun +Def +Indef +Nom +Acc +Gen
uN aN iN

LEXICON Root
kital    N ;
larum    N ;
faalin   N ;

LEXICON N
+Noun:0   CaseEnds ;

LEXICON CaseEnds
+Def+Nom:u    # ;
+Def+Acc:a    # ;
+Def+Gen:i    # ;

+Indef+Nom:uN # ;
+Indef+Acc:aA # ;
+Indef+Gen:iN # ;
```

Figure 4.26: A First Model of Pseudo-Arabic

1. The definite article, which we will model here as *al*, is optional, and it simply concatenates onto the beginning of the noun root as in *alkitalu*.
2. If the definite article prefix is present, the word must have a definite case suffix. That is, *kitalu*, *kitaluN* and *alkitalu* are valid words, but **alkitaluN* is ill-formed because it combines an overt definite article prefix with an indefinite case suffix.
3. The prefix *bi* is like an English preposition (“for”); like *al* it is optional and, if present, simply concatenates onto the beginning of the word. If both *bi* and *al* are present, *bi* must appear before *al*.
4. However, *bi* governs the genitive case, so if a noun has a *bi* prefix, it must have a genitive case suffix, either *i* or *iN*. If both *bi* and *al* are present, then the case suffix must be the definite genitive *i*.

Pseudo-Arabic thus displays a classic case of separated dependencies, where prefixes on one end of the word require restrictions on discontinuous suffixes that appear on the other end of the word. Such a separated dependency is awkward to encode directly in any **lexc**-like notation. The problem is inherent to finite-state machines because at any point during lookup, when a machine is in a particular state, it has no stack or other memory of what it has seen before. Thus when a finite-state lookup algorithm starts to explore the subnetwork of Arabic case suffixes, it can’t remember if it previously found an *al* or a *bi* prefix. The next transition is determined only by the next input symbol.

As always, our ultimate goal is to write a grammar that accepts all the valid words of a language and none of the invalid ones. Try to write a **lexc** grammar that captures the facts of Pseudo-Arabic. Use `+PREP+` on the lexical side to mark *bi* and `+ART+` to mark *al*.

One solution, a bad one, is shown in Figure 4.27. The model is bad both because it is hard to read and because it raises the old problem we saw in Figure 4.23: multiple copies of morphemes. Note that the article *al* is repeated twice and the noun roots have to be repeated no fewer than four times. Just imagine the problem if there were in fact tens of thousands of noun roots to list and maintain; the lexicographer would have to make every addition and every correction in four different places. In practice, such a repetitious **lexc** description is unmaintainable.

Constraining Separated Dependencies with Filters The irony is that the network created by compiling the grammar in Figure 4.27 is in fact correct; to model separated dependencies using a pure finite-state transducer, the noun roots and the article do have to be copied. What we need, until we introduce Flag Diacritics (see page 439), is a way to write the **lexc** grammar in a straightforward way so that the linguist maintains only a single copy of the noun roots, and so that the finite-state tools copy the relevant structures automatically and reliably at compile time. We do this by 1) writing a **lexc** grammar that purposely overgenerates and then 2), writing

```

Multichar_Symbols +Noun +Def +Indef +Nom +Acc +Gen
  +Prep+ +Art+ uN aN iN
LEXICON Root
  BiPref ;
  AlPref ;
  BareNounRoots ;
LEXICON BiPref
bi+Prep+:bi BiAl ;
LEXICON BiAl
al+Art+:al BiAlNounRoots ;
  BiNounRoots ;
LEXICON AlPref
al+Art+:al AlNounRoots ;
LEXICON BareNounRoots
kital N ;
larum N ;
faalin N ;
LEXICON BiAlNounRoots
kital BiAlN ;
larum BiAlN ;
faalin BiAlN ;
LEXICON BiNounRoots
kital BiN ;
larum BiN ;
faalin BiN ;
LEXICON AlNounRoots
kital AlN ;
larum AlN ;
faalin AlN ;
LEXICON N
+Noun:0 CaseEnds ;
LEXICON BiAlN
+Noun:0 DefGenCaseEnd ;
LEXICON BiN
+Noun:0 GenCaseEnds ;
LEXICON AlN
+Noun:0 DefCaseEnds ;
LEXICON CaseEnds
  DefCaseEnds ;
  IndefCaseEnds ;
LEXICON GenCaseEnds
  DefGenCaseEnd ;
  IndefGenCaseEnd ;
LEXICON DefCaseEnds
+Def+Nom:u # ;
+Def+Acc:a # ;
  DefGenCaseEnd ;
LEXICON DefGenCaseEnd
+Def+Gen:i # ;
LEXICON IndefCaseEnds
+Indef+Nom:uN # ;
+Indef+Acc:aN # ;
  IndefGenCaseEnd ;
LEXICON IndefGenCaseEnd
+Indef+Gen:iN # ;

```

Figure 4.27: A Bad Model of Pseudo-Arabic

filter networks that are composed with the lexicon to eliminate the overgeneration. The overgenerating *lexc* grammar is shown in Figure 4.28.

This grammar is much simpler to read and maintain, but by itself it overgenerates outrageously, including impossible string pairs such as the following:

Lexical: al+Art+kital+Noun+Indef+Acc
Surface: alkitalaN

Lexical: bi+Prep+larum+Noun+Def+Acc
Surface: bilaruma

Lexical: bi+Prep+Art+faalin+Noun+Indef+Gen
Surface: bialfaaliniN

A string like “alkitalaN” is, of course, ill-formed because the definite article prefix *al* co-occurs with an indefinite case suffix. Our next step is to characterize the strings that are bad and remove them from the network.

- Note that some of the ill-formed strings on the lexical side can be characterized in regular expressions such as

```
?* %+Art%+ ?* %+Indef ?*
```

or, equivalently

```
$( %+Art%+ ?* %+Indef ]
```

That is, any string pair that contains *+Art+* followed at any distance by *+Indef* on the lexical side is ill-formed. (The *\$* operator in **Xerox** regular expressions means “contains”.)

- Similarly, we can characterize the remaining bad lexical strings with the following regular expression.

```
$( %+Prep%+ ?* [ %+Acc | %+Nom ] ]
```

That is, any string pair that has *+Prep+* followed at any distance by either *+Acc* or *+Nom* on the lexical side is ill-formed.

- All the ill-formed lexical strings will therefore be matched by the following regular expression:

```
[ $( %+Art%+ ?* %+Indef ]  
| $( %+Prep%+ ?* [ %+Acc | %+Nom ] ]  
]
```

```

Multichar_Symbols +Noun +Def +Indef +Nom +Acc +Gen
  +Prep+  +Art+ uN aN iN

LEXICON Root
    BI ;
    AL ;

LEXICON BI
bi+Prep+:bi  AL ;

LEXICON AL
al+Art+:al   Nouns ;
             Nouns ;

LEXICON Nouns
kital      N ;
larum      N ;
faalin     N ;

LEXICON N
+Noun:0     CaseEnds ;

LEXICON CaseEnds
+Def+Nom:u   # ;
+Def+Acc:a   # ;
+Def+Gen:i   # ;

+Indef+Nom:uN # ;
+Indef+Acc:aN # ;
+Indef+Gen:iN # ;

```

Figure 4.28: A Pseudo-Arabic **lexc** Grammar that Purposely Overgenerates

- As we saw in the **xfst** chapter, page 94, if A denotes a language (a set of strings), then $\sim A$ denotes the COMPLEMENT of A , the language that contains all and only the strings that are not denoted by A . Therefore, the expression in Figure 4.29 will match all the valid strings (the complement of the invalid strings) on the lexical side of our overgenerating **lexc** grammar. This regular expression will be our FILTER; let us assume that it resides in file `filter.regex`.

```
~[  $[ %+Art%+ ?* %+Indef]
   | $[ %+Prep%+ ?* [ %+Acc | %+Nom ] ]
   ] ;
```

Figure 4.29: A Regular Expression Filter in file `filter.regex`

- Let us assume that the overgenerating grammar in Figure 4.28 has been compiled and saved to file `pseudo-arabic-lex.fst`. Now to remove the overgeneration, we **load stack** our overgenerating network onto the **xfst** stack, and then we compile our filter using **read regex**, which puts the filter network on The Stack on top of the overgenerating network. The state of The Stack is shown in Figure 4.30.

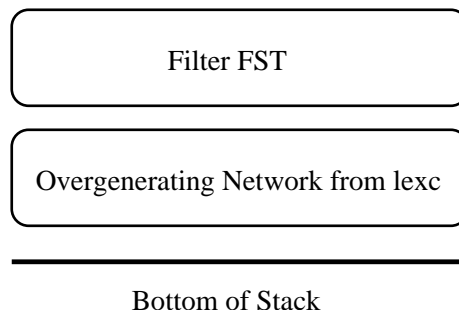


Figure 4.30: Overgenerating **lexc** Network and Filter on The Stack

- Now we simply invoke **compose net** and, in the composition operation, all and only the well-formed strings on the upper side of the **lexc** network are matched. The ill-formed strings are not matched and are therefore eliminated in the process of the composition. The commands are listed in Figure 4.31. Note that the filter in this case is composed on top of the **lexc** lexicon, which is where the tags are visible. Composition is an ordered operation, and the relative position of the arguments on The Stack is vitally important.

```

lexc>[0]: clear stack
lexc>[0]: load stack pseudo-arabic-lex.fst
lexc>[1]: read regex < filter.regex
lexc>[2]: compose net
lexc>[1]: save stack pseudo-arabic.fst

```

Figure 4.31: Composition of a Filter to Remove Overgeneration

```

lexc>[0]: clear stack
lexc>[0]: read regex @re"filter.regex"
.o. @"pseudo-arabic-lex.fst" ;
lexc>[1]: save stack pseudo-arabic.fst

```

Figure 4.32: Composition of a Filter to Remove Overgeneration

- Computing with regular expressions rather than on The Stack, the commands in Figure 4.32 are equivalent.
- All necessary copying, as for the noun roots, is performed automatically and reliably in the process of the composition.

replace rules as Filters In the Pseudo-Arabic example, we eliminated the over-generation by composing a filtering network on the lexical side. The filter matched all and only the valid strings, eliminating the invalid strings in the composition. It is also possible to visualize the filter as replace rules that eliminate the ill-formed strings, and this may be more intuitively satisfying to some linguists.

The trick is to write replace rules that map the offending ill-formed strings to the null automaton. The null automaton denotes the empty language and is not to be confused with the empty string that we denote as 0 (zero) or ϵ (epsilon). The empty language is the language that contains no strings at all, not even the empty string. One of the infinitely many ways of denoting this empty language in regular expressions is $\sim [?^*]$, i.e. the complement of the universal language. Alternatives include $\sim \$[]$, $[? - ?]$, $[a - a]$, etc. In any case, the filter, expressed as replace rules, could be written as shown in Figure 4.33. In the end, of course, it all amounts to the same thing. replace rules are compiled into finite-state transducers, and here they are composed on top of the overgenerating Pseudo-Arabic lexicon to eliminate the ill-formed strings.


```
[ ~[?*] <- $[ %+Art%+ ?* %+Indef ] ]
.0.
[ ~[?*] <- $[ %+Prep%+ ?* [ %+Acc | %+Nom ] ] ] ;
```

Figure 4.33: Filters Expressed as Replace Rules. These rules map ill-formed strings to the null language, which deletes them.

Exercises

Pseudo-Arabic Compile the overgenerating Pseudo-Arabic grammar as shown in Figure 4.28. How many words (pairs of strings) are there in the network? How many states and arcs are in the network? (The **lexc** compiler itself will tell you. Write down these numbers.) Save the overgenerating network to file.

Now write a filter regular-expression using the notation in Figure 4.29. Compose the networks, making sure that the filter is composed on top of the overgenerating lexicon transducer. How many strings are left? How many states and arcs are in the final restricted network? Filter restrictions can be expensive in memory space because parts of the original network get copied, perhaps several times.

Repeat the restriction but using the replace rules notation to write the filters (as in Figure 4.33). Confirm that these rules do the job equally well in taming the overgeneration of Pseudo-Arabic.

Esperanto Overgeneration In Section 4.3.5, we wrote an Esperanto grammar in **lexc** that handled nouns and adjectives and verbs. In fact, this grammar overgenerates in certain ways that we can now fix with filters.

1. Recompile the Esperanto grammar; following the original descriptions, it should accept an infinite number of strings (i.e. **lexc** will inform you that it is “circular”, that the network has at least one loop in it).
2. The strings will include examples like the following

```
Lexical:      hund+Noun+Aug+Aug+Aug+NSuff+Sg
Surface:      hundegegego
```

Let us assume for this exercise that any string with more than one +Aug tag on the upper-side is ill-formed. Similarly, any string with multiple +Fem or +Dim tags is ill-formed.

3. Write suitable filter rules to restrict the augmentative, diminutive and feminine overgeneration of your **lexc** Esperanto grammar. Compose the filters

on top of the lexicon transducer. How do these restrictions affect the size of your network (states, arcs, number of words)?

4. In Esperanto nouns with the *ge* prefix, meaning “masculine or feminine”, the feminine *in* suffix cannot co-occur. In other words, a lexical-side string with MF+ followed at any distance by +Fem is ill-formed. Write a lexical-side filter that kills words like **gehundinoj*, leaving good words like *gehundoj* and *gehundegoj*.
5. Let us assume (it may not be strictly true) that the *ge* prefix for masculine and feminine requires the plural suffix *j*. In other words, assume that the *ge* prefix (marked MF+ on the lexical side) is incompatible with +Sg. Write a suitable filter and compose it on top of the overgenerating lexicon transducer. In the process, words like **gehundo* should disappear, leaving good words like *gehundoj*.

4.5.4 Properties

lexc provides commands to mark a network with a list of properties, documenting information such as the natural language modeled, the author, the date of creation, the release version, etc. Such information becomes especially important in commercial applications. Here is the property list, displayed with the **props** command, from an Italian-morphology network built in 1995:

```
LANGUAGE: ITALIAN
CHARENCODING: ISO8859/1
VERSIONSTRING: "1.2"
COPYRIGHT: "Copyright (c) 1994 1995 Xerox Corporation."
COPYRIGHT2: "All rights reserved."
TIMESTAMP: "29 November 1995; 16:30 GMT"
AUTHOR: "Antonio Romano, Cristina Barbero, Dario Sestero"
```

Each line is of the form

```
PROPERTYNAME: PROPERTYVALUE
```

and if the PROPERTYVALUE contains spaces, it should be enclosed in double quotes. The choice of PROPERTYNAMEs and PROPERTYVALUEs is free, to be defined as necessary between the developer and the consumer. These properties do not affect in any way the operation of the network.

In practice, the most common way of defining a property list is to edit one using a text editor and store it to a file. The **lexc** command **reset-props** reads the property list from such a file and assigns the properties to the SOURCE or the RESULT network, overwriting any previous properties. The **add-props** command reads a suitably formatted property file and adds the properties to the network.

save-props writes the property list of a network out to file in the format expected by **add-props** and **reset-props**. To have the properties of a network displayed to the screen, use the **props** command.

4.5.5 **lexc** Strategy Summary

Good **lexc** programmers have learned the following facts and strategies:

- A LEXICON should subsume a generally coherent collection of morphemes.
- A LEXICON is a potential *target* for continuations from other morphemes.
- **lexc** allows each entry in a LEXICON to have its own continuation class. However, one style of **lexc** programming tries to maximize the coherence of each morpheme class, with each entry in a LEXICON sharing the same continuation class. This style of programming is facilitated by Flag Diacritics, which will be presented in Chapter 8.
- A **lexc** source file should avoid having multiple copies of the same morphemes, which are difficult to maintain in parallel.
- Separated dependencies cannot be modeled satisfactorily using **lexc** continuation classes alone. It is often better to write a **lexc** grammar that purposely overgenerates and then eliminate the overgeneration via the composition of suitable filters. Flag Diacritics (see Chapter 8) can have the same effect as filters while avoiding an increase in the size of the network.
- When compiling **lexc** grammars, read all error messages and diagnostics carefully. Be aware when you have loops in your grammars. Be aware of filters that may cause your networks to explode in size.
- Attaching properties to your networks is useful for long-term maintenance, especially in commercial development projects that involve multiple releases and versions.

4.5.6 Common **lexc** Errors

Everyone makes mistakes in programming, and certain **lexc** mistakes are so common as to be expected. At the risk of reinforcing the bad, some classic errors are listed here.

Error: Putting a Semicolon after the LEXICON Name

Because semicolons are placed after *entry* lines, beginning programmers often overgeneralize and try to put them after LEXICON names as well.

```
LEXICON Nouns ;      ! this semicolon is an error
dog      N ;
cat      N ;
```

In short, there should be *no* semicolon after a LEXICON name. The compiler will clearly signal such errors.

Error: Forgetting to Put a Semicolon after an Entry

Each entry must terminate with a semicolon, and the failure to type the semicolon is a common error.

```
LEXICON Nouns
dog      N ;
cat      N      ! Error: a missing semicolon
horse    N ;
```

In these cases, where a semicolon is missing, the compiler effectively sees a single ill-formed entry that looks like

```
cat N horse N ;
```

and signals an error. If you read the error messages carefully, these missing semicolons should be easy to find and fix.

Error: Failure to Declare a Multicharacter Symbol

One of the more serious and troubling errors, because it cannot be caught by the compiler, is the failure to declare a multicharacter symbol. In the example in Figure 4.34, the linguist obviously intended to use +Sg and +Pl as tags (single symbols), but because they are not declared as `Multichar_Symbols`, *lexc* will happily and silently explode them into separate concatenated symbols +, **S**, **g** and +, **P**, **l**.

Errors in symbol declaration can be particularly mysterious because the lexical string displayed as “girl+Noun+Sg” looks exactly like the string displayed as “girl+Noun+Sg” even if the first involves a single symbol +**Sg** and the second three separate symbols +, **S** and **g**. Things get really mysterious if a subsequent rule or filter depends on a tag like +Sg being a single character and it was never declared to be one. To the linguist trying to debug such a rule, it looks like it should fire, but it doesn’t.

The failure of a rule to fire on a string that “looks right” is often a sign that the string is not being divided up into symbols in the manner you intended. Learn to suspect undeclared multicharacter symbols.

One of the best and simplest mechanical tests for such problems is to check the alphabet of the network, which involves the following:

```

Multichar_Symbols  +Noun +Verb

LEXICON Root
boy      N ;
girl    N ;

LEXICON N
+Noun:0   Number ;

LEXICON Number
+Sg:0     # ;      ! error: failure to declare +Sg
+Pl:s     # ;      ! error: failure to declare +Pl

```

Figure 4.34: Failure to Declare Multicharacter Symbols

1. Push your compiled network onto the **xfst** stack,
2. Invoke the **upper-side net** utility to compute the network that accepts just the upper-side or lexical language, and
3. Invoke the **print labels** utility to see the full set of symbols.

Similarly, do the same test but invoking **lower-side net** and then **print labels** to see the lower-side or surface symbols. On the upper side, where multicharacter tags usually lurk, try to identify any omissions; this kind of negative testing is most successful if you know what you're looking for. For a more positive test, recall that (by **Xerox** convention) multicharacter tags always include a punctuation mark, like the plus sign or the circumfix or square brackets. If any of these symbols turn up in your lexical labels, you should suspect that a **lexc** string you intended as a multicharacter symbol was never declared and so was exploded apart into separate alphabetic symbols.

In other cases, and especially on the surface side, you will often find multicharacter symbols that you didn't expect. These often result from errors introduced inadvertently in **xfst** regular expressions or in **twolc**. We'll cover these and many more ways to use the finite-state tools to test your networks in Chapter 7.

Error: Trying to do Concatenation in an Entry Line

Each **lexc** entry consists of just two parts, a FORM and a CONTINUATION CLASS. Intuitively, many beginners try to place multiple forms on the same line, assuming that they will simply be concatenated. This would in fact be an improvement to the **lexc** syntax, but it just isn't allowed.

LEXICON Noun

```
dog +Noun:0 Number ;      ! error: two forms in one entry
girl +Noun:0 Number ;     ! error: two forms in one entry
```

Of course, concatenation can be performed within angle brackets (<>), where the notation of regular expressions is available, but here you must respect the conventions of *xfst* regular expressions, spacing out the symbols (or surrounding them with curly braces) yourself

LEXICON Noun

```
< d o g %+Noun:0 > Number ;      ! legal concatenations
                                     ! inside angle brackets
< {girl} %+Noun:0 > Number ;
```

4.6 Integration and Testing

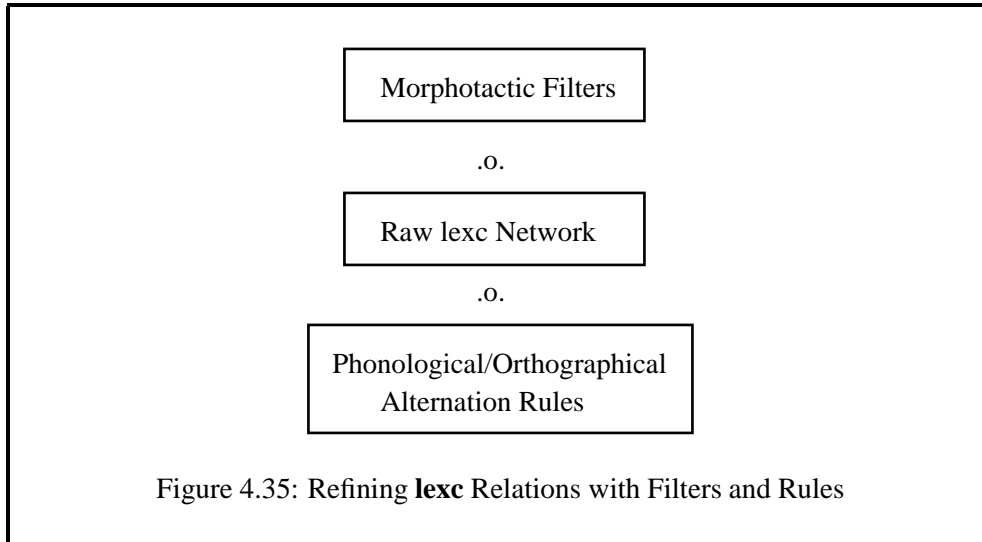
4.6.1 Mind Tuning for *lexc* Integration

A *lexc* grammar is usually only one part of a full grammar, which may consist of several *lexc* grammars, *twolc* grammars, and various kinds of filters and mappings described using regular expressions, including *xfst* replace rules. It's important to get a feeling for how a larger system will fit together, and this requires some mind tuning and review.

Thinking about Languages

The first kind of mind tuning needed by a new finite-state developer is to think of regular expressions, *lexc*, and other finite-state notations as formal ways to specify regular LANGUAGES and regular RELATIONS between languages. Here we are using the term LANGUAGE in its formal and somewhat perverse sense, meaning just a set of strings, where each string is composed of symbols from an alphabet. From our formal modeling point of view, a natural language such as Mongolian is simply a set of strings; and our goal as linguists writing a morphological analyzer of Mongolian is to define a formal language (another set of strings) that corresponds as closely as possible to real Mongolian. We further specify a relation between the language of Mongolian-like strings and another language of analysis strings (lexical strings) that we define ourselves. The linguist models Mongolian morphology much like an astrophysicist models the solar system, like an economist models an economic system, or like a botanist models photosynthesis.

Ideally, our formal Mongolian system should accept and generate all the strings that characterize real Mongolian, and it should not accept or generate any string which is not acceptable in real Mongolian. In other words, our formal Mongolian system should ideally never undergenerate or overgenerate. In practice, linguists are imperfect, and natural languages are somewhat fuzzy and moving targets—we must content ourselves with closer and closer approximations.



Most natural languages have some structure to their words; they build words from parts called morphemes, and there is a grammar that controls how valid words are formed. The study of such word-building rules, to discover the relevant grammar, is called MORPHOTACTICS or sometimes MORPHOSYNTAX; and **lexc** is designed principally to allow linguists to notate the morphotactic grammars that they discover, and to compile these grammars into finite-state networks that we can compute with.

We have already shown above (page 267) that **lexc** grammars in themselves are awkward or unsuitable for describing separated dependencies and other non-concatenative phenomena. And we showed ways that finite-state filters could be composed on top of **lexc** grammars to impose desired constraints. Indeed, we will often find it convenient and perspicuous to write a **lexc** grammar that purposely overgenerates, denoting an intermediate language or relation that will later be made more constrained and closer to the real natural-language target via composition with suitable morphotactic filters.

Just as raw **lexc** grammars often denote languages that are unfinished morphotactically, and need some subsequent filtering, the same languages are (except for preternaturally regular examples like Esperanto) usually unfinished phonologically or orthographically as well. That is, **lexc** grammars typically produce surface strings that need some degree of, and sometimes a lot of, modification or alternation before they look like the real strings of the target language. The linguist must discover the alternations applicable for each language, which may reflect orthographical conventions and/or phonological processes such as deletion, epenthesis, and assimilation. These alternations are notated using finite-state rules, either the replace rules presented in Sections 2.4.2 and 3.5.2 or the **twolc** rules to be introduced in Chapter 5. Such rules are compiled into finite-state transducers and are typically composed on the bottom of the **lexc** grammar as shown in Figure 4.35.

For example, the lower side of the original **lexc** grammar might produce an INTERMEDIATE STRING like *try+s* that needs to be mapped to *tries* before it is acceptable. Similarly, it might produce a Spanish-like string *vez+s* that needs to be mapped to the ultimate target *veces*, or an Aymara-like string *uta+ma+na+kappa+raki-i+wa* that needs to be mapped to *utamankapxarakiwa*. Such mappings are notated using **twolc** or replace rules, are compiled into transducers, and are typically composed on the bottom of the lexicon transducer. The notion of rule application reduces to the finite-state composition operation.

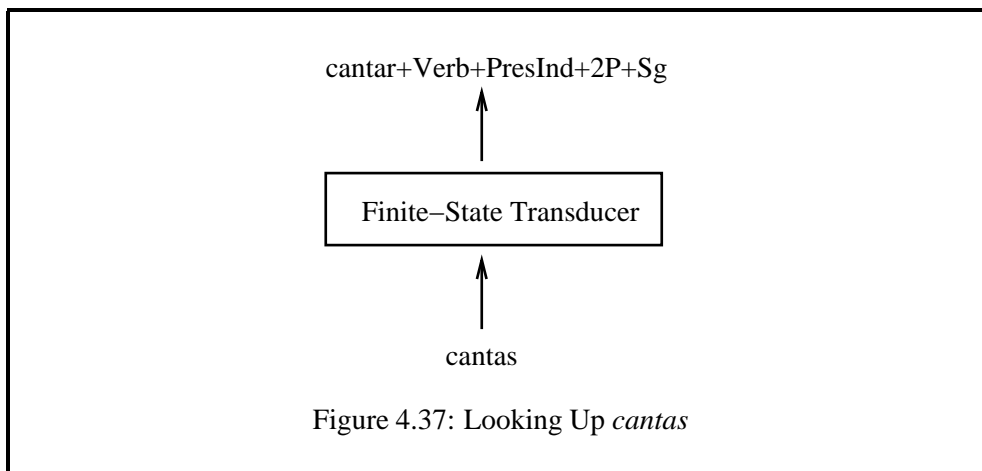
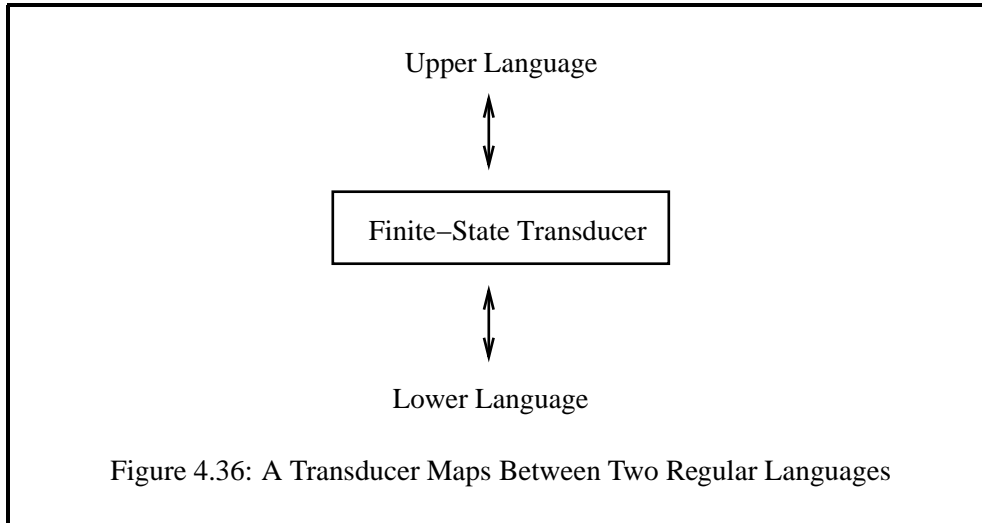
Just as it is awkward, impractical and often even undesirable to capture all the morphotactic facts in **lexc** alone, it is usually just as awkward, impractical and undesirable to capture all the morphophonological facts directly in **lexc**. Just as it is usually desirable to write a **lexc** grammar that overgenerates morphotactically, it is usually desirable to write **lexc** grammars that initially produce somewhat abstract and regularized surface forms that need subsequent fixing via alternation rules.

Thinking about Transducers

A finite-state or regular language, in our formal sense, is just a set of strings; and a simple finite-state automaton is an abstract machine that accepts all and only the strings of a regular language. A finite-state transducer is an abstract machine that maps between the strings of two regular languages. More formally, a transducer encodes a RELATION between two regular languages. At **Xerox**, we always picture transducers as having an upper side and a lower side. This up-and-down visualization is a **Xerox** convention, not necessarily shared with other researchers in finite-state networks, and one could alternatively visualize transducers left-to-right, right-to-left, or even upside down from the **Xerox** convention. However, this book and other **Xerox** documentation are at least consistent, and it is terribly important for learners to become comfortable with the distinction between up and down in a transducer.

Each transducer accepts two regular languages. The upper-side or lexical language is the set of strings that match the paths of symbols on the upper side of the arcs. The lower-side or surface language is the set of strings that match the paths of symbols on the lower side of the arcs. In addition to accepting the two languages (each on its appropriate side), the transducer describes a relation or mapping between the strings of the two languages such that for every string matched on the upper side, there is a related string or strings in the lower-side language; and for every string matched on the lower side, there is a related string or strings in the upper-side language. The **Xerox** visualization of a transducer and its two languages is shown in Figure 4.36.

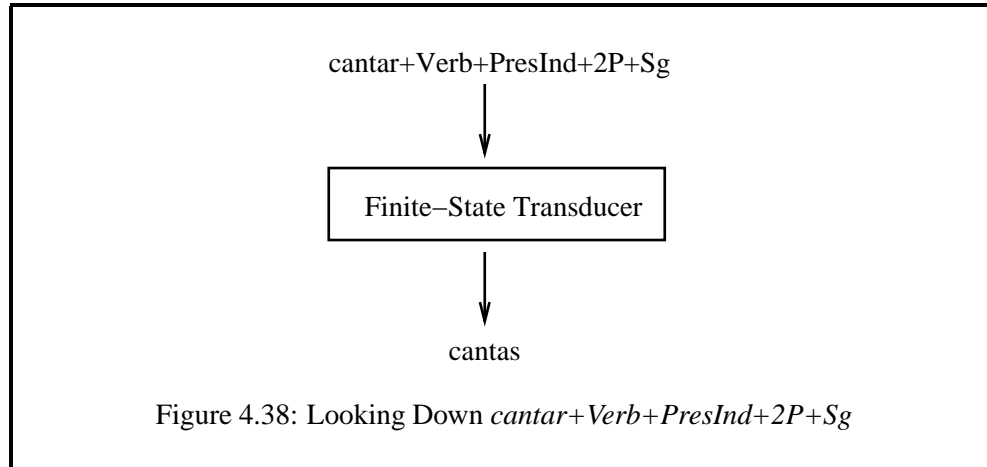
When a transducer subsumes a lexicon, typically created with **lexc**, and rules, created with **twolc** or replace rules, and perhaps filters, we call it a LEXICAL TRANSDUCER. By typical **Xerox** convention, the upper-level language in a system such as the Spanish Lexical Transducer consists of strings like



cantar+Verb+PresInd+2P+Sg

consisting of a baseform and tags, and the lower-level language consists of orthographical strings such as “cantas”. In other applications, it may be desirable to include the abstract morphophonological form of each suffix on the upper side, and the lower-side might represent phonological strings, using a suitable alphabet, rather than conventional orthographical strings.

When we LOOKUP a word, using the transducer, we apply a string to the lower side of the transducer, as in Figure 4.37. If the string is not, in fact, in the lower language of the transducer, there will be no match and no output. If however, as in the Spanish Lexical Transducer, the string “cantas” is indeed in the lower-side language, it will match, and the transducer will return the corresponding upper-side string, which happens to be “cantar+Verb+PresInd+2P+Sg”. That is, in lookup, the transducer relates the lower-level string “cantas” to the upper-



level string “cantar+Verb+PresInd+2P+Sg”. The process of lookup is also called ANALYSIS.

Transducers are inherently bidirectional and can equally well be run backwards to perform LOOKDOWN, also called GENERATION. In the case of lookdown, a string is applied to the upper side of the transducer as in Figure 4.38. If the input string is in fact in the upper language, it will match and the transducer will return one or more related strings from the surface language. If the input string is not in the upper language, there will be no match and no output.

Students are urged to review this and other relevant sections until the up/down visualization becomes second-nature. As simple as the concepts may seem, they prove to be surprisingly difficult for some students, and failure to distinguish up from down and lookup from lookdown leads to much confusion and distress.

Learn to distinguish between the upper side and the lower side of a transducer. Learn the distinction between lookup and lookdown.

Keeping up and down straight is especially important when trying to understand composition. A finite-state filter, as on page 279, that refers to tags has to be composed on the upper side of the **lexc** transducer, where the strings with the tags are. Similarly, a rule transducer to map intermediate surfacy strings into final surface strings would have to be composed on the bottom of the **lexc** transducer, where the surfacy strings are.

Keeping up and down straight is vitally important when composing transducers together.

More generally, a finite-state system may consist of many component **lexc** and

rule grammars, and rules can also be organized into multi-level cascades. Experienced developers think of each level as a regular language, and they can characterize what the strings look like at each level. Developers who cannot keep their own levels (languages) straight soon get lost.

Get used to thinking of each level in your system as a regular language of strings, and be prepared to describe what the strings look like in each of those languages. Make your various languages as consistent as possible, especially the ultimate lexical and surface languages.

Finally, experienced developers are always aware that the lower side of a final Lexical Transducer matches a language, a set of strings that corresponds as closely as possible to the strings of a real language like Spanish. Such developers are also aware that the upper side of the transducer also matches a language, this time a language consisting (in typical **Xerox** systems) of strings with baseforms followed by tags. This upper-side language is ultimately defined by the linguist, and experienced linguists put considerable thought and planning into making that language coherent, informative, consistent, as beautiful as possible, and flexible enough to support multiple projects. In Section 7.4.1, we will look at LEXICAL GRAMMARS and how they can be used for definition, testing and documentation of the upper-level language of a lexical transducer.

Thinking about Rules

The lower-side language of a **lexc** transducer typically consists of surfacy strings, but these strings are usually still abstract and unnaturally regular; they need to be mapped via rules into the genuine surface strings.

Developers are often faced with the real decision about how much of the work to do in the lexicon (with **lexc**) and how much to do with rules. The considerations are the following:

1. Aim to create an overall system that is formally beautiful, concise, maintainable and easily expandable. If an apparent simplification in one part of the system creates horrible complications in another, reexamine your strategy.
2. Aim to reduce the future work as much as possible to lexicography, the relatively simple process of adding new baseforms, each with an appropriate continuation class, to the lexicon. Try to design the system so that new regular vocabulary can be added by a native lexicographer with minimal training in **lexc**.
3. For suppletion, gross irregularities such as the English analysis of *went* as a form of *go*, or the analysis of the French *yeux* as the plural of *oeil*, definitely use the *upper:lower* notation in **lexc** rather than trying to derive such

examples via rules. This is sometimes referred to as handling an example by “brute force”. Try to do all the nasty and irregular examples during initial development, leaving only regular vocabulary, which represents the vast majority in most systems, for future lexicographers to add.

4. In most cases, irregular and superficially unpredictable mappings such as *dig/dug*, *swim/swam* and *think/thought* are also best handled in **lexc** using *upper:lower* entries. If a learner of a language simply has to memorize a phenomenon, then it’s a good candidate for brute-force handling in **lexc**.
5. Where phonological/orthographical alternations are regular, productive and predictable, try to handle them with rules rather than brute-forcing them in the lexicon. But if you find yourself writing alternation rules to handle individual words, or very finite sets of idiosyncratic words, consider going back into **lexc** and handling them by brute force.

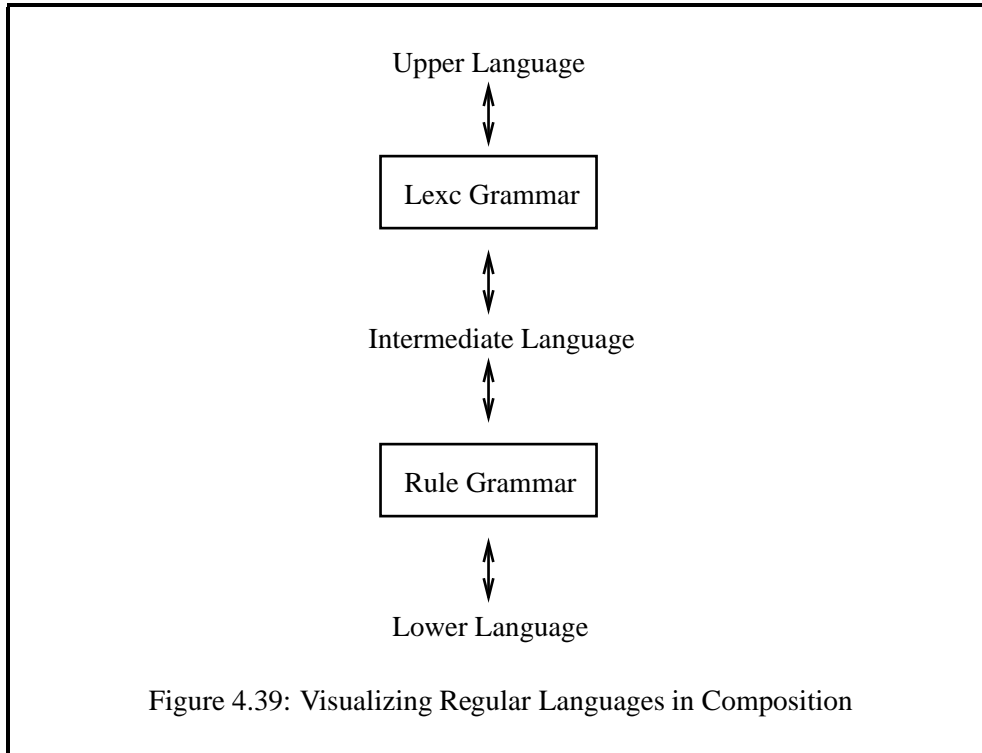
To say that there is a rule controlling certain phenomena implies predictability and productivity. When such rule-based alternations are handled by **twolc** or replace rules, the **lexc** lexicon stays simpler and the overall linguistic description is more motivated. While there is no hard-and-fast metric for judging among alternate linguistic descriptions that work, the ones which are cleanest and most concise, displaying creative laziness on the part of the developer, are preferred.

Thinking about Composition

Intermediate Languages The finite-state operation of composition has always been a difficult one for newcomers to finite-state theory. Most of the problems result from a failure to distinguish up and down in the transducers being composed and from a failure to think properly of transducers as abstract machines that relate two regular languages.

Consider the case in Figure 4.39, where a transducer compiled from a set of rules (either **twolc** or replace rules) is to be composed *on the bottom* of a transducer compiled from a **lexc** grammar. As usual, the **lexc** transducer maps between an upper-level regular language and a lower-level regular language. Similarly, the rule transducer maps between an upper-level language and a lower-level language. An important key to understanding composition is that if one transducer (here from the rule grammar) is being composed *under* another transducer (here from the **lexc** grammar), then the lower-side language of the **lexc** grammar will be intersected with the upper-side language of the rule grammar. In Figure 4.39, the shared intersected language is called the INTERMEDIATE LANGUAGE.

From a generation (lookdown) point of view, the lower-side output of the **lexc** network becomes the upper-side input of the rule network. Conversely, from the analysis (lookup) point of view, the upper-side output of the rule network becomes the lower-side input of the **lexc** network. In Figure 4.39, the upper-side language of the **lexc** network is not visible to the rule network; all the rules see are the



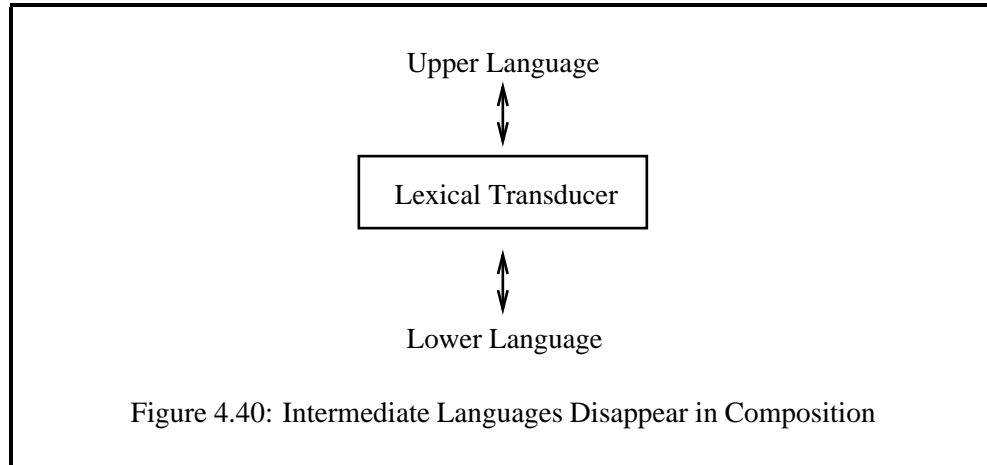
lower-side strings of the **lexc** network because, in this case, the rules are being composed on the bottom of the lexicon. Similarly, the lower-side language of the rule network is not visible to the **lexc** network. In the process of composition, as shown in Figure 4.40, the intermediate language disappears, and the resulting Lexical Transducer maps directly between the original upper-level language of the **lexc** network and the original lower-level language of the rule network.

Comparing Composition with UNIX Pipes Those students who are familiar with **UNIX** pipes may benefit from the following comparison, which can be pushed only so far. Consider a **UNIX** command like the following, where `input` is a text file containing 1000 words, with one word to a line.

```
unix> cat input | sort | uniq > output
```

The built-in **UNIX** utility **cat** will open and list the `input` file, and its output will be piped to the **UNIX** **sort** utility, which will sort it alphabetically; then that sorted output is in turn piped to the **uniq** utility, which removes duplicate words; the final output from **uniq** is then directed to another file called `output`.

This piping process is superficially very similar to what happens in finite-state generation (lookdown). The `input` file, a set of strings, is as it were applied to the “top” or input side of the **sort** utility, which performs a mapping, yielding an intermediate lower-side output that becomes the input (upper side) to the the **uniq**



utility, which in turn has its own lower-side output. There is nothing to prevent us from constructing elaborate cascades of piped utilities, and once such a piped command is constructed and tested, we might create an alias for it and reuse it frequently, eventually even forgetting that intermediate results are being produced and piped from one utility to the next. Think of the pipes (indicated in **UNIX** with vertical lines) as composition operators (\circ), and of the **sort** utility as being composed on top of the **uniq** utility; this is a superficial way of looking at composition that is useful to some students. As with piped **UNIX** utilities, it is vitally important to compose transducers in the right order, so that they “feed” each other properly; and it must be stressed that the piping analogy relates to transducers only when viewed in a generation or lookdown direction.

It is dangerous to push an analogy too far, and here we have to stop. In fact, finite-state transducers and composition are much more interesting than pipable **UNIX** utilities. For one thing, when you compose two or more transducers together, the component transducers are literally combined together into a single transducer. It’s as if, in our analogy, the utilities **sort** and **uniq** were being combined together algorithmically into a single program, which is definitely not possible in **UNIX**. Pipes always go through the step-by-step generation of the ultimate output, whereas a composed transducer produces the output in a single step, which is far more efficient. Finally, a Lexical Transducer (and transducers in general) can also be run backwards to do analysis, which is also not possible with **UNIX** utilities like **sort** and **uniq**.

Composing Transducers in *xfst* The *xfst* interface offers two ways to compose transducers:

1. In regular expressions, using the *.o.* operator, and
2. On The Stack, using the **compose net** command.

Suppose that we have previously compiled two transducers, a lexicon transducer defined with **lexc** and a rule transducer defined with **replace** rules, and have saved them to file as `lex.fst` and `rules.fst` respectively. Let us also suppose that we want to compose the rules under the lexicon. The following sequence of commands shows how to perform the composition using a regular expression. Recall that in **xfst** regular expressions, the at-sign operator interprets its argument as the filename of a pre-composed binary network and reads it in.

```
xfst[0]: read regex @"lex.fst" .o. @"rule.fst" ;
```

As in **UNIX** pipes, the first or leftmost argument is the “top” argument. We can also write a completely equivalent regular expression on multiple lines, which results in a more satisfactory visual representation of the lexicon transducer being composed on top of the rule transducer.

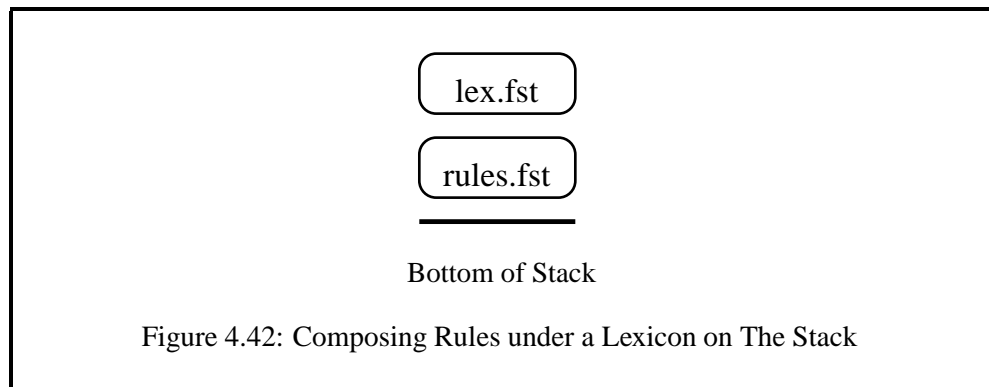
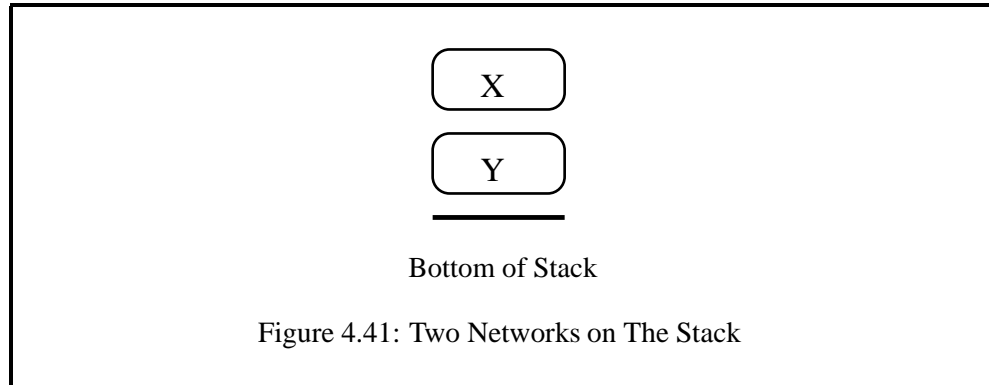
```
xfst[0]: read regex
@"lex.fst"
.o.
@"rule.fst" ;
```

If we happened to have multiple pre-compiled rule files `rule1.fst`, `rule2.fst` and `rule3.fst`, to be composed on the bottom of the lexicon, in that order, we could specify our regular expression as follows:

```
xfst[0]: read regex
@"lex.fst"
.o.
@"rule1.fst"
.o.
@"rule2.fst"
.o.
@"rule3.fst" ;
```

Performing the same composition on The Stack in **xfst** is one of the most consistently confusing operations for students. Stacks are last-in, first-out data structures, which means that if you push `Network1` onto The Stack, then `Network2` and then `Network3`, they will be popped off of The Stack in exactly the reverse order: first `Network3`, then `Network2` and finally `Network1`. The way that **xfst** pops off networks and assigns them as arguments to operations is not psychologically intuitive, and most students start out by building their stacks backwards. For un-ordered (commutative) operations, such as union and intersection, the order of the arguments on The Stack doesn't matter; but for ordered operations like **minus net** and **compose net**, correct ordering of arguments on The Stack is vital.

The **compose net** function, like all **xfst** stack-based operations, will pop its arguments off of The Stack one-by-one, compute the result, and push the result



back on The Stack. If nets X and Y are on The Stack as shown in Figure 4.41, and the user invokes **compose net**, first X will be popped off of The Stack to be assigned as the first argument of the composition, and then Y will be popped off The Stack to be assigned as the second argument. The result will be $[X \ .o.\ Y]$, with X being the first or top argument, and Y being composed under it.⁴ For success in performing ordered operations in **xfst**, try to visualize The Stack itself, as in Figure 4.41, where the arguments are ordered intuitively.

However, people find it difficult to visualize The Stack per se, and indeed The Stack is not particularly easy to visualize. Once a network is put on The Stack, it has no name or label; The Stack is a pile of anonymous networks. The problem is that to achieve the stack state shown in Figure 4.41, the user must first push the Y network onto The Stack, and then the X network. Users intuitively remember the order in which they push arguments on The Stack—and they tend not to visualize the resulting state of The Stack. So when users want to perform $[X \ .o.\ Y]$, their first reaction is almost always to push X on The Stack first and then Y, which for **xfst** is backwards.

⁴The **xfst** assignment of the first-popped network as the first or leftmost argument for the operation is the opposite of the convention used in Hewlett-Packard calculators and in most stack-based machines used by compilers and interpreters to evaluate expressions.

Let us assume again that we have two pre-compiled files, `lex.fst` and `rules.fst`, and that we need to compose the rules under the lexicon; i.e., we want to perform `[lex.fst .o. rules.fst]`. To achieve the stack state shown in Figure 4.42, the user must push `rules.fst` onto The Stack first and then `lex.fst`.

```
xfst[0]: load stack rules.fst
xfst[1]: load stack lex.fst
xfst[2]: compose net
```

The key in **xfst** stack-based operations is to visualize The Stack itself and make it look like the cascade of networks that need to be composed, with the upper argument(s) on top of The Stack. Because The Stack is so difficult to visualize, even experienced developers must often sketch their plans on paper before they can use The Stack reliably.

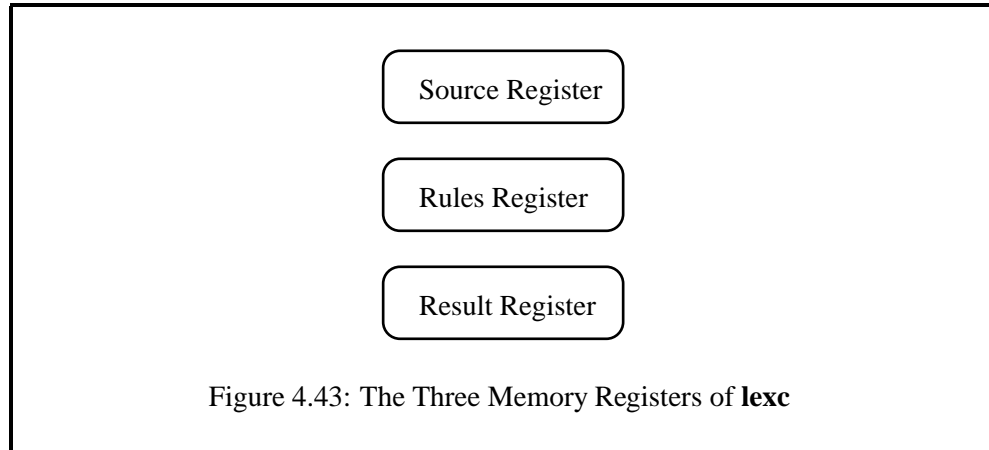
The ordering of arguments on The Stack in **xfst** is a trap waiting to catch the unwary. Beware.

Study stack-based composition until you are comfortable with it. An example below will ask you to write a fragment of an Irish lexicon, using **lexc**, and also replace rules to perform a necessary phonological/orthographical alternation. You will be asked to compose the resulting lexicon and rule transducers together first with an overt regular expression and then, equivalently, on The Stack.

Composing Lexicons and Rules in lexc The **lexc** interface does not use a stack but rather has three named registers: SOURCE, RULES and RESULT. The assumption in **lexc** is that linguists will use **lexc** to define the SOURCE (or lexicon) network, **twolc** or **xfst** to pre-compile (and save to binary file) the RULES network(s), and that the RULES network or networks are to be composed under the SOURCE network to compute the RESULT network. These basic assumptions reflect the geometry of early “Two-Level” morphology systems.

One typically uses the **compile-source** command to compile a **lexc** source file and put the result in the SOURCE register; if the file was previously compiled and saved as a binary file, it can be read back into the SOURCE register using **read-source**. The command **read-rules** can then be used to read in a binary file that was previously compiled by the **twolc** compiler or by **xfst**. The command **compose-result** then composes the rule network(s) (in the RULES register) under the lexicon network (in the SOURCE register) and places the resulting network in the RESULT register. The standard visualization of the three registers as in Figure 4.43, with the SOURCE above the RULES, is consistent with the order of the composition.

The Intersecting-Composition Algorithm The **compose-result** utility in **lexc** is based on a special algorithm called INTERSECTING-COMPOSITION, which is



different in some significant ways from the general composition algorithm used by the **compose net** utility in **xfst**. The differences between these two composition algorithms sometimes lead to confusion.

lexc was created before **xfst** Replace Rules, when **Xerox** linguists wrote two-level rules using the **twolc** rule compiler (see Chapter 5). Compilation of a set of two-level rules (see Section 5.2.4, Section 5.3.5) results initially in a set of separate networks, one for each rule, and these multiple networks can be saved to a single file via the **twolc** utility **save-binary** (Section 5.2.4).

The **lexc read-rules** command can read in such a binary file, containing multiple rule networks, and the intersecting-composition algorithm, invoked by **compose-result**, simultaneously intersects and composes the rule networks with the SOURCE network, putting the result in the RESULT register. The general composition algorithm used by the **xfst compose net** utility cannot simultaneously intersect and compose in this way.

The intersecting-composition algorithm invoked by the **lexc** utility **compose-result** can, as its name suggests, simultaneous intersect and compose a set of rule networks under the SOURCE network.

Another particular feature of intersecting-composition is that it assumes that any Flag Diacritics (Chapter 8) are contained only in the SOURCE network, i.e. that the rule networks contain no references to Flag Diacritics. The algorithm therefore allows rules to match lexical strings *ignoring Flag Diacritics*, and it also preserves any Flag Diacritics on the lower side of the SOURCE lexicon, projecting them onto the lower side of the RESULT lexicon. This is known as “treating Flag Diacritics as special”. **lexc** can treat Flag Diacritics as special because it assumes (fairly safely) that one is following the old two-level way of doing things.⁵ These

⁵If you have Flag Diacritics in both of two networks you are composing, and try to treat them as

features of intersecting-composition are hard for everyone to remember, especially as **twolc** rules are seldom used now.

Intersecting-composition is also perfectly capable of handling single rule networks compiled by **xfst** and single rule networks created in **twolc** by running **compile** and then **intersect**, which intersects the multiple rule networks into a single network, before saving to file with **save-binary**.

As already stated, the composition algorithm used by the **xfst compose net** utility is *not* intersecting-composition but rather general composition.

1. If you want to compose a set of rule networks from **twolc** using **xfst**, you must first pre-intersect them (using the **intersect** command of **twolc**) before saving them to file (**save-binary**).
2. **xfst** composition could be applied to any imaginable pair of networks, from who knows where. It cannot assume the two-level geometry as in **lexc**, where the SOURCE is assumed to be a lexicon network and the RULES are assumed to be compiled from **twolc** rules. Therefore it cannot assume that only one (at most) of the networks being composed contains Flag Diacritics. The default behavior of **xfst** composition is therefore to treat Flag Diacritics like any other multicharacter symbol; i.e. not special. Indeed, one can imagine cases where you want **xfst** rules to match on, map, and insert Flag Diacritics.
3. If you want Flag Diacritics to be treated as special in **xfst** (i.e. if you want **xfst** composition to handle Flag Diacritics the way **lexc** intersecting-composition does), then you need to set the following **xfst** variable before invoking **compose net**.

```
xfst[0]: set flag-is-epsilon ON
```

This variable is set OFF by default.

4.6.2 The check-all Utility

Introduction

The **check-all** utility in **lexc** is used to compare a network *before* composition with a set of rules to the result *after* composition of the rules. It helps you find what was gained and lost in the composition, which can be very helpful when debugging a complex set of rules.

In the **lexc** interface, the Before-FST is in the SOURCE register; it is typically compiled from a **lexc** source file using **compile-source**, but it could be any network read into the SOURCE register using **read-source**.

special, this leads to horrible mathematical problems.

In the **lexc** interface you read into the **RULES** register a pre-compiled (i.e. binary) file of rule transducers using the **read-rules** command, and then you invoke **compose-result**, which composes the **RULES** network on the bottom of the **SOURCE** network and puts the result in the **RESULT** register. So the network in the **RESULT** register is the After-FST. The overall plan is shown in Figure 4.44.

So after **compose-result** the **lexc** interface has the following networks to work with

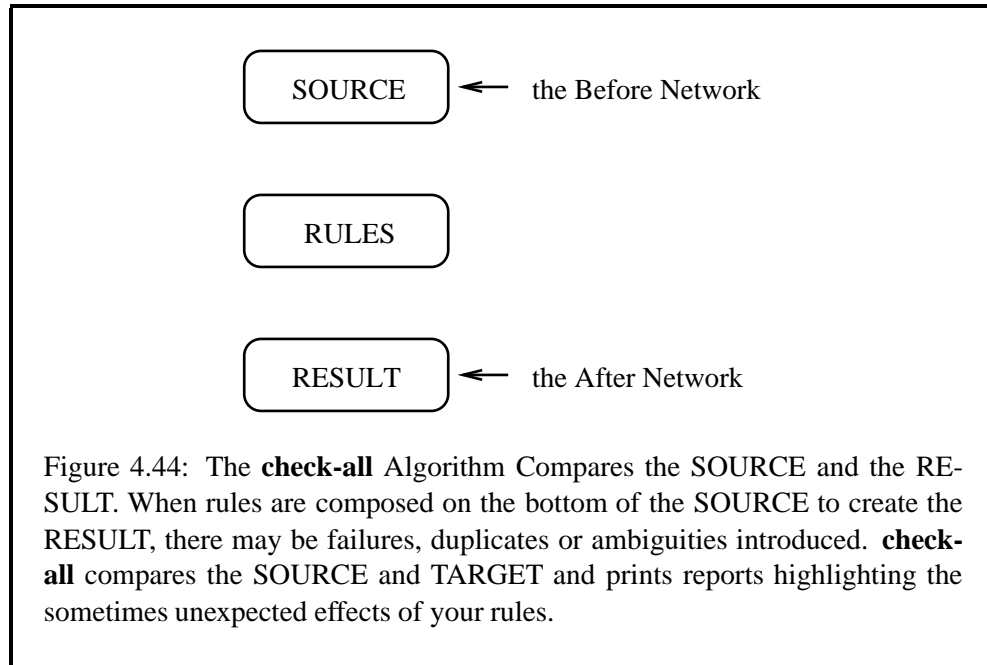


Figure 4.44: The **check-all** Algorithm Compares the **SOURCE** and the **RESULT**. When rules are composed on the bottom of the **SOURCE** to create the **RESULT**, there may be failures, duplicates or ambiguities introduced. **check-all** compares the **SOURCE** and **TARGET** and prints reports highlighting the sometimes unexpected effects of your rules.

check-all compares the **SOURCE** (Before) and **RESULT** (After) networks; it ignores the **RULES** network.

What Happens when Rules are Composed

Usually, and especially when we are working with a traditional two-level geometry where we have a single lexicon and a single set of rules to be composed on the lower side of the lexicon, we think of applying rules to modify the lower-side strings in the lexicon network. But a lot of things can happen during the composition of a set of rules, perhaps not all of them expected or desirable. In particular,

1. The effect of composing some rules may be to delete whole paths that existed in the Before-FST. This is especially easy when composing a rule like $a \rightarrow \sim [? *]$, which kills any path wherein the lower-side string contains the symbol **a**. This may be your intent (which is fine), but if you weren't expecting any paths to be deleted, then you might well want **check-all** to identify the paths that failed to survive the composition.

Switch	Default	Display
failures	ON	Source paths lost in the composition
duplicates	ON	Lexical strings in the result with multiple outputs
ambiguities	OFF	Surface strings in the result with multiple analyses
singles	OFF	Lexical strings in the result with single outputs

Figure 4.45: Switches that Control **check-all** Output

2. The effect of composing some rules may be to add new paths to the result. This is especially easy with rules like $a \rightarrow [a | b]$, which will tend to add a new RESULT path for every SOURCE path that has a string on the lower side containing a symbol **a**. The added paths will, of course, have a similar string on the lower side but with **b** instead of **a**. If that was your intent, then everything is fine. But if your rules are adding unexpected new paths to the RESULT, then you will want **check-all** to help you identify those new paths.

Output from check-all

The **check-all** utility always operates in the same way, counting phenomena known as SINGLES, FAILURES, DUPLICATES and AMBIGUITIES, and displaying a small summary table of the results. In addition, verbose output of the examples of each phenomenon is user-controlled with the switches shown in Figure 4.45.

The underlying **check-all** algorithm starts by extracting and determinizing a network representing the upper-side language of the SOURCE network (the Before-FSM). Any loops are removed. This network of strings, let us call it the upper-source network, is then compared with the RESULT network.

FAILURES are cases where a string in the upper-source network does not exist on the upper side of the RESULT network. These failures correspond to paths that were eliminated in the process of composing the RULES network. As developers are usually interested in seeing such cases, the **failures** toggle is appropriately ON by default, causing a listing of all upper-source strings that have no realization in the RESULT network.

DUPLICATES are cases where a string in the upper-source network exists on the upper side of the RESULT network and has more than one surface realization in the RESULT network. In most natural-language orthographies, such duplicates are rare, e.g. the French word for “I pay” can be spelled either *paye* or *paie*. The **duplicates** toggle is appropriately ON by default, assuming that the user wants to see a listing of such duplicates.⁶ **check-all** finds duplicates in the RESULT

⁶The exceptional cases would be like Arabic, where in the standard orthography short vowels and other diacritics are only optionally written in surface words, and so every lexical string in the final network has duplicates.

network, but it does not currently distinguish between duplicates introduced via the rule composition and duplicates that already existed in the SOURCE network.

AMBIGUITIES are cases where, in the RESULT network, a single surface string is related to two or more lexical strings. Ambiguities are therefore the opposite of duplicates, but the ambiguity of surface forms in natural-language orthography is common, often pervasive, and the **ambiguities** toggle is appropriate OFF by default, preventing a listing. **check-all** does not currently distinguish between ambiguities introduced via the rule composition and ambiguities that already existed in the SOURCE network.

SINGLES are cases where a string in the upper-source network exists on the upper side of the RESULT network and has a single surface realization in the RESULT network. The **singles** toggle is appropriately OFF by default, preventing the listing of such cases, which are usually overwhelming in number and uninteresting anyway.

Switch Settings The default switch settings therefore assume that the user wants to see a verbose listing of failures and duplicates, but not of ambiguities or singles. To see the current setting of all the switches, use the **props** command.

```
lexc> props
```

The switches can of course be toggled to match the needs of atypical cases, e.g. simply enter **ambiguities** to the **lexc** interface to toggle the switch ON or OFF. **lexc** will print a message indicating the new value:

```
lexc> ambiguities
(check-all) Ambiguities is ON.
```

Using **check-all** after Composition in **xfst**

Although **check-all** was originally intended for checking the RESULT network after **compose-result**, you can still use **check-all** even if you performed your composition in **xfst**. Assume that `lexicon.fst` is a file containing a binary network representing a lexicon, and that `rules.fst` is a file containing another binary network from compiled rules. Inside **xfst** we might compose `rules.fst` under `lexicon.fst`, writing the result of the composition out to file `result.fst`.

```
xfst[n]: clear stack
xfst[0]: read regex @"lexicon.fst" .o. @"rules.fst" ;
xfst[1]: save stack result.fst
xfst[1]: quit
```

We can then call up the **lexc** interface and do the following:

```
lexc> read-source lexicon.fst
lexc> read-result result.fst
lexc> check-all
```

So here we simply read the Before-FST (`lexicon.fst`) into the SOURCE register, read the After-FST (`result.fst`) into the RESULT register, and call **check-all** as usual.

Finding Ambiguities and/or Duplicates in a Single FST

After you have created, by whatever means, a single network and stored it in a binary file `network.fst`, it may be useful to know if it contains ambiguities and/or duplicates in the **check-all** sense.

An easy way to test this is to do the following in **lexc**:

```
lexc> read-source network.fst
lexc> source-to-result
lexc> check-all
```

Because the SOURCE network is simply copied into the RESULT register, there will obviously be no failures. But the listing of duplicates (the **duplicates** switch is automatically initialized to ON) will be done by default. If your `network.fst` represents a morphological analyzer for French, it should contain rare duplicate realizations like *paye* and *paie*, and **check-all** should enumerate such cases for you.

To check the network for ambiguities, you will have to toggle the **ambiguities** switch ON before invoking `check-all`, e.g.

```
lexc> read-source network.fst
lexc> source-to-result
lexc> ambiguities
(check-all) Ambiguities is ON.
lexc> check-all
```

Whenever a switch is toggled, as in this case, **lexc** responds with a short message informing you of the new value. To see the current setting of all the **lexc** switches, enter the **props** command or the **status** command.

```
lexc> status
SOURCE: (none)
RULES: (none)
RESULT: (none)
Switches: singles=OFF, duplicates=ON, failures=ON,
          ambiguities=OFF, obey-flags=ON, print-space=OFF,
          quit-on-fail=OFF, show-flags=OFF, time=OFF,
          verbose=OFF, completion=OFF
```

Basic	Lenited	Pronunciation
b	bh	/v/
m	mh	/v/
p	ph	/f/
f	fh	(silent)
s	sh	/h/
t	th	/h/
c	ch	/x/
d	dh	/ɣ/
g	gh	/ɣ/

Figure 4.46: Lenited Consonants in Irish Gaelic

4.6.3 Exercises

Irish Gaelic Lenition

Background The following *Gaeilge* (Irish Gaelic) example was kindly supplied by Donncha O’Croinin of the Institiúid Teangeolaíochta Éireann (The Linguistics Institute of Ireland). As usual, we have limited and simplified the data to make a manageable exercise, which will require both **lexc** and some replace rules written in a separate file and compiled with **xfst**.

Gaeilge has an interesting phonological alternation, shared with Scottish Gaelic and Welsh, that occurs at the beginning of certain words, an alternation called consonant lenition. We will model a small fragment of the data that includes positive and negative verbs.

The Data The following description will be taken as the data to be modeled:

1. The alphabet of Irish Gaelic is the following (we will deal only in lower-case letters):

a á b c d e é f g h i í l m n o ó p r s t u ú v

2. Many of the consonants have lenited forms, indicated *orthographically* by adding an *h* letter after the consonant. The lenited form of *b* is therefore spelled *bh*. The lenitable consonants, their lenited spellings, and their phonological values are shown in Figure 4.46.

As usual in text-based natural-language processing, our goal will be to capture the orthographical change that reflects the underlying phonological lenition.

3. The Irish Gaelic verb roots for our example are *rith*, *caith* and *bris*. These verbs can take various suffixes:

Infinitive:	(empty suffix)
Past:	(empty suffix)
Present:	eann
Future:	fidh
Conditional:	feadh

bris, *briseann*, *brisfidh* and *brisfeadh* are valid words.

4. Negative forms of these words (i.e. the same verbs used in negative syntactic constructions) are subject to initial-consonant lenition; e.g. the negative form of *briseann* is *bhriseann*. (Not all consonants undergo lenition; for example, *r* does not, so there is no form **rhith* as a form of *rith*.)

The Task

- Write a **lexc** grammar to generate lexical forms and a small grammar of replace rules to handle the orthographical alternation that reflects the lenition.
- Write the **lexc** grammar to produce a transducer, with baseforms and tags on top and intermediate strings on the bottom. The rules will then map these intermediate strings into the final surface strings.
- Define and use the following multicharacter symbols in your **lexc** grammar: +Verb, +Neg (i.e. negative), +Past, +Pres (i.e. present), +Fut (i.e. future), +Inf (i.e. infinitive) and +Cond (i.e. conditional). There's nothing sacred about these tags, but they have the advantages of being short and readable, and the same tags are already used (with the same meanings) in existing **Xerox** products.
- Using **lexc** LEXICONS and continuations, create a transducer that accepts string pairs like the following. Write the grammar so that the +Neg tag appears on both sides of the lexicon transducer while other tags appear only on the upper side. Symbol pairs are roughly lined up in these examples.

Lexical:	bris+Verb+Pres
Surface:	bris eann

Lexical:	bris+Verb+Pres+Neg
Surface:	bris eann +Neg

- Open a new file with your text editor and write replace rules that map negative strings like “briseann+Neg” into “bhriseann”, and then mapping the +Neg symbol to zero on the surface. Make the rules general enough to handle all the data cited above. The first rule should look something like this:

```
[...] -> h | | .#. [ b | m | p | f | s | t | c | d | g ]
      _ ?* %+Neg
```

- Compile the rules using the **read regex** utility of **xfst** and save the result in a file named something like `irish-rul.fst`. Then, in **lexc**, compile the **lexc** file using **compile-source** as usual; save the result in `irish-lex.fst`. Enter **xfst** again, read the rule **FST** onto the stack, then read the lexicon **FST** onto The Stack (this will put the lexicon **FST** above the rule **FST**) and invoke the **compose net** utility. The result left on The Stack should be a single transducer that maps directly from “bris+Verb+Pres” to “briseann” and, more interestingly, from “bris+Verb+Pres+Neg” to “bhriseann”.
- Test the Result using **lookdown** and **lookup** inside **lexc**, or using **apply up** and **apply down** in **xfst**.

4.7 **lexc** Summary and Usage Notes

4.7.1 Mindtuning

Input

lexc input is a plain ASCII text file created using **vi**, **xemacs** or a similar text editor.

Output

The **lexc** compiler produces as output a finite-state transducer (**FST**) in **Xerox** standard format. This format is completely compatible with the networks that are produced and read by **xfst** and **twolc**.

lexc Interface

lexc is accessed via the **lexc** interface. In addition to compiling **lexc** text files, this interface offers the ability to compose a lexical **FST** with rule **FST**s pre-compiled by **xfst** or **twolc**. The interface also offers some useful testing facilities including **lookup**, **lookdown**, **check-all**, **random-surf**, **random-lex** and a few other helpful routines for manipulating finite-state machines.

Avoiding the lexc Interface

If you just need to compile **lexc** source files for subsequent use in **xfst**, you can compile them directly from **xfst** using the **read lexc** command. If your **lexc** source file is `navajo-lex.txt`, the following commands will compile it and place the resulting network on the **xfst** stack.

```
xfst[0]: read lexc < navajo-lex.txt
xfst[1]:
```

Right-Recursive Phrase-Structure Grammar

The **lexc** grammar, summarized below, is a kind of right-recursive phrase-structure grammar. There are other languages for specifying **FSTs**, but **lexc** is specially designed to facilitate the creation of finite-state lexicons for natural language, where one typically needs to enter thousands of strings representing roots and affixes. Compared to alternatives like **xfst**, **lexc** syntax facilitates the typing in of typical lexical entries, and the **lexc** compiler is optimized to perform large lexical unions efficiently.

4.7.2 lexc Syntax**Multichar_Symbols Declaration**

The first (optional) statement in a **lexc** source file is the declaration of multicharacter symbols, separated by spaces. The declaration may extend over multiple lines, and there is no semicolon or other overt terminator.

```
Multichar_Symbols +Noun +Verb +Adj +Sg +Pl
                  +Bare +3PSg
```

The failure to declare all the multicharacter symbols intended in the grammar is a common error.

Comments

Comment can appear anywhere and are introduced by an exclamation mark (!). Everything from the exclamation mark to the end of the line is considered a comment and is ignored by the compiler.

```
! this is a lexc comment--use lots of them
```

LEXICONS

LEXICON Root The body of a **lexc** source file consists of one or more named LEXICONS, each one containing one or more entry lines. The LEXICON with the reserved name `Root` corresponds to the start state of the resulting network.

```

LEXICON Root
dog          N ;
cat          N ;
bird         N ;
sing         V ;
prevaricate V ;

```

If no LEXICON Root is present, then **lexc** will try to use the first LEXICON in the file as the start state.

User-Named LEXICONS Each additional LEXICON must have a unique name chosen by the user, such as N or V.

```

LEXICON N
+Sg:0      # ;
+Pl:s      # ;

LEXICON V
+Bare:0    # ;
+3PSg:0    # ;

```

Entries

Entry Fields Entry lines inside a LEXICON are always built on the following abstract template.

```

Data          ContinuationClass ;

```

The first field consists of the data itself; the second field consists of the name of a CONTINUATION CLASS; and a semicolon terminates the entry.

The continuation class is the name of another sublexicon to which the given data can “continue” to form legal words. The pound-sign (#) is a special continuation class designating the end of word.

```

Data          # ;

```

Data Formats

Simple String Entries The data field can appear in one of four formats. The first is the simplest, consisting of a string of characters that often looks like a normal word or morpheme. Here is a small LEXICON named Nouns with three simple-string entries.

LEXICON Nouns

elephant	Noun ;
dog	Noun ;
book	Noun ;

lexc assumes that a string of data characters written as *dog* is to be interpreted as three separate symbols, and it compiles *dog* as the network shown in Figure 4.47.

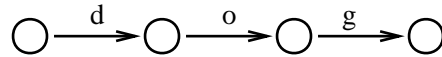


Figure 4.47: The Network Compiled from the Data *dog*

lexc's assumption that entry strings like *dog* are to be “exploded” into separate symbols is very useful for specifying natural-language lexicons, where the lexicographer must often type in tens of thousands of such strings.⁷

Upper:Lower Entries The second format for **lexc** data in an entry is again designed with natural languages in mind. The *upper:lower* format indicates a string of upper or lexical characters which are to be mapped to a string of lower or surface characters, and the two strings are separated by a colon.

man:men	Noun ;
child:children	Noun ;

N.B. that the format is *upper:lower*, with the upper-level string on the left-hand side of the colon. **lexc** by default again “explodes” each string and compiles this notation into **FSTs** as shown in Figure 4.48.

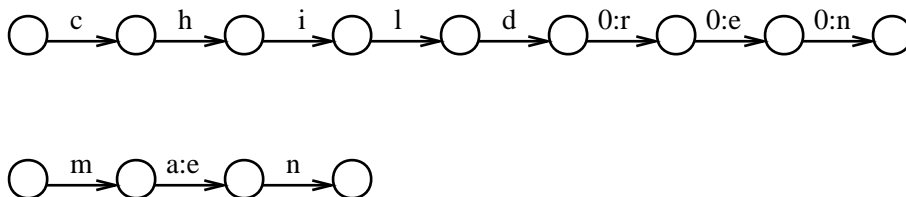


Figure 4.48: The Networks Compiled from *child:children* and *man:men*

⁷In **xfst**, strings of symbols must be typed with the letters separated by spaces, or surrounded by curly braces; you could create a large natural-language lexicon in **xfst**, but it would be tedious.

When the two strings are of unequal length, as in `child:children`, the compiler automatically fills out the end of the shorter string with zeros (epsilons). This `upper:lower` notation is especially useful for handling suppletions and other gross irregularities.

Regular-Expression Entries The third legal format for **lexc** data is a regular expression. When this format is used, the regular expression must be delimited with angle brackets. The angle brackets are the signal to the **lexc** compiler that it should compile the contained expression according to the normal assumptions for **xfst** regular expressions—and not according to the normal assumptions for **lexc**.

```
< a b* c+ (d) >          N ;
```

Note that inside angle brackets, all the usual assumptions about regular expressions apply, including the fact that the string *dog*, intended to be three separate symbols, would have to be represented inside angle brackets as `[d o g]` or `{dog}`. If you write the string `dog` inside angle brackets, without spaces or surrounding brackets, it will be interpreted as a single multi-character symbol, just as it would be inside **twolc** or inside an **xfst** regular expression.

Empty-Data Entries Finally, note that the *Data* part of a **lexc** entry can be empty, in which case the entry adds no symbols but simply indicates a possible continuation to the named `ContinuationClass`.

```
LEXICON Foo
      ContinuationClass ; ! an empty-data entry
```

Empty entries can be used to implement optionality, and they are commonly seen under *LEXICON Root* to indicate an initial branching to various major word-class LEXICONS, e.g.

```
LEXICON Root
      Nouns ;
      Verbs ;
      Adjectives ;
      Adverbs ;
      Conjunctions ;
      TimeStrings ;
      CurrencyString ;
      RomanNumerals ;
```

4.7.3 Special Attention

Special Characters

The special characters are different in **xfst** and **lexc**, and developers switching from one language to the other often falls into the cracks.

Because **lexc** is designed primarily for typing strings of characters that occur in natural language, the general assumption is that one will *not* need regular expressions and all the special characters associated with them. Thus, outside of angle brackets, **lexc** assumes that plus signs (+), stars (*), circumflexes (^) and most other characters have their literal value.

Outside of the angle brackets, very few characters are special to **lexc**, namely angle brackets themselves, colons, semicolons, pound signs and exclamation marks; see Table 4.1. To literalize them, use a preceding percent sign: %<, %>, %#, %;, %:, %!.

Special Character	Why It's Special in lexc
<	Introduces regular-expression data
>	Terminates regular-expression data
#	A special continuation class designating end-of-word
;	Terminates an entry
:	Separates <i>upper:lower</i> data in entries
!	Introduces comments

Table 4.1: Special Characters in **lexc**, Even Outside Angle Brackets. To literalize these characters, precede them with a percent sign.

Inside angle brackets, which allow you to specify regular expressions within **lexc**, the normal assumptions of **xfst** regular expressions apply:

- In regular expressions, and inside angle brackets in **lexc**, strings of characters written together are automatically assumed to represent a single symbol. E.g.

```
< a b quark >
```

specifies the concatenation of three symbols, with the third being a multi-character symbol spelled *quark*. It is almost always a Bad Thing to have multicharacter symbols like **quark** that are easily confused with concatenations of normal alphabetic symbols.

- In regular expressions, and inside **lexc** angle brackets, to specify the **FST** that looks like Figure 4.47 you would have to specify

< d o g >

or

< {dog} >

If you instead write < *dog* >, it will be compiled as shown in Figure 4.49.

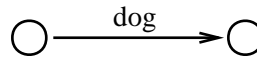


Figure 4.49: The Network Compiled from the **lexc** Regular-Expression Entry <dog>

- In **xfst** regular expressions, and inside **lexc** angle brackets, plus signs, stars and other regular-expression symbols are special symbols. If you want to indicate a literal plus sign, for example, you must unspecialize it with a preceding percent sign (%).

< %+Noun:o 0:n 0:d >

Explosion vs. Multicharacter Symbols

lexc is designed to facilitate the building of finite-state lexicons for natural language, and its design is based on the assumption that the lexicographer will be typing a lot of strings like *elephant* that should be exploded into individual symbols.

```

LEXICON NRoot
antelope    N ;
elephant    N ;
horse       N ;
raccoon     N ;
...
zebra       N ;
  
```

Declared `Multichar_Symbols` override the explosion assumption.

```
Multichar_Symbols +Noun +Verb +Sg +Pl +Sg ^FEATURE
```

Given such a declaration, whenever **lexc** sees the string `+Noun`, it will *not* explode it but should instead treat it as a single symbol. Thus the entry

```
+Noun:ond                N ;
```

will be compiled as in Figure 4.50.

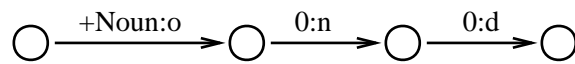


Figure 4.50: The Network Compiled from `+Noun : ond` if `+Noun` is Defined as a Multicharacter Symbol

Failure to Declare Multicharacter Symbols

The failure to declare a multicharacter symbol is a very common error in **lexc** programming. For example, if you start using a tag like `+Adv` (for Adverbs) but forget to declare it in the `Multichar_Symbols` statement, **lexc** will happily and silently explode it into four separate symbols, `+`, `A`, `d` and `v`. See Section 4.5.6 for information on when to suspect and how to find such omissions. See also Chapter 7, which is devoted to testing.

Chapter 5

The twolc Language

Contents

5.1	Introduction to twolc	308
5.1.1	History	309
	The origins of finite-state morphology	309
	Two-Level Morphology	312
	Linguistic Issues	316
	Two-Level Rule Compilers	317
	Implementations of Two-Level Morphology	318
5.1.2	Visualizing twolc Rules	319
5.1.3	Plan of Attack	321
5.2	Basic twolc Syntax	321
5.2.1	Alphabet	322
5.2.2	Basic Rules	323
	Rule Syntax	323
	twolc Rule Operators	324
	twolc Rule Contexts	325
5.2.3	Thinking in twolc	329
5.2.4	Basic twolc Interface Commands	331
5.2.5	Exercises	333
	The kaNpat Exercise	333
	Brazilian Portuguese Pronunciation	334
5.3	Full twolc Syntax	336
5.3.1	Header Sections	336
	Sets	336
	Definitions	337
	Diacritics	337
5.3.2	Full twolc Rule Syntax	337

	Multiple Contexts	337
	Rule-Scoped Variables	337
5.3.3	Full twolc Interface	339
	Utilities for Rule Testing and Intersection	339
	Miscellaneous twolc Utilities	342
5.3.4	Exercises	343
	Monish Vowel Harmony	343
5.3.5	Understanding twolc and Networks	344
5.4	The Art and Craft of Writing twolc Grammars	346
5.4.1	Using Left-Arrow and Right-Arrow Rules	346
5.4.2	Multiples Levels of Two-Level Rules	347
5.4.3	Moving Tags	350
	Tag-Moving Exercise 1	351
	Tag-Moving Exercise 2	351
	Tag-Moving Exercise 3	351
5.5	Debugging twolc Rules	352
5.5.1	Rule Clashes	352
	Right-Arrow Conflicts	352
	Left-Arrow Rule Conflicts	353
	Spurious Compiler Warnings	360
	Why Context Subtraction Works	361
5.5.2	Epenthesis Rules	362
5.5.3	Diacritics	364
	The Raw Facts about Diacritics	364
	The Motivation for Diacritics	365
	Noticing and Ignoring Diacritics	365
	Diacritics and Rule Conflicts	366
5.5.4	Classic twolc Errors	366
5.6	Final Reflections on Two-Level Rules	369
5.6.1	Upward-Oriented Two-Level Rules	369
5.6.2	Comparing Sequential and Parallel Approaches	370

5.1 Introduction to twolc

twolc, for **Two-Level Compiler**, is a high-level language for describing morphological alternations as in *fly:flies*, *swim:swimming* and *wiggle:wiggling*. The **twolc** syntax is based on the declarative system of rule constraints, known as TWO-LEVEL RULES, proposed in Kimmo Koskenniemi's 1983 dissertation (Koskenniemi, 1983; Koskenniemi, 1984).

Like the replace rules described in Section 3.5.2, **twolc** rules denote regular relations. But **twolc** rules have a distinct syntax and semantics, and they require the linguist to adopt a different mindset and grammar-writing approach. This chapter describes the syntax of **twolc** source files, the **twolc** compiler interface, and the use of the results within finite-state systems.

At **Xerox**, where developers have a choice, they are increasingly avoiding **twolc** and using replace rules. For the time being, some developers will have to learn **twolc** as well as replace rules, especially those who need to support legacy systems.

twolc rules are always written in a file using a text editor like **xemacs**, **emacs** or **vi**. The **twolc** compiler interface, written in **C**, provides commands for the syntactic checking of source files, compilation, testing, and writing of the results to file. Two distinct formats of output can be generated: 1) standard **Xerox** binary format suitable for composition, inside **lexc**, with lexicons and other transducers produced with the **Xerox** finite-state tools, and 2) TABULAR format, suitable for use by traditional two-level or “KIMMO” systems, in particular those written with the Lingsoft **TwoL** implementation and Evan Antworth’s **PC-KIMMO** (Antworth, 1990), which is available from the Summer Institute of Linguistics (SIL).¹

The introduction will continue with a bit of history, laying the groundwork for understanding why **twolc** rules were invented and how they work. Special attention will be paid to visualizing how **twolc** rules can fit into an overall grammar.

5.1.1 History

The origins of finite-state morphology

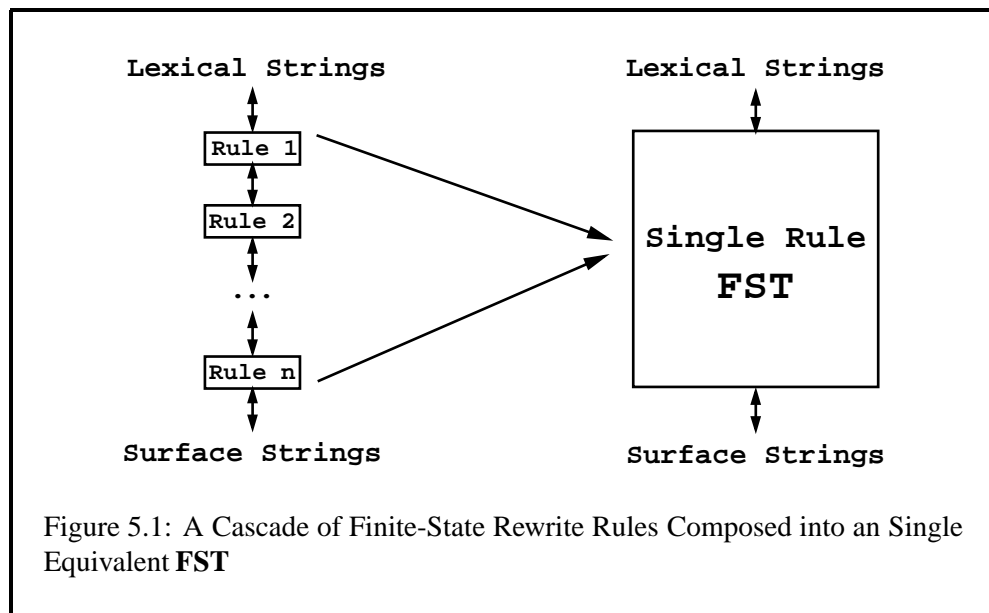
Traditional phonological grammars, formalized in the 1960s by Noam Chomsky and Morris Halle (Chomsky and Halle, 1968), consisted of an ordered sequence of REWRITE RULES that converted abstract phonological representations into surface forms through a series of intermediate representations. Such rewrite rules have the general form $\alpha \rightarrow \beta/\gamma _ \delta$ where α , β , γ , and δ can be arbitrarily complex strings or feature-matrices. The rule is read “ α is rewritten as β in the environment between γ and δ ”. In mathematical linguistics (Partee et al., 1993), such rules are called CONTEXT-SENSITIVE REWRITE RULES, and they are more powerful than regular expressions or context-free rewrite rules.

In 1972, C. Douglas Johnson published his dissertation, *Formal Aspects of Phonological Description* (Johnson, 1972), wherein he showed that phonological rewrite rules are actually much less powerful than the notation suggests. Johnson observed that while the same context-sensitive rule could be applied several times recursively to its own output, phonologists have always assumed implicitly that the site of application moves to the right or to the left of the string after each application. For example, if the rule $\alpha \rightarrow \beta/\gamma _ \delta$ is used to rewrite the string $\gamma\alpha\delta$ as $\gamma\beta\delta$, any subsequent application of the same rule must leave the β part

¹<http://www.sil.org/computing/catalog/pc-kimmo.html>

unchanged, affecting only γ or δ . Johnson demonstrated that the effect of this constraint is that the pairs of inputs and outputs produced by a phonological rewrite rule can be modeled by a finite-state transducer. Unfortunately, this result was largely overlooked at the time and was rediscovered by Ronald M. Kaplan and Martin Kay around 1980 (Kaplan and Kay, 1981; Kaplan and Kay, 1994). Putting things into a more algebraic perspective than Johnson, Kaplan and Kay showed that phonological rewrite rules describe REGULAR RELATIONS. By definition, a regular relation can be represented by a finite-state transducer.

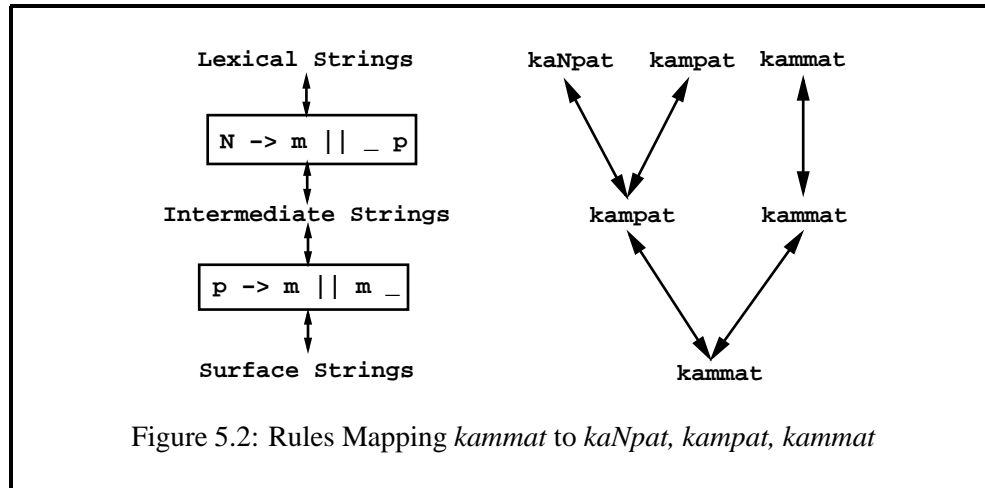
Johnson was already aware of an important mathematical property of finite-state transducers (Schützenberger, 1961): there exists, for any pair of transducers applied sequentially, an equivalent single transducer. Any cascade of rule transducers can in principle be composed into a single transducer that maps lexical forms directly into the corresponding surface forms, and vice versa, without any intermediate representations. Later, Kaplan and Kay had the same idea, illustrated in Figure 5.1.



These theoretical insights did not immediately lead to practical results. The development of a compiler for rewrite rules turned out to be a very complex task. It became clear that building a compiler required as a first step a complete implementation of basic finite-state operations such as union, intersection, complementation, and composition. Developing a complete finite-state calculus was a challenge in itself on the computers that were available at the time.

Another reason for the slow progress may have been that there were persistent doubts about the practicality of the approach for morphological *analysis*. Traditional phonological rewrite rules describe the correspondence between lexical forms and surface forms as a uni-directional, sequential mapping from lexical

forms to surface forms. Even if it was possible to model the *generation* of surface forms efficiently by means of finite-state transducers, it was not evident that it would lead to an efficient analysis procedure going in the reverse direction, from surface forms to lexical forms.



The *kaNpat* exercise (see Section 3.5.3) is a simple illustration of the problem. The transducers compiled from the two *xfst* replace rules,

$N \rightarrow m \mid \mid _ p$

and

$p \rightarrow m \mid \mid m _$

map the lexical form “*kaNpat*” unambiguously down to “*kammat*”, with “*kampak*” as the intermediate representation (see Figure 3.13). However if we apply the same transducers in the upward direction to the input “*kammat*”, we get the three results “*kaNpat*”, “*kampak*” and “*kammat*” shown in Figure 5.2. The reason is that the surface form “*kammat*” has two potential sources on the intermediate level; the downward application of the $p \rightarrow m \mid \mid m _$ rule maps both “*kampak*” and “*kammat*” to the same surface form. The intermediate form “*kampak*” in turn could come either from “*kampak*” or from “*kaNpat*” by the downward application of the $N \rightarrow m \mid \mid _ p$ rule. The two rule transducers are unambiguous when applied in the downward direction but ambiguous when applied in the upward direction.

This asymmetry is an inherent property of the generative approach to phonological description. If all the rules are deterministic and obligatory and if the order of the rules is fixed, then each lexical form generates only one surface form. But a surface form can typically be generated in more than one way, and the number of possible analyses grows with the number of rules involved. Some of the analyses may eventually turn out to be invalid because the putative lexical forms, say

“kammät” and “kampät” in this case, might not exist in the language. But in order to look them up in the lexicon, the system first had to complete the analysis, producing all the phonological possibilities. The lexicon was assumed to be a separate module that was used subsequently to accept or reject the possible analyses. Depending on the number of rules involved, a surface form could easily have dozens or hundreds of potential lexical forms, even an infinite number in the case of certain deletion rules.

Although the generation problem had been solved by Johnson, Kaplan and Kay, at least in principle, the problem of efficient morphological analysis in the Chomsky-Halle paradigm was still seen as a formidable challenge. As counterintuitive as it was from a psycholinguistic point of view, it appeared that analysis was much harder computationally than generation. Composing all the rule transducers into a single one would not solve the “over-analysis” problem. Because the resulting single transducer is equivalent to the original cascade, the ambiguity remains.

The solution to the over-analysis problem should have been obvious: to formalize the lexicon itself as a finite-state transducer and compose the lexicon with the rules. With the lexicon included in the composition, all the spurious ambiguities produced by the rules are eliminated at compile time. The runtime analysis becomes more efficient because the resulting single transducer contains only lexical forms that actually exist in the language.

The idea of composing the lexicon and the rules together is not mentioned in Johnson’s book or in the early **Xerox** work. Although there obviously had to be some interface relating a lexicon component to a rule component, these were traditionally thought of as different types of objects. Furthermore, rewrite rules were seen as applying to individual word forms; the idea of applying them simultaneously to a lexicon as a whole required a new mindset and computational tools that were not yet available.

The observation that a single finite-state transducer could encode the inventory of valid lexical forms as well as the mapping from lexical forms to surface forms took a while to emerge. When it first appeared in print (Karttunen et al., 1992), it was not in connection with traditional rewrite rules but with an entirely different finite-state formalism that had been introduced in the meantime, Kimmo Koskenniemi’s TWO-LEVEL RULES (Koskenniemi, 1983).

Two-Level Morphology

In the spring of 1981 when Kimmo Koskenniemi came to the USA for a visit, he learned about Kaplan and Kay’s finite-state discovery. (They weren’t then aware of Johnson’s 1972 publication.) **Xerox** had begun work on the finite-state algorithms, but they would prove to be many years in the making. Koskenniemi was not convinced that efficient morphological analysis would ever be practical with generative rules, even if they were compiled into finite-state transducers. Some other way to use finite automata might be more efficient.

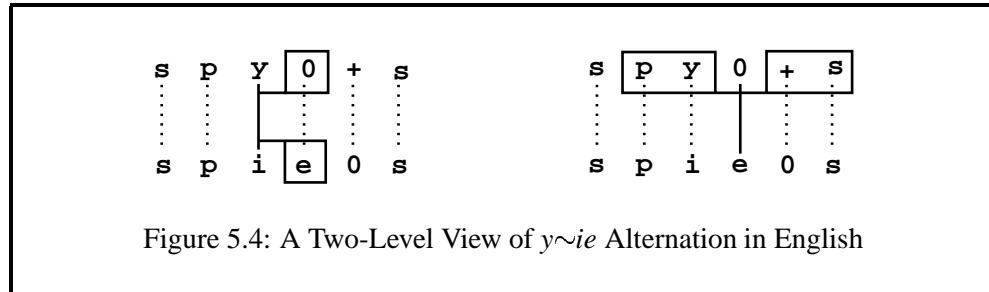


Figure 5.3: An Example of Two-Level Constraints

Back in Finland, Koskenniemi invented a new way to describe phonological alternations in finite-state terms. Instead of cascaded rules with intermediate stages and the computational problems they seemed to lead to, rules could be thought of as statements that directly constrain the surface realization of lexical strings. Multiple rules would be applied not sequentially but in parallel. Each rule would constrain a certain lexical/surface correspondence and the environment in which the correspondence was allowed, required, or prohibited. For his 1983 dissertation, Koskenniemi constructed an ingenious implementation of his constraint-based model that did not depend on a rule compiler, composition or any other finite-state algorithm, and he called it TWO-LEVEL MORPHOLOGY. Two-level morphology is based on three ideas:

- Rules are symbol-to-symbol constraints that are applied in parallel, not sequentially like rewrite rules.
- The constraints can refer to the lexical context, to the surface context, or to both contexts at the same time.
- Lexical lookup and morphological analysis are performed in tandem.

To illustrate the first two principles we can turn back to the *kaNpat* example again. A two-level description of the lexical-surface relation is sketched in Figure 5.3. As the lines indicate, each symbol in the lexical string “kaNpat” is paired with its realization in the surface string “kammaat”. Two of the symbol pairs in Figure 5.3 are constrained by the context marked by the associated box. The **N:m** pair is *restricted* to the environment having an immediately following **p** on the lexical side. In fact the constraint is tighter. In this context, all other possible realizations of a lexical **N** are *prohibited*. Similarly, the **p:m** pair requires the preceding surface **m**, and no other realization of **p** is allowed here. The two constraints are independent of each other. Acting in parallel, they have the same effect as the cascade of the two rewrite rules in Figure 5.2. In Koskenniemi’s notation, these rules are written as $N:m \Leftrightarrow _ p:$ and $p:m \Leftrightarrow :m _$, where \Leftrightarrow is an operator that combines a context restriction with the prohibition of any other realization for the lexical symbol of the pair. The **p** followed by a colon in the right context of first rule, $p:$, indicates that **p** refers to a lexical symbol; the colon preceding **m** in the left context of the second rule, $:m$, indicates that **m** is a surface symbol.



Two-level rules may refer to both sides of the context at the same time. The $y \sim ie$ alternation in English plural nouns could be described by two rules: one realizes **y** as **i** in front of an epenthetic **e**; the other inserts an epenthetic **e** between a lexical consonant-**y** sequence and a morpheme boundary (+) that is followed by an **s**. Figure 5.4 illustrates the **y:i** and **0:e** constraints.

Note that the **e** in Figure 5.4 is paired with a **0** (= zero) on the lexical level. Formally the rules are expressed as $y:i \Leftrightarrow _ 0:e$ and $0:e \Leftrightarrow y: _ \%+:$. From the point of view of two-level rules, zero is a symbol like any other; it can be used to constrain the realization of other symbols. In fact, all the other rules must “know” where zeros may occur. In two-level rules, the zeros are not epsilons, even though they are treated as such when two-level rules are eventually applied to strings.

Like replace rules, two-level rules describe regular relations; but there is an important difference. Because the zeros in two-level rules are in fact ordinary symbols, a two-level rule represents an *equal-length relation* (see Section 2.3). Counting the “hard zeros”, each string and its related strings always have exactly the same length. This has an important consequence: although transducers cannot in general be intersected (see Section 2.3.3), equal-length transducers are a special case, and so Koskenniemi’s constraint transducers *can* be intersected. In fact, when a set of two-level transducers are applied in parallel, the apply routine in a KIMMO-style system simulates the intersection of the rule automata and the composition of the input string with the virtual constraint network (see Section 1.6).

Figure 5.5 illustrates the upward application of the **N:m** and **p:m** rules sketched in Figure 5.3 to the input “kammatt”. At each point in the process, all lexical candidates corresponding to the current surface symbol are considered one by one. If both rules accept the pair, the process moves on to the next point in the input. In the situation shown in Figure 5.5, the pair **p:m** will be accepted by both rules. The **N:m** rule accepts the pair because the **p** on the lexical side is required to license the **N:m** pair that has tentatively been accepted at the previous step. The **p:m** rule accepts the **p:m** pair because the preceding pair has an **m** on the surface side.

When the pair in question has been accepted, the apply routine moves on to consider the next input symbol and eventually comes back to the point shown in Figure 5.5 to consider other possible lexical counterparts of a surface **m**. They will all be rejected by the **N:m** rule, and the apply routine will return to the previous **m**

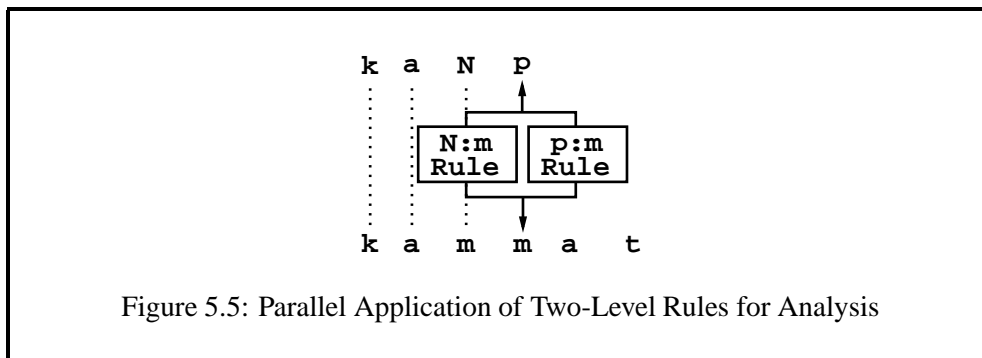


Figure 5.5: Parallel Application of Two-Level Rules for Analysis

in the input to consider other alternative lexical counterparts for it such as **p** and **m**. At every point in the input the apply routine must also consider all possible deletions, that is, pairs such as **+:0** and **e:0** that have a zero on the input side.

Applying the rules in parallel does not in itself solve the over-analysis problem discussed in the previous section. The two constraints sketched above allow “kammatt” to be analyzed as “kaNpat”, “kampat”, or “kammatt”. However, the problem is easy to manage in a system that has only two levels; the possible upper-side symbols are constrained at each step by consulting the lexicon, which is itself implemented as a kind of network. In Koskenniemi’s two-level system, lexical lookup and the analysis of the surface form are performed in tandem. In order to arrive at the point shown in Figure 5.5, we must have traversed a path in the lexicon that contains the lexical string in question, see Figure 5.6. The lexicon thus acts as a continuous lexical filter on the analysis. The analysis routine only considers symbol pairs whose lexical side matches one of the outgoing arcs of the current state of the lexicon network.

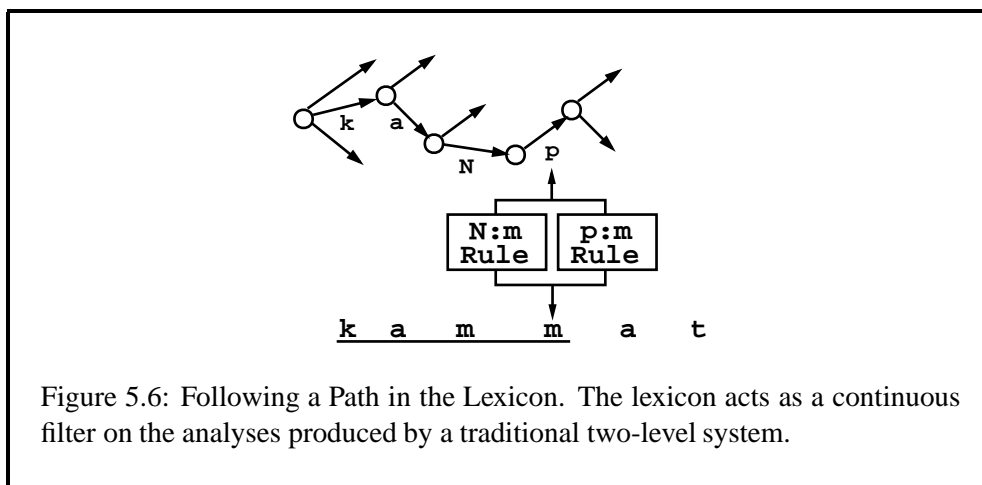


Figure 5.6: Following a Path in the Lexicon. The lexicon acts as a continuous filter on the analyses produced by a traditional two-level system.

In Koskenniemi’s 1983 system, the lexicon was represented as a forest of tries (also known as letter trees), tied together by continuation-class links from leaves

of one tree to roots of another tree or trees in the forest.² Koskenniemi's lexicon can be thought of as a partially deterministic, unminimized simple network. In the **Xerox lexc** tool, the lexicon is actually compiled into a minimized network, typically a transducer, but the filtering principle is the same. The **lookup** utility in **lexc** matches the lexical string proposed by the rules directly against the lower side of the lexicon. It does not pursue analyses that have no matching lexical path. Furthermore, the lexicon may be composed with the rules at compile time to produce a single transducer that maps surface forms directly to lexical forms, and vice versa (see Section 4.6.1).

Koskenniemi's two-level morphology was the first practical general model in the history of computational linguistics for the analysis of morphologically complex languages. The language-specific components, the rules and the lexicon, were combined with a universal runtime engine applicable to all languages. The original implementation was primarily intended for analysis, but the model was in principle bidirectional and could be used for generation.

Linguistic Issues

Although the two-level approach to morphological analysis was quickly accepted as a useful practical method, the linguistic insight behind it was not picked up by mainstream linguists. The idea of rules as parallel constraints between a lexical symbol and its surface counterpart was not taken seriously at the time outside the circle of computational linguists. Many arguments had been advanced in the literature to show that phonological alternations could not be described or explained adequately without sequential rewrite rules. It went largely unnoticed that two-level rules could have the same effect as ordered rewrite rules because two-level rules allow the realization of a lexical symbol to be constrained either by the lexical side or by the surface side. The standard arguments for rule ordering were based on the *a priori* assumption that a rule could refer only to the input context.

But the world has changed. Current phonologists, writing in the framework of OT (Optimality Theory), are sharply critical of the "serialist" tradition of ordered rewrite rules that Johnson, Kaplan and Kay wanted to formalize (Prince and Smolensky, 1993; Kager, 1999; McCarthy, 2002).³ In a nutshell, OT is a two-level theory with *ranked* parallel constraints. Many types of optimality constraints can be represented trivially as two-level rules. In contrast to Koskenniemi's "hard" constraints, optimality constraints are "soft" and violable. There are of course many other differences. Most importantly, OT constraints are meant to be universal. The fact that two-level rules can describe orthographic idiosyncrasies such as the *y~ie* alternation in English with no appeal to universal principles makes the approach uninteresting from the OT point of view.⁴

²The *TEXFIN* analyzer developed at the University of Texas at Austin (Karttunen et al., 1981) had the same lexicon architecture.

³The term *SERIAL*, a pejorative term in an OT context, refers to *SEQUENTIAL* rule application.

⁴Finite-state approaches to Optimality Theory have been explored in several recent articles (Eis-

Two-Level Rule Compilers

In his 1983 dissertation, Koskenniemi introduced a formalism for two-level rules. The semantics of two-level rules were well-defined but there was no rule compiler available at the time. Koskenniemi and other early practitioners of two-level morphology had to compile their rules *by hand* into finite-state transducers. This is tedious in the extreme and demands a detailed understanding of transducers and rule semantics that few human beings can be expected to grasp. A complex rule with multiple overlapping contexts may take hours of concentrated effort to compile and test, even for an expert human “compiler”. In practice, linguists using two-level morphology consciously or unconsciously tended to postulate rather surface lexical strings, which kept the two-level rules relatively simple.

Although two-level rules are formally quite different from the rewrite rules studied by Kaplan and Kay, the basic finite-state methods that had been developed for compiling rewrite-rules were applicable to two-level rules as well. In both formalisms, the most difficult case is a rule where the symbol that is replaced or constrained appears also in the context part of the rule. This problem Kaplan and Kay had already solved by an ingenious technique for introducing and then eliminating auxiliary symbols to mark context boundaries. Another fundamental insight was the encoding of contextual requirements in terms of double negation. For example, a constraint such as “*p* must be followed by *q*” can be expressed as “it is not the case that something ending in *p* is not followed by something starting with *q*.” In Koskenniemi’s formalism, the same constraint is expressed by the rule $p \Rightarrow \neg q$.

In the summer of 1985, when Koskenniemi was a visitor at the Center for the Study of Language and Information (CSLI) at Stanford, Kaplan and Koskenniemi worked out the basic compilation algorithm for two-level rules. The first two-level rule compiler was written in InterLisp by Koskenniemi and Karttunen in 1985-87 using Kaplan’s implementation of the finite-state calculus (Koskenniemi, 1986; Karttunen et al., 1987). The current C-version of the compiler, based on Karttunen’s 1989 Common Lisp implementation, was written by Lauri Karttunen, Todd Yampol and Kenneth R. Beesley in consultation with Kaplan at **Xerox PARC** in 1991-92 (Karttunen and Beesley, 1992). The landmark 1994 article by Kaplan and Kay on the mathematical foundations of finite-state linguistics gives a compilation algorithm for phonological rewrite rules and for Koskenniemi’s two-level rules.⁵

The **Xerox** two-level compiler has been used to compile rules for large-scale morphological analyzers for French, English, Spanish, Portuguese, Dutch, Italian

ner, 1997; Frank and Satta, 1998; Karttunen, 1998).

⁵The Kaplan and Kay article appeared many years after the work on the two-level compiler was completed but before the implementation of the so-called REPLACE RULES in the current **Xerox** regular-expression compiler. The article is accurate on the former topic, but the compilation algorithm for replace rules (Karttunen, 1995; Karttunen, 1996; Kempe and Karttunen, 1996) differs in many details from the compilation method for rewrite rules described by Kaplan and Kay.

and many other languages.

Implementations of Two-Level Morphology

The first implementation of Two-Level Morphology (Koskenniemi, 1983) was quickly followed by others. The most influential implementation was by Lauri Karttunen and his students at the University of Texas (Karttunen, 1983; Gajek et al., 1983; Dalrymple et al., 1983). Published accounts of this project inspired many copies and variations, including those by Beesley (Beesley, 1989; Beesley, 1990). A copy-righted but freely distributed implementation of classic Two-Level Morphology, called **PC-KIMMO**, available from the Summer Institute of Linguistics (Antworth, 1990), runs on PCs, Macs and Unix systems.⁶

In Europe, two-level morphological analyzers became a standard component in several large systems for natural-language processing such as the British Alvey project (Black et al., 1987; Ritchie et al., 1987; Ritchie et al., 1992), SRI's CLE Core Language Engine (Carter, 1995), the ALEP Natural Language Engineering Platform (Pulman, 1991) and the MULTEXT project (Armstrong, 1996). ALEP and MULTEXT were funded by the European Commission. The MMORPH morphology tool (Petitpierre and Russel, 1995) built at ISSCO for MULTEXT is now available under GNU Public License.⁷

Some of these systems were implemented in Lisp (Alvey), some in Prolog (CLE, ALEP), some in C (MMORPH). They were based on simplified two-level rules, the so-called PARTITION-BASED formalism (Ruessink, 1989), which was claimed to be easier for linguists to learn than the original Koskenniemi notation. But none of these systems had a finite-state rule compiler. Another difference was that morphological parsing could be constrained by feature unification. Because the rules were interpreted at runtime and because of the unification overhead, these systems were not very efficient, and two-level morphology acquired, undeservedly, a reputation for being slow. MMORPH solves the speed problem by allowing the user to run the morphology tool off-line to produce a database of fully inflected word forms and their lemmas. A compilation algorithm has since been developed for the partition-based formalism (Grimley-Evans et al., 1996), but to our knowledge there is no publicly available compiler for it.

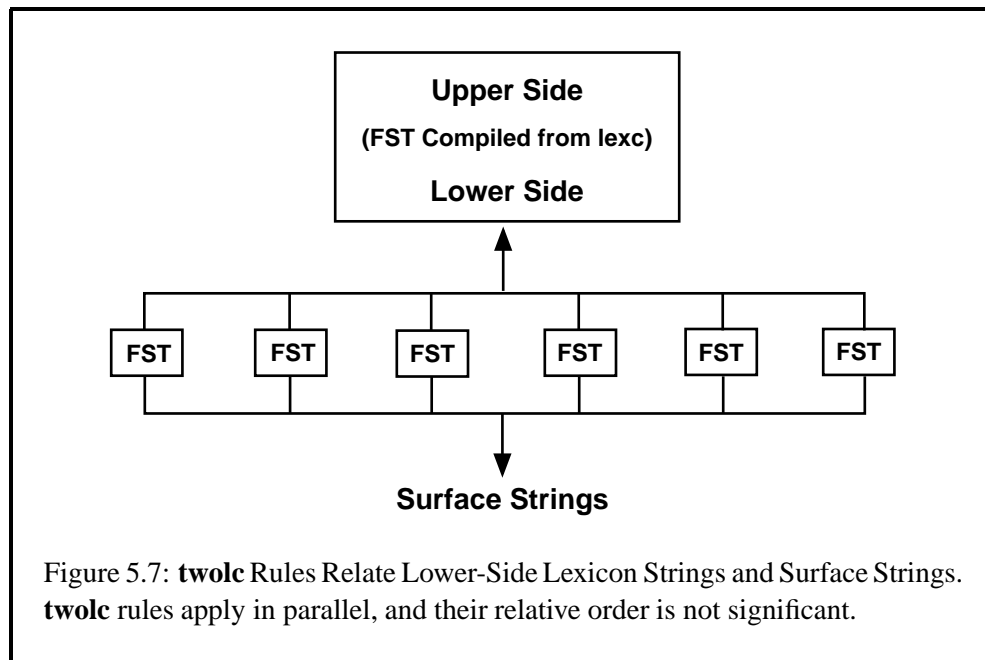
A considerable amount of work has been done, and continues to be done, in the general framework of Two-Level Morphology, and the **twolc** compiler has made that work much less onerous. The newer **Xerox** replace rules, which are part of an extended regular-expression language and are compiled using the regular-expression compiler in **xfst**, have largely supplanted **twolc** rules in some applications. For the time being, some linguistic developers will have to master both replace rules and **twolc** rules.

⁶<http://www.sil.org/computing/catalog/pc-kimmo.html>

⁷<http://packages.debian.org/stable/misc/mmorph.html>

5.1.2 Visualizing **twolc** Rules

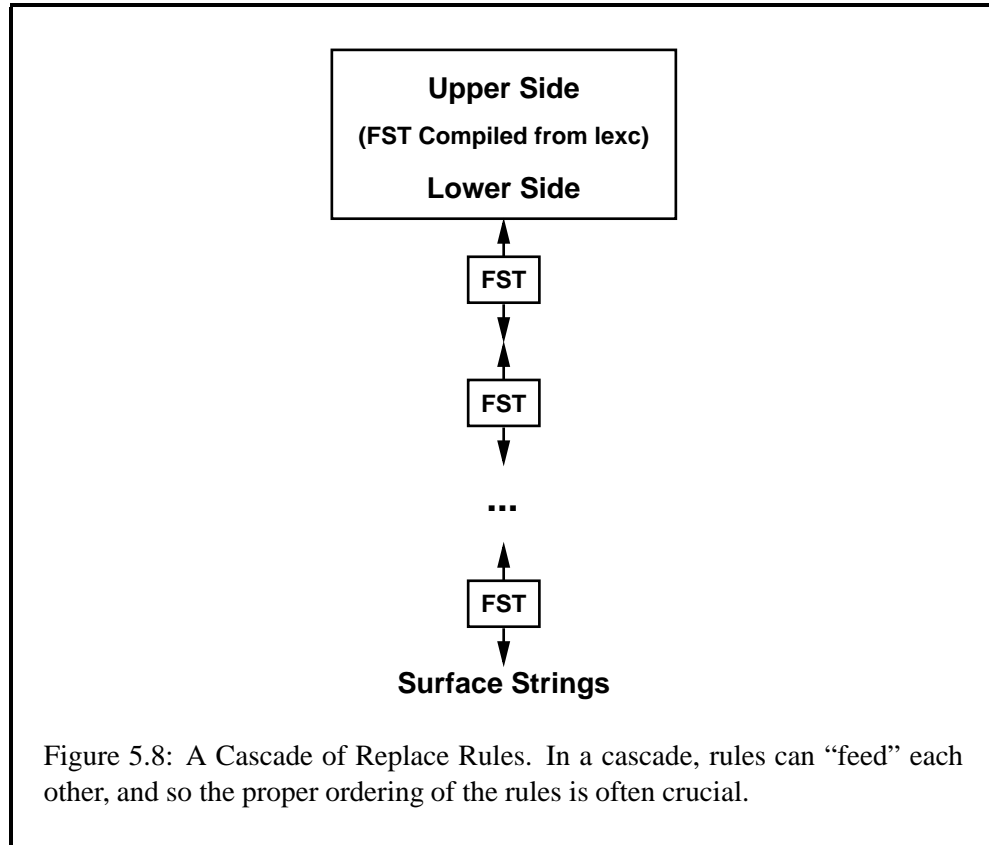
At **Xerox**, **twolc** rules are compiled automatically into rule transducers and are typically composed on the lower side of lexicon transducers made with **lexc**. As shown in Figure 5.7, the two-level rules conceptually map between strings on the lower side of the lexicon and real surface strings. That is, the upper side of each **twolc** rule transducer refers to the lower-side language of the lexicon, and the lower side of each rule transducer is the surface language. The **twolc** rules are arranged in one horizontal level and apply in parallel—the order of the rules in a **twolc** file is therefore not significant.



This model of parallel, simultaneously applied **twolc** rules must be sharply contrasted with the serial or cascade model of replace rules, as shown in Figure 5.8. Replace rules are arranged vertically, mapping strings from the lower side of the lexicon through a cascade of steps down to the surface level. The following differences must be noted and understood:

- Replace rules are organized vertically, in a cascade, and so they potentially feed each other. In contrast, a grammar of **twolc** rules is organized horizontally; the **twolc** rules apply in parallel and do not feed each other.
- Because they can feed each other, replace rules must usually be ordered carefully. A grammar of **twolc** rules, on the other hand, can be written in any order without affecting the output at all. Rule order in a **twolc** grammar is formally insignificant.⁸

⁸In a **PC-KIMMO**-like system, in which each two-level rule is stored as a separate transducer



- Replace rules conceptually produce many intermediate languages (levels) when mapping between the lower side of the lexicon and the final surface language. **twolc** rules each map directly from the lower-side of the lexicon to the surface in one step.
- **twolc** rules conceptually apply simultaneously; this avoids the ordering problems but means that sets of **twolc** rules must be carefully written to avoid nasty and often mysterious conflicts among rules. Correct, non-conflicting **twolc** rule sets are notoriously difficult to write when some of the rules perform deletion or, especially, epenthesis.
- Replace rules are compiled using the normal **xfst** regular-expression compiler, which is invoked by the **read regex** and **define** commands; replace rules are just an extension of the **Xerox** regular-expression metalanguage. **twolc** rules must be compiled using the dedicated **twolc** compiler that can be accessed only through the **twolc** interface.

and consulted individually at each step of analysis, rule order can affect the performance but not the output of a system. In particular, performance can be improved by identifying the rules that most frequently block analysis paths in practice, and ordering them early in the set so that they are checked first.

A system of replace rules is relatively easy to check and modify because each rule can be applied individually. The output from one rule can be typed as input to the next rule and the effect of a whole cascade of replace rules can be checked step by step. When we move to **twolc** rules, however, the semantics of the rules demand that we conceive of them always as simultaneously applied constraints on the relation between the lexical language and the surface language. Because **twolc** rules are designed to apply in parallel, it is difficult to test them individually.

Written correctly, grammars of unordered **twolc** rules can perform the same mappings that require carefully ordered cascades of replace rules. Conversely, any **twolc** grammar can be rewritten as a grammar of replace rules; this formal equivalence is guaranteed by the fact that both formalisms are just metalanguages for describing regular relations. The practical choice is therefore one of notational perspicuity, human ease of use, and human taste. **Xerox** developers, given a choice between **twolc** rules and replace rules, are increasingly choosing replace rules.

5.1.3 Plan of Attack

The presentation will continue as follows:

- Section 2 (Basic **twolc** Syntax) describes the basic syntax and interface commands needed to get started, consolidating the concepts with some practical exercises.
- Section 3 (Full **twolc** Syntax) completes the formal description.
- Section 4 (The Art and Craft of Writing **twolc** Grammars) discusses some useful tricks and idioms.
- Section 5 (Debugging **twolc** Rules) explains why **twolc** rules conflict with each other and how to understand and resolve the clashes. The troublesome epenthesis rules and diacritics, as well as classic programming errors, are also discussed.
- Section 6 (Final Reflections on Two-Level Rules) looks at the real choice between using **twolc** or replace rules in various applications. The possibility of upward-oriented two-level rules is also explored.

5.2 Basic **twolc** Syntax

This section contains a description of the basic **twolc** syntax that you need to get started. **twolc** source files are created using a text editor such as **emacs**, **xemacs** or **vi**. Each **twolc** file consists of named **SECTIONS**, two of which, the **Alphabet** section and the **Rules** section, are obligatory.

5.2.1 Alphabet

The `Alphabet` keyword introduces the obligatory Alphabet section, which must appear at the top of the **twolc** source file. The Alphabet section must contain at least one declared *upper:lower* symbol pair and is terminated by a semicolon.

```
Alphabet a:a ;
```

Figure 5.9: A Minimal Alphabet Section. The Alphabet section is required and must appear at the top of the **twolc** source file.

twolc grammars operate relative to an alphabet of character pairs such as **a:a**, **a:0**, **0:u**, **e:i**, etc. In **twolc** rules, the notation **z:y** must always have a single symbol on the left and a single symbol on the right of the colon. As in regular expressions, the symbol to the left of the colon is the upper-side symbol, and the symbol on the right of the colon is the lower-side symbol. When only a single symbol **a** is written, as in Figure 5.10, it is automatically interpreted by **twolc** as a shorthand notation for **a:a**.

```
Alphabet a ;
```

Figure 5.10: Declaration of **a** is Equivalent to **a:a**

The alphabet in a **twolc** grammar always consists of **SYMBOL PAIRS**, sometimes called **FEASIBLE PAIRS** in two-level systems. **twolc** uses the **u:d** notation to designate a symbol pair with **u** on the upper side and **d** on the lower side. Both **u** and **d** must be single symbols in a **twolc** grammar.

By default, **twolc** assumes that the identity symbol pairs, **a:a**, **b:b**, **c:c**, **d:d**, etc., are part of the alphabet; they do not normally have to be declared. Alphabet pairs like **e:i** and **h:0**, representing phonological or orthographical alternations, can be declared explicitly in the `Alphabet` section, or they can be declared implicitly simply by using them in a rule.

The assumptions about default identity pairs, such as **a:a**, **b:b**, **c:c**, etc., being in the alphabet are overridden by the explicit “mentioning” of particular symbols. For example, the declaration of the symbol pair **a:e** in Figure 5.11 involves mentioning both **a** and **e**, and so it overrides the default assumption that **a:a** and **e:e** are

possible character pairs in the alphabet.

```
Alphabet a:e ;
```

Figure 5.11: Declaring **a:e** Suppresses the Default Declaration of **a:a** and **e:e**

In practice, declaring **a:e** by itself causes **twolc** to conclude that **a** can appear only on the lexical side and that **e** can appear only on the surface side. If you wish to declare **a:e** and yet retain **a:a** and **e:e** as possible symbol pairs, you must then declare **a:a** and **e:e** as well as in Figure 5.12; alternatively **a:a** and **e:e** are declared implicitly if they are used anywhere in a rule.

```
Alphabet a:e a e ;
```

Figure 5.12: Regaining **a:a** and **e:e** by Overt Declaration

In practice, the alphabet in a **twolc** grammar is often a source of errors and mystery, especially when a mistake in a rule inadvertently declares an unintended symbol pair.

5.2.2 Basic Rules

The `Rules` keyword introduces the obligatory Rules section, which must contain at least one rule.

Rule Syntax

```
"Unique Rule Name"  
Center <=> LeftContext _ RightContext ;
```

Figure 5.13: The Most Commonly Used **twolc** Rule Template

The most commonly used **twolc** rules are built on the template shown in Figure 5.13. Each **twolc** rule must be preceded by a unique name in double quotes.

The *Center* part on the left side of a **twolc** rule typically consists of a single symbol pair like **u:d**, where **u** is the upper-level symbol and **d** is the lower-level symbol.

```
Rules

"Rule 1"
  s:z <=> Vowel _ Vowel ;
```

Figure 5.14: A Simple Rules section with one Two-Level Rule. The keyword *Rules* introduces the section, which must contain one or more rules. Each rule must have a unique name.

The *Center* may also be a union of two or more symbol pairs, e.g. [u1:d1 | u2:d2], that are subject to identical constraints. For example, to indicate that the voiced stops **b**, **d**, and **g** are all realized as unvoiced in the same context (e.g. at the end of the word, as in some Germanic languages), one would write the *Center* as [b:p | d:t | g:k] as in Figure 5.15.

```
"Rule 2"
  [ b:p | d:t | g:k ] <=> _ .#. ;
```

Figure 5.15: A Two-Level Rule with a Complex Left-Hand Side

After the *Center* comes an operator; the most commonly used is the <=> or double-arrow operator, typed as a left angle-bracket, an equal sign, and a right angle-bracket. After the operator, the *LeftContext* and *RightContext* are arbitrarily complex **twolc** regular expressions that surround an underscore (`_`) indicating the environment in which the *Center* relation is constrained. The *LeftContext* and/or the *RightContext* may be empty. There may be multiple contexts, and each is terminated with a semicolon as shown in Figure 5.16.

One or more rules may appear in the `Rules` section. The order of the rules in a **twolc** file has no formal effect on the functioning of the grammar.

twolc Rule Operators

In addition to the double-arrow <=> **twolc** rule operator, which is by far the most frequently used in practice, there are also the right-arrow =>, the left arrow <=, and the negated left-arrow /<= operators. All four operators are exemplified and explained in Table 5.1.

```
"Rule 3"
  n:m <=> .#. _ ;
           u _ i ;
           i _ u ;
```

Figure 5.16: A Two-Level Rule with a Multiple Contexts. Each context must be terminated with a semicolon.

The key to success in reading, writing, and debugging **twolc** rules is to know the semantics of the rule operators by heart. As unfashionable as it may seem, the student is urged to commit Table 5.1 to memory. There is no avoiding the fact that the mastery of **twolc** rules is difficult for many students, but it's impossible for those who do not learn the semantics of the rule operators.

Figure 5.17 shows examples of rules with the four different operator types and the string pairs that they allow and block; the blocked string pairs are crossed out. After learning the semantics of **twolc** rule operators, you should be able to explain why the crossed-out string pairs are blocked by each rule, and why the other string pairs are allowed.

Note that in **twolc** rules the left-arrow and right-arrow constraints are not symmetrical.

twolc Rule Contexts

Complex left contexts and right contexts are built using a syntax that resembles **xfst** regular expressions in many, but not all, ways. The following are the main points of **twolc** regular expressions:

- A notation $u : d$ is a regular expression that denotes the relation of upper-side **u** to lower-side **d**. In **twolc** the colon notation $u : d$ must always have at most a single symbol on each side of the colon.
- **twolc** contexts are always two-level. If a rule context includes a symbol x written alone, it is interpreted as $x : x$. The context written $p :$ matches any symbol pair in the alphabet having **p** on the upper side. The context written $: q$ matches any symbol pair in the alphabet having **q** on the lower side. The notation $:$, i.e. a colon with a space on each side, matches any symbol pair in the alphabet. The notation $?$ also matches any symbol pair in the alphabet.

	<i>Positive Reading</i>	<i>Negative Reading</i>
$a:b \Leftrightarrow l_r ;$	1. If the symbol pair $a:b$ appears, it must be in the context l_r . 2. If lexical a appears in the context l_r , then it must be realized on the surface as b .	1. If the symbol pair $a:b$ appears outside the context l_r , FAIL. 2. If lexical a appears in the context l_r and is realized as anything other than b , FAIL.
$a:b \Rightarrow l_r ;$	If the symbol pair $a:b$ appears, it must be in the context l_r .	If the symbol pair $a:b$ appears outside the context l_r , FAIL.
$a:b \leq l_r ;$	If lexical a appears in the context l_r , it must be realized on the surface as b .	If lexical a appears in the context l_r and is realized as anything other than b , FAIL.
$a:b / \leq l_r ;$	Lexical a is never realized as b in the context l_r .	If lexical a is realized as b in the context l_r , FAIL.

Table 5.1: **twolc** Rule Operator Semantics

- The left side of the left context and the right side of the right context are extended to infinity by concatenating the relation $:*$ on the appropriate side of each context, as shown in Figure 5.18.
- Bracketing can be used as in **xfst** regular expressions: $[b]$ is equivalent to b . An empty pair of brackets, $[\]$, denotes the empty relation that maps the empty string to itself.
- The 0 (zero) in **twolc** rules denotes not the empty string (as in **xfst**) but a special “hard zero” symbol. More will be said about this below. (As part of a larger expression like 007 , the zero is not hard.) The expression $b:0$ denotes the relation of the upper-side symbol b to the lower-side hard-zero symbol 0 , which, within **twolc** rules, is an ordinary symbol just like b .
- The concatenation of regular expressions X and Y is notated $X Y$, i.e. separated by white space and without any overt operator.
- The union of regular expressions X and Y is notated $X | Y$. In **twolc**, both curly brackets and square brackets can be used to surround unioned expressions.
- Optionality is indicated by surrounding an expression with parenthesis: (X) is equivalent to $[X | []]$.

$a:b \Leftrightarrow l _ r ;$! lar	lar	lbr	xay
	! lbr	lar	lbr	xby
$a:b \leq l _ r ;$! lar	lar	lbr	xay
	! lbr	lar	lbr	xby
$a:b \Rightarrow l _ r ;$! lar	lar	lbr	xay
	! lbr	lar	lbr	xby
$a:b / \leq l _ r ;$! lar	lar	lbr	xay
	! lbr	lar	lbr	xby

Figure 5.17: Rule Constraints Allow Certain String Pairs and Disallow Others. Review the semantics of two-level rules until you understand why each rule blocks and allows what it does.

- The Kleene Star (*), meaning zero or more iterations, can be postfixed to an expression, e.g. X^* .
- The Kleene Plus (+), meaning one or more iterations, can be postfixed to an expression, e.g. X^+ .
- The notation X^n , where n is an integer, denotes n iterations: e.g. X^3 is equivalent to $[X X X]$.
- The notation X^n, m , where n and m are integers, denotes n to m iterations.⁹
- The notation X/Y denotes the language X , ignoring any intervening strings from language Y .
- $\setminus X$ denotes the union of all the symbol pairs in the alphabet excluding pairs that belong to X . E.g. $\setminus s : z$ will match any single symbol pair in the alphabet except $s:z$.
- $\$X$ denotes the relation of all string pairs containing X . $\$a : b$ is equivalent to $[?* a:b ?*]$.
- Punctuation symbols normally interpreted as regular-expression operators can be unspecialized by preceding them with a percent sign; e.g. $\%+$ denotes

⁹In **xfst** regular expressions, the notation is slightly different: $X^{\{n, m\}}$.

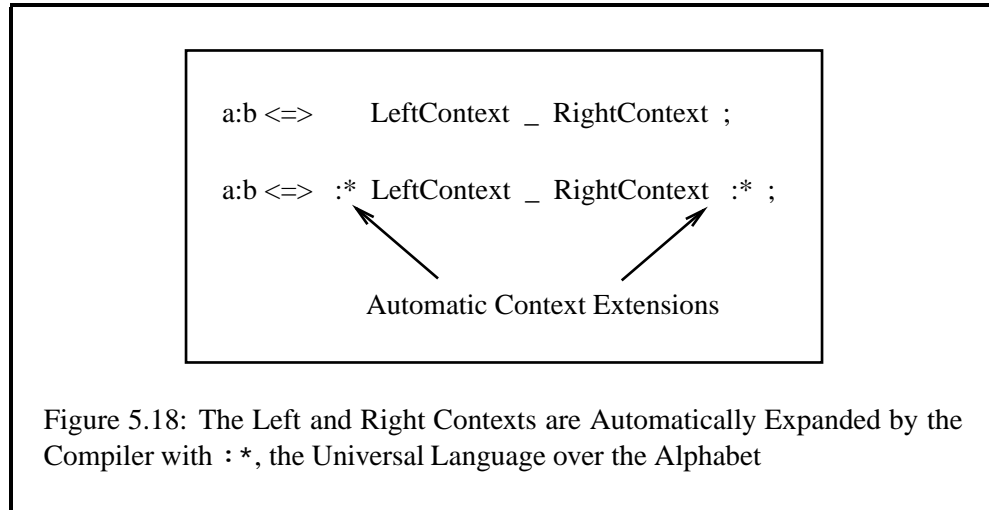


Figure 5.18: The Left and Right Contexts are Automatically Expanded by the Compiler with `:*`, the Universal Language over the Alphabet

the literal plus sign rather than the Kleene Plus operator. The notation `%0` denotes the literal digit symbol zero (parallel to 1, 2, 3, etc.) rather than the hard zero.

- The notation `.#.` denotes the absolute beginning or absolute end of a string. This notation can appear in rule contexts, but it is not a symbol per se.

Alphabets and contexts in **twolc** grammars are always two-level.

In **twolc** rules, contexts are always two-level, matching strings on both the lexical (upper) and surface (lower) sides. Although two-level regular expressions denote relations, and although relations cannot usually be intersected, complemented or subtracted, **twolc** relations are a special case that do allow these operations. The following notations are therefore valid in **twolc** regular expressions.

- Where X and Y denote **twolc** relations, $X - Y$ denotes the relation X not including any relation in Y .
- Where X and Y denote **twolc** relations, $X \& Y$ denotes the intersection of X and Y .
- Where X denotes a **twolc** relation, $\sim X$ denotes the complement of that relation.

These operations are legal in the **twolc** regular-expression language because the hard zero in pairs such as `e:0` is treated as an ordinary symbol. Consequently **twolc** rules always denote regular equal-length relations, which are closed even under complementation, intersection, and subtraction.

5.2.3 Thinking in twolc

The way to go about writing **twolc** rules is to

1. Write out a lexical string. This will typically be a string from the lower side of a lexicon transducer created with **lexc**. In our *kaNpat* example, the lexical string is “kaNpat” itself (see Section 3.5.3).

kaNpat

2. Then write out, under the lexical string, the ultimate surface string that you want to generate using the entire grammar of rules.

Lexical: kaNpat
Surface: kammatt

3. Align the two strings, symbol by symbol. Pad out the strings with hard zeros if necessary until they are of *exactly* the same length. Where you put the zeros is ultimately up to you, but be consistent and maximize the linguistic motivations for your choices. You will end up with a pair of strings consisting of a sequence of symbol pairs. These are the two levels of a **twolc** or any two-level morphology. The following lineup might be appropriate for a language where an underlying “banay+at” string is realized as “banat” on the surface.

Lexical: banay+at
Surface: ban000at

4. Identify the ALTERNATIONS or discrepancies between the two strings. Write **twolc** rules as necessary to allow and constrain the symbol pairs that account for the alternations. The *banay+at* example just above would require rules to constrain where **a:0**, **y:0** and **+:0** can and must occur. The *kaNpat* example will require two rules to constrain where **N:m** and **p:m** can and must occur.

Of course, for most real natural languages, there are many alternations between the lexical and surface strings, and you will have to write out and align many pairs of strings as part of your grammar planning. If you don’t write out and line up your lexical and surface strings, identify your symbol pairs, and then write suitable **twolc** rules, in that order, you’re doing it wrong.

There are often several reasonable ways to line up the strings, with hard zeros in different places, that lead to equivalent results. For example, the “banay+at” string might be aligned as

Lexical: banay+at
Surface: bana000t

However, any grammar of compatible rules will depend on the examples being lined up consistently. It is highly recommended that you document your rules with comments showing some sample pairs of lexical and surface strings to remind yourself how you originally decided to line up the symbol pairs. Changing the lineup conventions in the middle of development may require changes to multiple rules—remember that all the rules apply in parallel and must agree in how the two levels line up.

Plan and document the lexical-surface lineup of all symbol pairs in your examples. This lineup must be motivated and consistent so that a coherent set of two-level rules can be written.

twolc comments are preceded by an exclamation mark and continue to the end of the line.

```
! twolc comments extend to the End of Line

! banay+at      comment alignments for future reference
! ban000at

! kaNpat
! kammatt

! Comments are Good Things, use lots of them
```

Use plenty of comments in your **twolc** files, including examples that show how the lexical and surface strings are lined up symbol by symbol.

twolc rules differ from replace rules both in their syntax and semantics, and this is a common source of confusion. The following points need to be emphasized:

1. Basic **twolc** rules constrain a single pair of symbols. The left side of a **twolc** rule consists of a single symbol pair **u:d**, with one single symbol **u** on the upper side and another single symbol **d** on the lower side. Multicharacter symbols are possible, and as in **xfst** they are declared implicitly by writing several symbols together: e.g. **%+Noun:o**. As the plus sign is a special character in **twolc** rules, literalize it where necessary by preceding it with a percent sign (%).¹⁰

¹⁰In **twolc** regular expressions, special characters cannot be literalized by surrounding them in double quotes, as in **xfst**.

2. **twolc** contexts are always interpreted to be two-level. That is, in **twolc** rules, the contexts always refer to both the upper and lower sides of the relation. You can leave the lower side unspecified by leaving the right (lower) side of a pair empty, as in the `u :` notation. You can leave the upper side unspecified, as in the `:l` notation. A colon by itself, i.e. `:`, refers to any single symbol pair in the alphabet.

3. **twolc** grammars always work relative to an alphabet of symbol pairs. The alphabet for a **twolc** rule transducer is the collection of overtly and implicitly declared symbol pairs.

4. Within **twolc** rules, the 0 (hard zero) character is a real character and should be treated as such when aligning examples and writing rule contexts. The hard zeros of **twolc** are therefore different from the zeros in **xfst** and **lexc**, where 0 simply denotes an epsilon or empty string.

5. **twolc** rule arrows have their own semantics, as shown in Table 5.1. The left-arrow and right-arrow restrictions are not symmetrical, and both are quite different from the semantics of **xfst** replace rules.

5.2.4 Basic **twolc** Interface Commands

In order to attempt the first exercise below, you will need to know the basic commands in the **twolc** interface to read and compile your **twolc** source file. The **twolc** interface is invoked by entering **twolc** at the command line.

```
unix> twolc
```

twolc will respond with a welcome banner, a menu of commands, and a **twolc** prompt.

```

*****
*               Two-Level Compiler 3.1.8 (7.8.1)               *
*                   created by                                   *
*                   Lauri Karttunen, Todd Yampol,              *
*                   Kenneth R. Beesley, and Ronald M. Kaplan.  *
*   Copyright (c) 1991-2002 by the Xerox Corporation.          *
*                   All Rights Reserved.                       *
*****

Input/Output -----
Rules:          read-grammar.
Transducers:   install-binary, save-binary, save-tabular.
Lexicon:       install-lexicon, uninstall-lexicon.
Operations -----
Compilation:   compile, redo.
Intersection:  intersect.
Testing:       lex-test, lex-test-file, pair-test,
               pair-test-file.
Switches:      closed-sigma, quit-on-fail, resolve, time,
               trace.
Display -----
Result:        labels, list-rules, show, show-rules.
Misc:          banner, storage, switches.
Help:          completion, help, history, ?.
Type 'quit' to exit.
twolc>

```

You can cause the menu of commands to be redisplayed at any time by entering a question mark. You will need to know just a few utilities from the **twolc** interface to get started.

- **read-grammar** *filename* reads in your source *filename* and checks for purely syntactic errors.
- **compile** causes a successfully read-in **twolc** source file to be compiled into finite-state transducers, one transducer for each rule. Compilation is therefore a two-step process of **read-grammar** followed by **compile**.
- **lex-test** allows you to test your compiled rules immediately by manually inputting lexical strings for generation.
- **save-binary** saves the compiled rule transducers to file. Each individual transducer is written into the file separately, they are not automatically combined into a single network by intersection.
- **save-tabular** is used to save compiled **twolc** rules in the tabular format required by **TwoL** and **PC-KIMMO**.

- **quit** exits from **twolc**.

5.2.5 Exercises

The kaNpat Exercise

The first exercise, just to get used to using basic **twolc** commands, is to redo the *kaNpat* example using **twolc** rules. The first step, as usual, is to write out the relevant examples as string pairs, with the lexical string on the upper side and the surface string on the lower side.

Lexical:	kaNpat	kampat	kammat
Surface:	kammat	kammat	kammat

That is, we want lexical “kaNpat” to map to surface “kammat”, lexical “kampat” also to map to surface “kammat”, and in addition lexical “kammat” will map to itself. A little study of these examples will show that in some precise environments a lexical **N** has to map to a surface **m**, and in other precise environments a lexical **p** must map to a surface **m**. In addition we want to allow **m:m** and **p:p** in other environments.

Type in the following **twolc** source file using a text editor and save it to file as `kaNpat-twolc.txt`.

```
! This is a comment. Use lots of them.

Alphabet m p ;

Rules

"N:m rule"
N:m <=> _ p: ;

"p:m rule"
p:m <=> :m _ ;
```

Enter **twolc** and use **read-grammar** and **compile** to compile the rules.

```
twolc> read-grammar kaNpat-twolc.txt
twolc> compile
```

If **read-grammar** reports syntax errors, re-edit the source file until it reads in cleanly. During compilation, ignore any warning messages about the left-arrow restriction of the N:m rule being redundant. Any genuine error message indicates a typing error that you should correct before proceeding. Test the grammar using **lex-test**, inputting the lexical strings “kaNpat”, “kampat” and “kammat”.

```
twolc> lex-test kaNpat
```

They should all have the same surface output “kammatt”. Input other strings like “book” and “monkey” that are not affected by the rules; they should all be mapped to themselves.

Edit the source file, reverse the order of the rules, recompile and retest. Such reordering has no effect in a **twolc** grammar because the rules are applied in parallel.

In this example, **m** and **p** are declared in the Alphabet section; this is equivalent to declaring **m:m** and **p:p**. These declarations are required because we “mention” the symbols **p** and **m** in the pairs **p:m** and **N:m** in the rules. Unless we go back and explicitly declare **m:m** and **p:p**, **twolc** will assume that **p** can appear only on the lexical side and that **m** can appear only on the surface side. We don’t declare **N:N**, in this case, because **N** stands for an abstract underspecified morphophoneme that should never reach the surface as **N**. In a more complete grammar, **N:n** and **N:NG** (where **NG** is the name of a single symbol) may also be possible realizations in other environments, but **N** itself should never reach the surface level. Any UNKNOWN symbol that does not appear anywhere in the rules is treated as an identity pair. If you try the input “hello” in **lex-test** on your grammar, you will see that the output string is “hello”. Because the symbols **h**, **e**, **l**, and **o** are not affected by any rule in the *kaNpat* grammar, the compiler does not impose any constraints on them.

Note that the **N:m** rule has a right context **p:** which matches all symbol pairs in the alphabet that have **p** on the upper side. Because the parallel **p:m** rule is relating this same upper-side **p** to a lower-side **m**, why is it important for the **N:m** rule to specify a right context of **p:** rather than **p:p** or **p**?

Similarly, the **p:m** rule has a left context **:m**, which matches all symbol pairs in the alphabet having **m** on the lower-side. Why is it important in this case to specify a left context of **:m** rather than **m:m** or **m**?

Try editing the source file to make the contexts more specific and see if the rules still produce the correct output. You will find in writing **twolc** rules that making contexts too specific is just as dangerous as not making them specific enough.

Brazilian Portuguese Pronunciation

The next exercise is to redo the Brazilian-Portuguese Pronunciation exercise using **twolc** rules. Refer back to page 150 for the facts to be captured.

The first step, as always with **twolc** rules, is to line up many examples as lexical/surface string pairs. Match the lexical and surface symbols one-to-one as economically, consistently and beautifully as possible. Use hard zeros to pad out the lexical and surface strings so that each pair of strings has the same length. Use zeros consistently in parallel examples. Table 5.2 shows our recommended way to line up the Portuguese-pronunciation examples, but it is ultimately up to each linguist to define how the levels line up. Another way of saying this is that it is

me mi	disse Jis0i	tarde tarJi	partes parCis	verdade verdaJi
do du	tio Ciu	filho fiL0u	ninho niN0u	carro kaR0u
caro karu	camisa kamiza	vez ves	zebra zebra	casa kaza
peruca piruka	braço brasu	chato \$0atu	chatinho \$0aCiN0u	interesse interes0i
homem 0omem	rápido Rápidu	peru piru	braços brasus	hostil 0osCil
antes anCis	paredes pareJis	livros livrus	cada kada	cedilha seJiL0a
pedaço pedasu	parte parCi	parede pareJi	sabe sabi	simpático simpáCiku
filhos fiL0us	case kazi	ninhos niN0us	cantar kantar	bicho bi\$0u
time Cimi	fortes forCis	usar uzar	dente denCi	tempo tempu
dia Jia	rato Ratu	cimento simentu	cases kazis	luz lus
cachorro ka\$0oR0u	vermelho vermeL0u	diferentes JiferenCis	sonho soN0u	e i

Table 5.2: Pairs of Strings for the Portuguese Pronunciation Example. The **twolc** rules should accept the upper-side strings as input and produce the lower-side strings as output.

up to the linguist to decide where the zeros are. It is also up to the linguist to be consistent in placing the zeros.

The fully lined-up string pairs will identify the symbol alternations between the lexical and surface levels. Write the necessary **twolc** rules to constrain the relation between lexical and surface strings. Keep the semantics of **twolc** rules constantly in mind. Remember that you are writing rules that will apply in parallel, and remember also to treat zeros as real symbols inside **twolc** rules.

Here are a few hints:

- There should be only one rule to constrain each symbol pair such as d:J and r:R.
- A rule may have multiple contexts, each context terminated with a semi-colon.

- The dollar sign (\$) is a special character in **twolc**, as in other **Xerox** regular expressions. To use it as a literal character representing the phoneme /f/, it must be made un-special by preceding it with the literalizing percent sign (%).
- Remember not to make your contexts overly specific. In several rules you will need to specify a context only on the upper side, e.g. `h :`. In others, you will need to specify only a lower-side context, e.g. `: i`.
- Once the rules have been successfully compiled, try testing a few examples manually using the command **lex-text**.
- For batch testing, type the upper-side strings into a file, one word to a line, and use the command **lex-test-file** to generate all of them. Edit and retest your rules until all the examples are generated correctly.

This exercise is not simple for most people. A solution is provided on page 628.

5.3 Full twolc Syntax

5.3.1 Header Sections

Each **twolc** source file must start with an Alphabet section and include a Rules section as described above. In addition, the following optional sections may appear between the Alphabet and Rules sections.

Sets

To aid in writing rules, you can optionally define sets of characters. The keyword `Sets` is followed by assignment entries:

```
Sets
  Vowels = a e i o u ;
  Conson = b c d f g h j k l m n p q r s t v w x y z ;
```

Each entry consists of a set name on the left side, an equal sign, and a list of symbols; and each entry is terminated with a semicolon. Because the semicolon indicates the end of each set, you can continue the listing over multiple lines.

Set definitions can include previous Set definitions.

```
Sets
  Vowels = a e i o u ;
  Conson = b c d f g h j k l m n p q r s t v w x y z ;
  ExtendedVowels = Vowels w y ;
```

These defined set names can then be used in rule contexts and in rule-scoped variables (to be presented below). A bug (or “feature”) of sets is that all the symbols mentioned in a set must be listed explicitly in the Alphabet section.

Definitions

If multiple rules require the same left or right context (or a significant part of a context) it may be wise and convenient to define the context in the optional Definitions section. The syntax is similar to that in the Sets section, but the right side of each assignment is a **twolc** regular expression. Definitions can make use of previously declared definitions.

```

Definitions
  XContext = [ p | t | k | g:k ] ;
  YContext = [ m | n | n g ] ;
  ZContext = [ a | e | i | o | u ]* XContext ;

```

Diacritics

Diacritics, explained below, are a deprecated feature of **twolc**. Do not confuse **twolc** Diacritics with the very different Flag Diacritics explained in Chapter 8.

5.3.2 Full twolc Rule Syntax**Multiple Contexts**

A single rule can have multiple contexts, e.g.

```

"Rule 1"
s:z <=> Vowel _ Vowel ;
      Vowel _ [ m | n ] ;

```

Each context must end with a semicolon. Left-arrow restrictions are interpreted conjunctively, imposing the constraint that a lexical **s** must be realized as **z** in all of the indicated contexts. Right-arrow restrictions are interpreted disjunctively, limiting the symbol pair, here **s:z**, to appearing in any of the indicated contexts (but in no other contexts).

Rule-Scoped Variables

A **twolc** rule may contain any number of local VARIABLES that range over a set of simple symbols. Variables and the assignment of values to them are specified in a *where* clause that follows the last context of the rule. A *where* clause consists of (1) the keyword *where* followed by (2) one or more variable declarations, and (3) an optional keyword (*matched* or *mixed*) that specifies the mode of value assignment. A variable declaration contains (1) a local-variable name chosen by the programmer (2) the keyword *in*, and (3) a range of values. The range may be either a defined set name or a list enclosed in parentheses containing symbols or set names, for example.

```
"foo rule"
Cx:Cy <=> _ .#. ;
           where Cx in (b d c g)
                Cy in (p t c2 k)
           matched ;
```

In this rule, the two local variables, Cx and Cy, have four possible values. The keyword `matched` means that the assignment of values to the two variables is done in tandem so that when Cx takes its nth value, Cy has its nth value as well. The compiler interprets this type of rule as an intersection of four independent subrules:

```
b:p <=> _ .#. ;
d:t <=> _ .#. ;
c:c2 <=> _ .#. ;
g:k <=> _ .#. ;
```

If the keyword `matched` is not present at the end of a `where` clause, the assignment of values to variables is not coordinated. In this case, assigning values freely to the two variables would create 16 subrules with all possible pairings of values from the two sets. The keyword `mixed` indicates such a free assignment overtly.

Matched variables are also convenient when there is a dependency between the correspondence and context parts of the rule. A case of that sort is a rule that creates a geminate consonant at a morpheme boundary:

```
"gemination"
%+:Cx <=> Cy _ Vowel ;
           where Cy in (b c d f g k l m n p r s t v z)
                Cx in (b k d f g k l m n p r s t v z)
           matched ;
```

For example, the rule would realize lexical forms such as “big+er”, “picnic+ed” and “yak+ing” as the appropriate English surface forms “bigger”, “picnicked” and “yakking”.

Variables can also be used to encode dependencies between the left and right parts of the context. For example, if `HighLabial` is defined as

```
HighLabial = u y ;
```

the following rule realizes **k** as **v** in two contexts. (Assume that `ClosedOffset` is a regular expression defined in the Definitions section.)

```
"Gradation of k between u/y" ! k weakens to v between
                               ! u's and y's
k:v <=> Cons Vx _ Vx ClosedOffset ;
           where Vx in HighLabial ;
```

In this case no variable occurs in the correspondence part of the rule; the compiler only needs to expand the context part as follows:

```
k:v <=>  Cons  u  _  u  ClosedOffset ;
          Cons  y  _  y  ClosedOffset ;
```

Because the rule contains only one variable, the interpretation of the “Gradation of k between u/y” rule is the same regardless of whether the keyword matched is present or absent.

5.3.3 Full twolc Interface

Utilities for Rule Testing and Intersection

In addition to the basic **twolc** operations described above, there are also the following utilities that may prove useful. Use **help** for more information.

install-binary

install-binary is a command to read a binary file from disk for use in the **twolc** interface. A binary file is one that has already been compiled, typically by previous invocations of **read-grammar**, **compile** and then **save-binary**.

intersect

When **twolc** compiles a set of rules, each rule is stored as a transducer, and the transducers are kept separate by default. To force **twolc** to intersect the various transducers into a single transducer, invoke **intersect** after invoking **compile**.

```
twolc> read-grammar yourlang-twolc.txt
twolc> compile
twolc> intersect
twolc> save-binary yourlang-twolc.fst
```

Keeping the transducers separate by default is important for rule testing with **pair-test** (see below). If all the rule transducers were automatically intersected into a single transducer, then **pair-test** could not identify which of the component rules was responsible for blocking a mapping between two strings.

lex-test

The **lex-test** utility allows the linguist to test a set of compiled rules by manually typing in lexical strings for generation. **lex-test** outputs the surface form or forms dictated by the rules.

Note that **lex-test** applies the rule transducers in a downward direction, performing generation on the input strings. The opposite, analysis, direction is

not generally possible because most practical rule sets produce an infinite number of analyses when those analyses are not constrained by a lexicon.

lex-test-file

lex-test-file is like **lex-test** except that you specify an input file (consisting of a list of strings, written one to a line, to be generated from) and an output file for the results. The interface prompts you for the filenames.

pair-test

pair-test is a useful utility that allows you to identify which rule in your rule set is blocking a desired mapping. Suppose that you have written a set of 50 rules, and that one of the desired and expected mappings is the following:

```
Lexical: simpático
Surface: simpáCiku
```

If, in fact, you input “simpático” to **lex-test** and there is no output, or if the output is not the desired “simpáCiku”, then identifying the villain in a set of 50 rules may be anything but trivial. The solution is to invoke **pair-test**, which will prompt you for the lexical string and the surface string that you think should be generated from it. Then **pair-test** will run the two strings, symbol pair by symbol pair, through the rule set and identify by which rule, and at which position in the string, the desired mapping is being blocked.

Note that for **pair-test** the lexical string and the surface string must contain exactly the same number of symbols, including hard zeros positioned in exactly the right places. It’s at times like this, testing for blocked derivations, that you will be especially grateful for your own documentation showing how lexical and surface strings are supposed to line up.

pair-test-file

The **pair-test-file** utility is like **pair-test** except that it takes its input (pairs of strings) from a file you specify and outputs its result to another file that you specify. The input file should be formatted as follows, with a blank line

between the pairs of strings:

```
simpático
simpáCiku
```

```
partes
parCis
```

```
vermelho
vermeL0u
```

```
ninho
niN0u
```

```
carro
kaR0u
```

If you maintain a file of valid lexical-surface pairs, i.e. mappings that the rules should perform, then re-running that file periodically through **pair-test-file** is a valuable form of regression testing.

install-lexicon

install-lexicon allows you to load a lexicon (or pseudo lexicon) into the **twolc** interface for filtering out conflict reports. When compiling two-level rules, **twolc** is extremely good (sometimes dismayingly good) at identifying and reporting possible conflicts between rules. But **twolc** assumes that *any* possible string, i.e. the universal language, could be applied for generation—if the conflicts occur only for input strings that cannot, in fact, occur in the natural language in question, then the rule writer may find the conflict hard to understand, and fixing the rules may seem pointless or at least a profound nuisance.

Consider the following two rules.

```
"Rule 1"
x:y <=> _ a ;

"Rule 2"
x:z <=> i _ ;
```

During compilation, **twolc** will dutifully report a left-arrow conflict between the two rules, recognizing that when generating from a possible input string like “ixa”, Rule 1 would insist on realizing the **x** as **y**, yielding “iya”; whereas the second rule would insist on realizing the same **x** as **z**, yielding

“iza”. This is a fatal left-arrow conflict, assuming that a string containing “ixa” might indeed be input for generation.

However, it may be the case in the language in question that the sequence “ixa” simply never occurs; and therefore the potential conflict between the two rules just doesn’t matter. It is in such cases that **install-lexicon** can prove useful. Let us assume that the linguist has already built a lexicon transducer, perhaps using **lexc**, and that this transducer has been saved to file as `lex.fst`. Let us also assume that the lower-side language of `lex.fst` naturally contains no strings containing “ixa”. After **install-lexicon** has been invoked to read this `lex.fst` into **twolc**’s lexicon buffer, **twolc** will subsequently consult the lower side of the lexicon to filter out rule-conflict messages that just don’t matter. For the two rules above, the potential conflict will be ignored, and no conflict message will be reported, when **twolc** finds that the lower-side language of the lexicon includes no string containing the problematic “ixa” sequence.

The lexicon loaded using **install-lexicon** need not be a real lexicon, and it need not even be two-level. In the case above, the linguist could compile, using **xfst**, a simple regular expression

```
~$[ i x a ] ;
```

denoting the language of all strings that do not contain “ixa”. Installing this pseudo-lexicon will be sufficient to suppress the left-arrow conflict report for the cited rules. **install-lexicon** can therefore use either a real lexicon or a pseudo-lexicon, and the pseudo-lexicon could of course be arbitrarily complex, tailored to filter out all kinds of error messages that are not relevant for the language being modeled.

uninstall-lexicon

The **uninstall-lexicon** utility removes and discards a lexicon previously installed using **install-lexicon**. Once uninstalled, the lexicon will no longer be used for filtering out conflict messages.

Miscellaneous twolc Utilities

In addition, **twolc** provides more utilities for timing, tracing, examining compiled rules, etc. Most of these utilities are more useful to **twolc** maintainers and debuggers than to the average user.

You can cause the full menu of **twolc** utilities to be displayed at any time by entering a question mark (?) at the prompt. Use the **help** utility to see some short documentation for each command.

5.3.4 Exercises

Monish Vowel Harmony

Review the facts of the mythical Monish language starting on page 166. Then redo the exercise using **lexc** and **twolc**.

- Create a **lexc** file called `monish-lex.txt`, compile it using **lexc**, and save the result to file as `monish-lex.fst`.
- Create a **twolc** file for Monish vowel-harmony called `monish-rul.txt`, compile it using **twolc**, and save the result as `monish-rul.twol`. The trick is to write a grammar that realizes each underspecified vowel as front or back depending on the frontness or backness of the previous vowel. Two possible solutions are shown on page 634.
- Compose the lexicon and rules using the **lexc** interface, and save the result to file as `monish.fst`, i.e.

```
lexc> read-source monish-lex.fst
lexc> read-rules monish-rul.twol
lexc> compose-result
lexc> save-result monish.fst
```

- Test the resulting system using **lookup** and **lookdown** in **lexc**. It should analyze and generate examples like the following.

yääqin+Perf+2P+Pl yääqinenémerä
fesééng+Opt+False+1P+Pl+Incl fesééngiddéqäåbigä
bunoots+Int+Perf+2P+Sg bunootsuukonóma
tsarlók+Opt+False+1P+Sg tsarlókuddóqaaba
ntonól+Imperf+1P+Pl+Excl ntonólómbaabora

- Test for bad data as well. Your system should not be able to analyze the following ill-spelled Monish words.

yääqinenémorä [contains 'o' in a front-harmony word]

tsarlókuddóqaabe [contains 'e' in a back-harmony word]

5.3.5 Understanding twolc and Networks

When you compile a two-level rule grammar with **twolc** and save the result as a binary file using **save-binary**, the transducers are saved into the file one-by-one. They are not intersected into a single network, unless you have explicitly already done so by invoking the **intersect** command.

A file that contains several compiled two-level rules can be loaded back into **twolc** and **lexc**, but in general it is not advisable to load such a file into **xfst**. In **xfst**, the multiple rule networks are pushed onto The Stack as usual; but any subsequent stack operation on them, such as composition, is not likely to produce the intended result. Intersection of two-level networks in **xfst** works correctly but prints a flurry of warning messages if the rules contain one-sided epsilon pairs because intersection is not in general applicable to such networks.

xfst is a general tool for computing with finite-state networks, and the transducers that represent compiled two-level rules have no special status within **xfst**. Let us recall, for example, that the **apply up** and **apply down** commands in **xfst** apply only the top network on The Stack to the input. If the stack contains several rule transducers that were intended to be applied in parallel, the result will not be as expected.

In **lexc**, you use the command **compile-source** or **read-source** to put a network in the **lexc** SOURCE register, and you use **read-rules** to read one or more rule networks, compiled previously by **twolc**, into the RULES register; when you then invoke **compose-result**, **lexc** uses a special INTERSECTING COMPOSITION¹¹ algorithm that composes the SOURCE with the RULES and puts the resulting single network into the RESULT register. For example, if you have a lexicon `my-lex.fst` previously compiled and saved in **lexc**, and if you have a file `my-rul.twol` that you compiled and saved in **twolc**, the following series of **lexc** commands will compose them and write out the single result to the file `result.fst`.

```
lexc> read-source my-lex.fst
lexc> read-rules my-rul.twol
lexc> compose-result
lexc> save-result result.fst
```

Note here that **read-source** is used to read in an already compiled transducer from file. If all you have is the `my-lex.txt` source file, do the following instead to compile the **lexc** source file and put the lexicon network in the SOURCE register. Note that the rule file `my-rul.twol` must always be compiled separately by **twolc** and stored using **save-binary**; you cannot compile a source file of **twolc**

¹¹As its name suggests, the intersecting composition algorithm performs the intersection and composition of the rules simultaneously. See pages 254, 290 and 289.

rules from inside **lexc**.

```
lexc> compile-source my-lex.txt
lexc> read-rules my-rul.twol
lexc> compose-result
lexc> save-result result.fst
```

There are good practical reasons for storing the results **twolc** by default as a set of separate rule networks. These rule transducers are not intersected, unless you specifically tell **twolc** to do so using **intersect**, for four reasons.

1. **twolc** allows you to **show** the transducer corresponding to an individual rule, which can be useful for debugging. The command **show** is followed by the name of a rule.

```
twolc> show unique_rule_name
```

2. The **pair-test** utility applies the rules individually to a pair of strings and tells you which rule is preventing the mapping; it can't identify the individual rule blocking the mapping if all the rule transducers are intersected into a single transducer.
3. **twolc** also offers **save-tabular** output, which outputs the rule transducers in a state-table format suitable for use by **PC-KIMMO** and the **TwoL** implementation of Lingsoft.¹² Here again, separate rule transducers are needed for debugging, and an intersected rule transducer might grow too big for practical use on small machines.
4. The intersection of large rule systems may be impossible or impractical because the size of the resulting network may be very large. In the worst case, the intersection of two networks, one with k states, other with n states, may produce a network with the product $k \times n$ states.

The recipe to compile and intersect the rules within **twolc** looks like this:

```
twolc> read-grammar yourlang-rul.txt
twolc> compile
twolc> intersect
twolc> save-binary yourlang-rul.fst
```

If the rule networks can be intersected into a single network, all further processing, such as the composition with an already compiled source lexicon, can be done with the generic **xfst** interface. The intersection should be done within **twolc** because

¹²If you hand-compile your two-level rules for **PC-KIMMO** they must be typed in this same state-table format.

it will be done more efficiently and without provoking the warning messages that **xfst** would generate.

The composition of an entire lexicon transducer with a full set of **twolc** rules can blow up in size, at least temporarily during the computation. The intersecting-composition algorithm in **lexc** was once the only way to handle such large compositions, and it may still be the only way in certain cases. In the meantime, the general composition algorithm, as found in **xfst**, has become more efficient and may be used directly in most or all cases in which the intersection of the rules is practical.

5.4 The Art and Craft of Writing **twolc** Grammars

5.4.1 Using Left-Arrow and Right-Arrow Rules

In most practical **twolc** rule systems, the vast majority of the rules are, and should be, double-arrow (\Leftarrow) rules. Attempts by beginners to use left-arrow (\Leftarrow) and right-arrow (\Rightarrow) rules are usually mistakes, revealing a poor grasp of the semantics of the rule operators (see Table 5.1, page 326).

One case in which the use of single-arrow rules is justified and necessary is when the realization of a lexical symbol, e.g. **e**, is obligatorily one thing, e.g. **i**, in context A, but either **e** or **i** in a different context B.

Modifying the Portuguese Pronunciation data, let us assume a dialect in which the pair **e:i** can appear at the end of a word, and in this context, a lexical **e** must be realized as **i**. That would normally be expressed with a double-arrow rule as in Rule 1.

```
"Rule 1"
e:i <=> _ .#.
```

Let us further assume that the pair **e:i** is also possible in the context $. \# . p _ r$, but that the realization of lexical **e** in this context is not obligatory: it could be realized as either **i** or as **e**. If the Alphabet supports only **e:e** and **e:i** as possible realization of **e**, then we would notate these phenomena with a right arrow (\Rightarrow) as in Rule 2.

```
"Rule 2"
e:i => .#. p _ r ;
```

The problem here is that Rule 1 and Rule 2 are in right-arrow conflict; Rule 1 states that the pair **e:i** can occur only in the context $_ . \# .$, and Rule 2 states that the same pair can occur only in the context $. \# . p _ r$. The solution that avoids all conflicts is the following pair of rules, one using the left arrow, and one the right arrow.

```
"Rule 1' "
e:i <= _ .#.

"Rule 2' "
e:i => .#. p _ r ;
      _ .#. ;
```

Rule 1' now indicates that if a lexical **e** appears at the end of a word, it must be realized as **i**, but this left-arrow rule places no constraints on where the pair **e:i** can appear. Rule 2' now indicates that the pair **e:i** can appear only in the two contexts indicated, without forcing lexical **e** to be realized as **i** in either of them. By combining the left-arrow and right-arrow constraints of the two rules, the desired behavior is achieved. The lexical input “sabe” will be generated as “sabi”, and the lexical input “peru” will be generated as both “peru” and “piru”.

Such alternations that are obligatory in one context but optional in another are fairly rare, at least in standard orthographies. The single-arrow rules might be more useful when modeling phonological alternations.

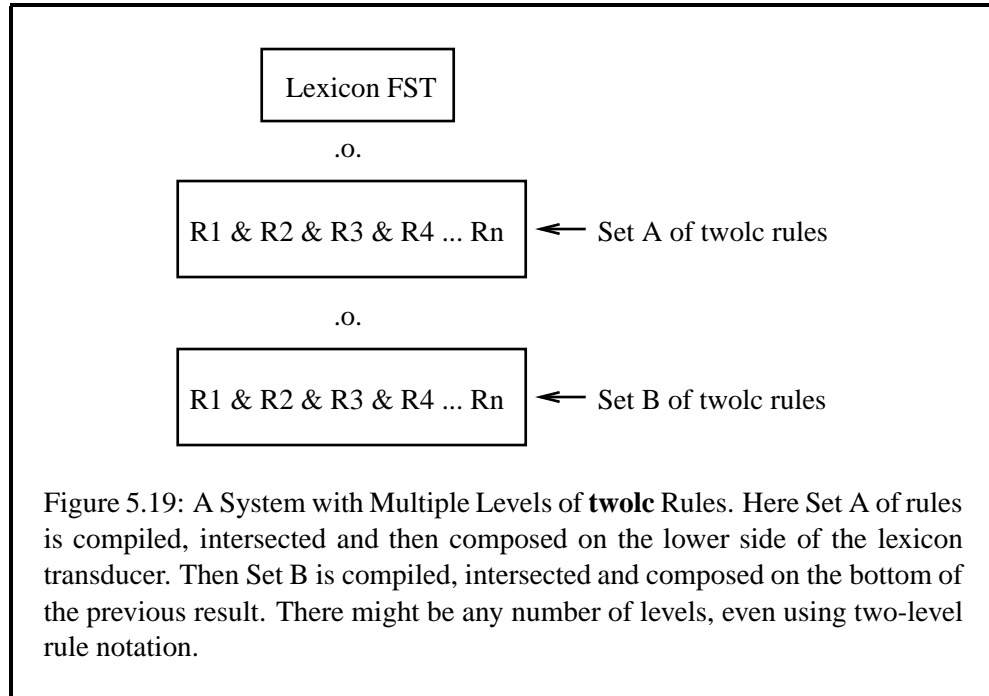
5.4.2 Multiples Levels of Two-Level Rules

Koskenniemi-style Two-Level Morphology and the general two-level OT approaches to phonology and morphology claim that it is not only possible but virtuous to have a single level of simultaneous rules that directly constrain lexical-surface mappings. Proponents insist that the intermediate languages suggested by sequential (cascaded) rules do not exist and have no place in the notation.

However, given that sets of two-level rules compile into transducers, there is no formal restriction against writing a system that includes multiple levels of two-level rules, or even mixtures of sets of two-level rules with sequential replace rules. The multi-level system outline shown in Figure 5.19, including two sets of two-level rules, Set A and Set B, is not only possible but is typical of some commercial morphological analyzers that were built at **Xerox** in the early 1990s, before replace rules became available.

In such a system, the upper side of the Set A rules matches the lower side of the LexiconFst; and the lower-side of the Set A rules is not the final surface language but an intermediate language I. Similarly, the upper side of the Set B rules matches I, and the lower side is, finally, the real surface language. So in practice, there may be any number of levels and intermediate languages in a system, even when using the **twolc** rule notation. Of course, once the composition is performed, the result is a single transducer, and all the intermediate languages disappear. To computational linguists who are more impressed by formal power than by the superficial expression of the grammar, the prohibition against multiple levels seems more than a bit religious.

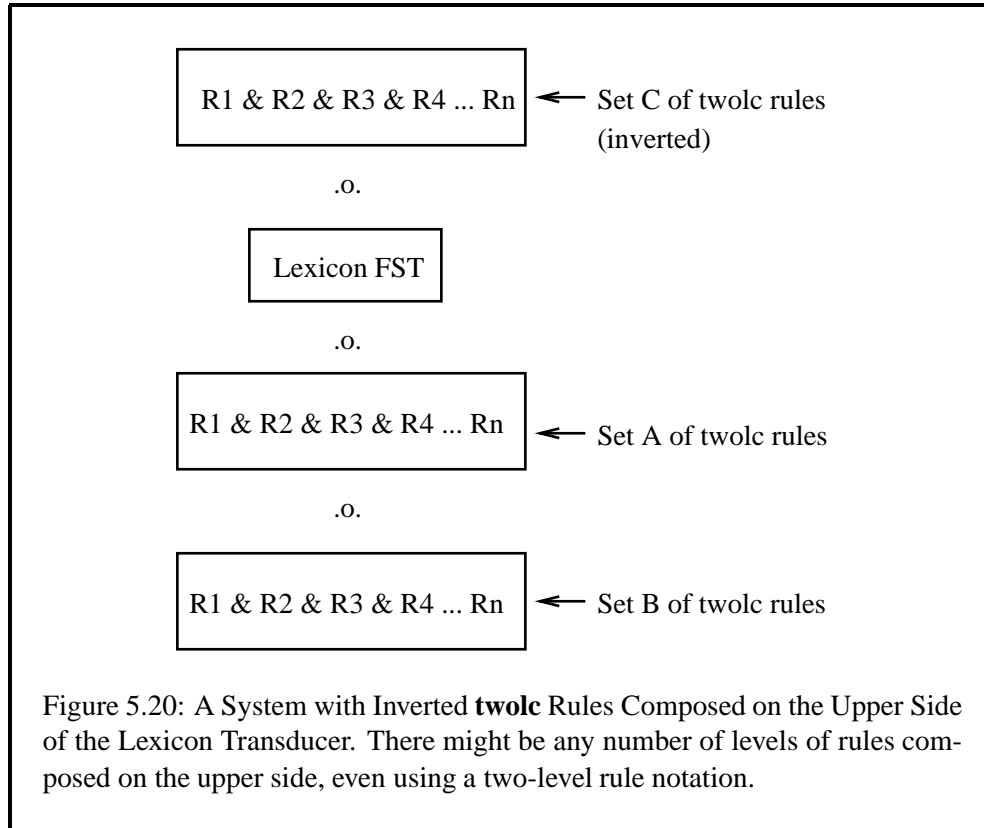
It is often convenient to compose filtering or modifying transducers on the upper side of a lexicon, and in the days before replace rules this was done with



twolc rules as well. However, **twolc** rules assume a downward orientation, with the input on the upper side; so to write rules that map upward from the upper side of the lexicon transducer to a modified upper-side language, the trick is to write the rules (we'll call them Set C) upside down, compile them into a transducer, and then invert the transducer before composition. Again, there may be any number of transducers, compiled from **twolc** rules and inverted, to be composed on the upper side of the lexicon. Such inversions and compositions are easily performed in **xfst**, as shown in Figure 5.20.

Using the **lexc** interface instead of **xfst**, the composition of multiple rule transducers on the lower side of a lexicon, and the composition of inverted transducers on the top of a lexicon, are a bit more difficult. The **lexc** interface has built into it the KIMMO-like assumption that there is but one lexicon, stored in the SOURCE register, and one set of rules, read into the RULES register; and the **lexc** command **compose-result** composes the rules on the lower side of the lexicon and stores the result in the RESULT register.

However, there are some **lexc** interface tricks that allow inversions and multiple compositions. Assuming that Sets A, B, and C of **twolc** rules have been pre-compiled and saved in files `A.fst`, `B.fst` and `C.fst`, and assuming that the lexicon is stored in `lexicon.fst`, the **lexc**-interface idiom for performing



the various inversions and compositions shown in Figure 5.20 is

```
lexc> read-source lexicon.fst
```

```
lexc> read-rules A.fst
```

```
lexc> compose-result
```

```
lexc> result-to-source
```

```
lexc> read-rules B.fst
```

```
lexc> compose-result
```

```
lexc> result-to-source
```

```
lexc> invert-source
```

```
lexc> read-rules C.fst
```

```
lexc> compose-result
```

```
lexc> invert-result
```

Notice here that to compose rules on the upper side of the lexicon, the lexicon is first inverted so that the upper side and lower side are switched. The **twolc**

interface can then compose *C.fst* on the new lower side (old upper side) of the lexicon to create a result that is upside down. The result is then inverted to get the final transducer. This is obviously a bit awkward, and most **Xerox** developers now avoid the **lexc** interface, and very often **twolc** rules themselves, in favor of **xfst** and replace rules.

5.4.3 Moving Tags

Occasionally linguists feel a need to “move” a tag that was most naturally put in an undesirable position by a **lexc** grammar. Such moving is accomplished by introducing a new tag via an epenthesis rule in the desired new location, and then deleting the original tag, mapping it to the empty string.

Such “moving” of tags is not always recommendable, as it may result in the copying of large portions of your networks. The following little **lexc** grammar (*adj-lexc.txt*) illustrates the problem:

```
! adj-lexc.txt

Multichar_Symbols Neg+ +Adj

LEXICON Root
    Adjectives ;

LEXICON Adjectives
    NegPrefix ;
    AdjRoots ;

LEXICON NegPrefix
    Neg+:un AdjRoots ;

LEXICON AdjRoots
    healthy Adj ;
    worthy Adj ;
    likely Adj ;

LEXICON Adj
    +Adj:0 # ;
```

In the **lexc** grammar, the *un-* prefix is naturally paired with the **Neg+** tag on the lexical side, yielding string pairs like the following (showing epsilons overtly as zeros).

```
Upper: healthy+Adj          Neg+0healthy+Adj
Lower: healthy0            u  nhealthy0
```

Now suppose that we really want to have lexical strings that always begin with a root, with all the tags concatenated after it. That is, in the negative case we might want something more like

```
Upper: 00healthy+Adj+Neg
Lower: unhealthy0 0
```

with all the tags, including the +Neg, on the end. This will almost double the size of the transducer, because it creates a separated dependency (see Sections 4.5.3 and 8.3.1), but it makes an instructive exercise.

Tag-Moving Exercise 1

Take the **lexc** grammar as shown above, compile it using **lexc**, and store the binary result as `adj-lexc.fst`. Then “move” the tags by composing **xfst** replace rules on the top. One rule should map a single empty string [...] upwards to +Neg at the very end of the word, if the string is currently marked with a Neg+; and the other should map the original tag Neg+ upwards to epsilon. In essence, the moving of tags will involve the insertion of one tag and the deletion of another. Do the composition on the **xfst** stack, or using regular expressions, arranging the order of the networks carefully. One solution is shown on page 635.

Tag-Moving Exercise 2

Do the same using **twolc** rules and composing the lexicon with the rules inside **lexc**. This will be a bit tricky. **twolc** rules have a built-in downward orientation, and the usual assumption is that they will be composed on the *lower side* of a source lexicon inside **lexc**. To modify the upper side of a transducer, as in this case, the trick is to write the **twolc** rules upside-down. Then, inside **lexc**, read the lexicon network and *invert* it before composing the **twolc** rules (which were written upside-down). Then *invert* the result. The **lexc** interface includes the necessary inversion commands.

```
lexc> read-source adj-lexc.fst
lexc> invert-source
lexc> read-rules my-rul.twol
lexc> compose-result
lexc> invert-result
```

A solution is shown on page 635.

Tag-Moving Exercise 3

Do the same exercise using **twolc** rules, but doing the composition on the **xfst** stack. Compile and intersect the rules in **twolc**, save the resulting network into a file, and load the file into **xfst**. Next, invert the rule network and compose it on upper side of the lexicon using **compose net** utility of **xfst**. A solution is shown on page 637.

The problem with all of these solutions is that the size of the transducer jumps from 17 states, 21 arcs and 6 paths to 31 states, 35 arcs for the same 6 paths. That's a high price to pay for a cosmetic change in the spelling of the lexical strings.

5.5 Debugging *twolc* Rules

5.5.1 Rule Clashes

Because all the rules in a *twolc* grammar apply in parallel, simultaneously, there are abundant opportunities for rules to get in each other's way and conflict. Rule conflicts are discovered and reported by the rule compiler, and some of them are even resolved automatically, with appropriate warning messages. However, all rule conflicts are technically errors, and some of them cannot be resolved automatically, so every *twolc* user needs to understand rule conflicts and learn the idioms for resolving them. As a practical matter, always read the error and warning messages carefully.

There are two basic kinds of rule conflict:

1. Right-Arrow Conflicts
2. Left-Arrow Conflicts

The key to understanding rule conflicts is understanding the semantics of the *twolc* left and right arrows. Review the semantics on page 326 as necessary.

Right-Arrow Conflicts

It is possible for two rules to be in a right-arrow conflict. Right-arrow conflicts are usually benign. Here is a simple example.

```
"Rule 1"
a:b <=> l _ r ;

"Rule 2"
a:b <=> x _ y ;
```

From the right-arrow point of view, Rule 1 states that the pair **a:b** can occur only in the environment `l _ r`; and at the same time, Rule 2 states that the same pair **a:b** can occur only in a different environment `x _ y`. Wherever one rule succeeds, the other will fail; they are in mortal conflict.

Fortunately, such conflicts are resolved quite easily. If both contexts are in fact valid, then simply collapse the two rules into one rule with two contexts.

```
"Rule 3"
a:b <=> l _ r ;
          x _ y ;
```


The **twolc** compiler will then constrain **a:b** to appear *either* in the first context $l - r$ or in the second context $x - y$.

The rule writer may prefer to leave the resolution of right-arrow conflicts to the compiler itself. The **twolc** interface has a special switch that can be toggled off and on with the command **resolve**. The default position is ON. When the **resolve** switch is ON, the compiler looks for and tries to resolve rule conflicts. In the case of a right-arrow conflict, it assumes that the rule writer intends to allow the particular realization in all of the contexts that are mentioned in the rules. In the case at hand, the effect is that both Rule 1 and Rule 2 are in fact compiled as Rule 3, and the compiler prints the message:

```
>>> Resolving a => conflict with respect to 'a:b'
      between "Rule 1"
          and "Rule 2"
```

If the **resolve** flag is OFF and a right arrow conflict is not resolved manually, the effect is that lexical **a** cannot be realized as **b** in either of the two environments.

Right-arrow conflicts occur when two right-arrow or double-arrow rules constrain the same feasible pair to occur in different environments. The conflict is usually resolvable by collapsing the two rules into one rule with two contexts. The compiler will do this automatically by default.

Left-Arrow Rule Conflicts

Left-arrow conflicts are more difficult to detect and resolve correctly. Consider the grammar in Figure 5.21. According to the semantics of **twolc** rule operators, the left-arrow constraint of Rule 4 states that if lexical **a** is between a left context that ends with **l** and a right context that begins with **r**, it must be realized as **b**. Simultaneously, the left-arrow constraint of Rule 5 states that a lexical **a** in this same environment must be realized as **c**. The two rules impose contradictory requirements: wherever Rule 4 matches and succeeds, Rule 5 will fail, and vice versa. Because of the conflict, a lexical **a** has no valid realization in the context $l - r$.

Two rules are in left-arrow conflict when they each impose a left-arrow restriction, have overlapping contexts, and try to realize the same lexical symbol in two different ways.

If the **resolve** switch is ON, the compiler looks for and tries to resolve left-arrow conflicts as best it can. It tries to determine which of the rules has a more specific context. In the case of Rules 4 and 5, there is no difference; the contexts are

```

Alphabet a a:b a:c b c ;

Rules

"Rule 4"
a:b <=> l _ r ;

"Rule 5"
a:c <=> l _ r ;

```

Figure 5.21: Rule 4 and Rule 5 are in Left-Arrow Conflict

identical. In that case, the compiler will arbitrarily give precedence to whichever rule comes first. Here Rule 4 will prevail over Rule 5. The compiler reports:

```

>>> Resolving a <= conflict with respect to 'a:b' vs. 'a:c'
      between "Rule 4"
          and "Rule 5"
      by giving precedence to "Rule 4"

```

In effect, Rule 5 becomes completely irrelevant. In the context `l _ r`, the preferred Rule 4 requires **a** to be realized as **b**. Because the right-arrow restriction of Rule 5 does not allow the **c** realization in any other contexts, the result is that **a** can never be realized as **c**. In a case such as this one, automatic conflict resolution is as good as no resolution at all. There is a logical error that has to be corrected by the rule writer. The only real benefit is that the conflict is noticed and reported.

```

Alphabet a a:b a:c b c ;

Rules

"Rule 6"
a:b <=> _ r ;

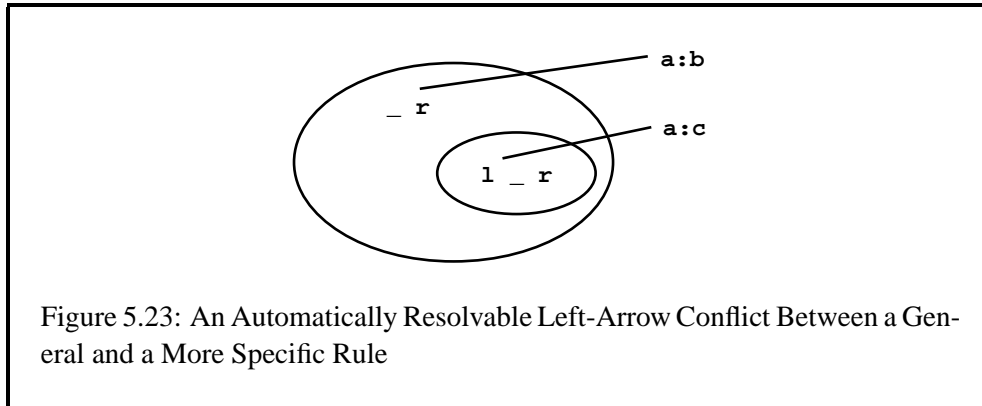
"Rule 7"
a:c <=> l _ r ;

```

Figure 5.22: Rule 6 and Rule 7 are in Left-Arrow Conflict

A more interesting case of a left-arrow conflict is demonstrated by Rules 6 and 7 in Figure 5.22. Rule 6 and Rule 7 have the same right context. Because the left context of Rule 6 is empty, the rule requires that a lexical **a** be realized as **b** in front

of a following **r** regardless of what precedes it on the left. Rule 7 is more specific. It requires **a** to be realized as **c** only when it is both preceded by **l** and followed by **r**. Note that every instance of the context **l _ r** is also an instance of the context **_ r**. Formally speaking, the context of Rule 7 is SUBSUMED by the more general context of Rule 6. Figure 5.23 illustrates the relationship between the contexts of the two conflicting rules.



In such cases, the compiler resolves the conflict by giving precedence to the more specific rule. It interprets the \leq part of Rule 6 as if the rule writer had written it as $a:b \mid a:c \leq _ r$. This does no harm because in a two-level grammar all the rules work together. Rule 6 can be relaxed to allow **a** to have either realization in its context because Rule 7 will prevent the **c** realization outside its own specific context. The combined effect is that **a** is realized as **c** between **l** and **r** and as **b** in all other cases when **r** follows.

In a system of replace rules, there are no conflicts between rules, but the rule writer has to order the rules in a proper way. In particular, a specific rule always has to be ordered before a more general rule. In phonological literature (Chomsky and Halle, 1968) this is called DISJUNCTIVE ORDERING. The way in which the **twolc** compiler resolves left-arrow conflicts is motivated by this tradition.

There are, however, left-arrow conflicts that cannot be resolved in a principled way. When **twolc** discovers and reports an unresolvable left-arrow conflict, read the error message carefully, identify the rules involved, review the semantics of **twolc** left arrows if necessary, and make sure that you understand *why* the rules are in conflict. DO NOT start making changes until you understand the conflict completely. Beginners too often start changing arrows, fiddling with contexts, and flailing about aimlessly. Take the time to understand first.

After you understand what each rule is intended to do, and why they are in conflict, then one of them will need to be changed intelligently to resolve the conflict. If one of the rules is simply in gross error, then fixing it should be easy. Where both of the rules seem right, then the usual problem is that one needs to be made more specific in its contextual requirements. Consider this simple case:

```
"Rule 8"
a:b <=> l _ ;

"Rule 9"
a:c <=> _ r ;
```

Rule 8 and Rule 9 are in left-arrow conflict because both left contexts can match a string ending with **l**, and both right contexts can match a string beginning with **r**. The rules are in mortal conflict with respect to how strings such as “**lar**” should be realized, and the compiler cannot resolve the conflict for you. Very often in such cases, one of the rules is the general case and the other is a more specific exception. If so, the error can be fixed by simply making one of the rules more specific than the other. That is, the context of the more specific rule should be completely subsumed by the context of the more general rule instead of just partially overlapping with it. As we showed above, the compiler then resolves such conflicts automatically.

```
"Rule 8"
a:b <=> :* l _ :* ;

"Rule 9"
a:c <=> :* _ r :* ;
```

Figure 5.24: Automatic Context Extension. The **twolc** rule compiler automatically extends the left context to the left, and the right context to the right, with the universal relation over the alphabet of feasible pairs.

```
"Rule 8"
a:b <=> l _ ;

"Rule 9'"
a:c <=> .#. [ :* - [ :* l ] ] _ r ;
```

Figure 5.25: Resolving a Left-Arrow Conflict, Giving Precedence to Rule 8, by Context Subtraction. Whereas Rule 9 matches anything for the left context, Rule 9' matches any left context, except (minus) left contexts that end with **l**. The subtracted left context, here **l**, might be an arbitrarily complex regular expression.

If this is truly a case of partial overlap between two rules, the rule writer has to decide which of the two rules should apply in the context where they are in conflict. The general idiom for fixing such left-arrow conflicts is called **CONTEXT SUBTRACTION**. In the case above, if we want to give Rule 8 precedence over Rule 9, with respect to strings such as “lar”, we must subtract the left context of Rule 8, here any string ending in **l**, from the left context of Rule 9, which is the universal relation. Recall that the rule compiler automatically extends each context with the universal relation over the feasible pairs of the alphabet, as shown in Figure 5.24. The new non-conflicting version, Rule 9’ in Figure 5.25, has the same right context as the original rule, but the left context is modified by removing, i.e. subtracting, all strings that match the left context of the competing rule. Because the complement of language L , written $\sim L$, is equivalent to $: * - L$, i.e. to the universal relation minus the strings in L , Rule 9’ could be written equivalently as in Figure 5.26.

```
"Rule 8"
a:b <=> l _ ;

"Rule 9'"
a:c <=> .#. ~[:* l] _ r ;
```

Figure 5.26: Resolving a Left-Arrow Conflict, Giving Precedence to Rule 8, by Context Complementation. Whereas Rule 9 matched anything for the left context, Rule 9’ matches any left context that does not end in **l**.

While context subtraction and the equivalent context complementation make perfect sense, they are notoriously difficult for students to grasp. Indeed, even experienced developers have had to reinvent the idiom several times. The key to understanding context subtraction is to remember that rules are compiled automatically with $: *$ added on the left of the left context and $: *$ on the right of the right context, as shown in Figure 5.24 and in Figure 5.18 on page 328. The full left context of Rule 8 is therefore $[: * l]$ and the full left context of the original Rule 9 is $: *$, the Kleene-star relation on the pair alphabet. The expression $[: * - [: * l]]$ in the left context of Rule 9’ subtracts the full left context of Rule 8 from the full left context of the original Rule 9.

Note that the initial **a** in strings such as “ar” should be mapped to **c** by these rules; Rule 9’ must be written to allow the possibility of an empty-string left context. The subtraction would not have the intended effect without indicating the explicit word-boundary marker **.#.** in the left context of 9’ because the relation $[: * - [: * l]]$ includes the empty string. Of course, the equivalent $\sim[: * l]$ also includes the empty string. As we have seen, the compiler always concatenates the relation $: *$ to the left side of whatever the rule writer has specified. The concate-

nation of a Kleene-star relation to anything that contains the empty string is equivalent to the Kleene-star relation itself.¹³ That is, $:* [:* - [:* - 1]]$ and $:* \sim [:* 1]$ denote the same relation as $:*$. Without the initial word boundary $.\#.$ explicitly specified, the left context of Rule 9' would be exactly equivalent to the left context of the original rule. The intent of Rule 9' is to restrict the left context to "all sequences of symbol pairs that do not end with 1". Because this includes the empty context, there is no way to formally express the idea without the boundary symbol.

The boundary symbol $. \# .$ is necessary in all cases where a specific context (some subset of $:*$) includes the empty string. Without the initial $. \# .$, a rule such as $a : c \Leftrightarrow . \# . : * - [: * 1] _$ or $a : c \Leftrightarrow . \# . \sim [: * 1] _$ is vacuous. Without the final $. \# .$ the optional right context in a rule like $a : b \Leftrightarrow _ (r : *) . \# .$ has no effect.

In the example at hand, where the original context consists of just one symbol, it might seem that there is a simpler solution using \backslash , the symbol-complement operator. If, as in Rule 9 Not-Quite-Right, we change the left context of Rule 9 to be $\backslash 1$, that is any single symbol or symbol pair except for **1**, then the conflict is resolved favoring Rule 8.

```
"Rule 8"
a:b <=> 1 _ ;

"Rule 9 Not-Quite-Right"
a:c <=> \1 _ r ;
```

However, whereas the original Rule 9 maps the initial **a** in strings such as "ar" to **c**, the Not-Quite-Right modified version does not; it requires at least one symbol of left context. To get the correct result, we must specifically handle the case of the empty context as shown below.

```
"Rule 9'"
a:c <=> [ . \# . | \1 ] _ r ;
```

All the versions of Rule 9' and 9'' are equivalent. They all require **a** to be realized as **c** at the beginning of a string or after any symbol or symbol pair other than **1**. Note that the backslash idiom, even with the left word-boundary specified, cannot resolve clashes where the left context is over one symbol long. Context subtraction, or context complementation, is the general idiom for such conflict resolution.

¹³See the discussion of the Restriction operator in Section 2.4.1.

Let's look now at the opposite case: If we want to resolve the conflict between Rule 8 and Rule 9 in favor of the latter one, we must subtract the right context of Rule 9 from the right context of Rule 8 as shown in Figure 5.27. The new right context `[: * - [r : *]] . # . ;` denotes any string, except those beginning with `r`. The trailing `. # .` in the modified Rule 8' is necessary for the reasons discussed above.

```
"Rule 8' "
a:b <=> l _ [ : * - [ r : * ] ] . # . ;

"Rule 9"
a:c <=> _ r ;
```

Figure 5.27: Resolving a Left-Arrow Conflict, Giving Precedence to Rule 9, by Context Subtraction of the Right Context. The right context of Rule 8' now matches all possible strings, including the empty string, except for those strings that begin with `r`.

An equivalent result is obtained via context complementation, shown in Rule 8'' in Figure 5.28.

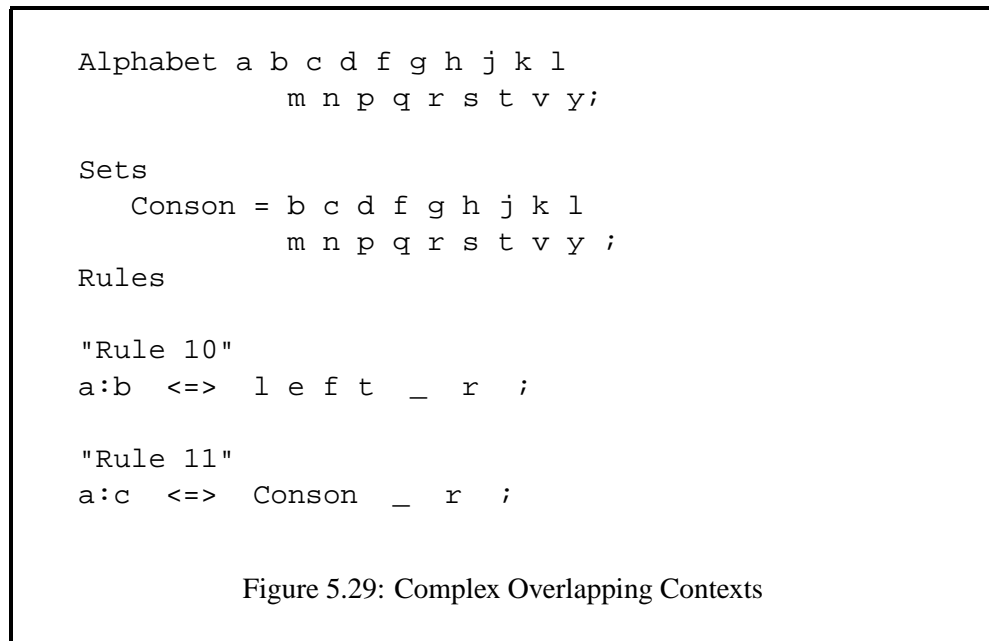
```
"Rule 8'' "
a:b <=> l _ ~[ r : * ] . # . ;

"Rule 9"
a:c <=> _ r ;
```

Figure 5.28: Resolving a Left-Arrow Conflict, Giving Precedence to Rule 9, by Context Complementation of the Right Context. The right context shown here as `r` could be an arbitrarily complex regular expression. The right word-boundary `. # .` must be explicitly indicated as shown.

The context-subtraction idiom also works where the overlapping contexts are more complex, as in Figure 5.29. Let us assume that the intent of the rule writer is that the more general Rule 11 is to apply whenever the left context ends in a consonant, except when the left context ends with the more specific context `[l e f t]`. That is, Rule 10 expresses an exception to the more general rule. The context subtraction idiom works perfectly in this case too, but it is not necessary to use it because the compiler will automatically resolve the conflict in the desired way. That is, it automatically relaxes the left-arrow restriction of Rule

10 to allow a lexical **a** to be realized as either **b** or **c**.



If the rule writer opts for the manual solution by context subtraction, the left context of Rule 11 should be modified as shown below in Rule 11'.

```

"Rule 11'"
a:c <=> .#. [[ :* Conson] - [:* l e f t ]] _ r ;

```

The automatic resolution of the conflict will produce a transducer for Rule 11 that is different from the transducer compiled from the manually modified Rule 11'. However, the intersection of the “relaxed” version of Rule 11 with Rule 10 is identical to the transducer resulting from the intersection of Rule 10 and Rule 11'. That is, considering the rule system as a whole, the automatic resolution of left-arrow conflicts and the manual method of context subtraction produce exactly the same result.

Spurious Compiler Warnings

Although the compiler correctly detects most left-arrow conflicts, it is not perfect. In some cases the compiler issues spurious warnings. A simple example is shown in Figure 5.30. The two rules in Figure 5.30 allow a lexical **a** to be realized as either **b** or **c** but require that whichever way is chosen in a particular case, it must be chosen consistently. That is, “aa” can be realized as “bb” or as “cc” but not as “ac” or “ca”.


```

Alphabet a:b a:c;

Rules

"Rule 12"
a:b <=> ${a:b} _ ;

"Rule 13"
a:c <=> ${a:c} _ ;

```

Figure 5.30: **twolc** is Not Perfect. In spite of warnings from the compiler, there is no real \leq conflict between these two rules.

The compiler compiles the rules correctly but gives a spurious warning about a left-arrow conflict.

```

*** Warning: Unresolved <= conflict with respect to
    'a:b' vs. 'a:c'
    between "Rule 12"
        and "Rule 13"

**** Conflicting pair(s) of contexts:
$a:b _ ;
$a:c _ ;
Left context example:    a:c a:b

```

The reason is that the compiler cannot “see ahead” what effects the rule or rules it is working on are going to have at the end. At the point where the compiler is comparing the context where **a** must be realized as **b** and the context where **a** must be realized as **c**, it sees a potential conflict in the case where a string containing two **as** has been seen and one of the **as** has been realized as **b** and the other as **c**. At the point where the compiler is comparing the two rules, it cannot yet deduce that the combined effect of the two rules is to disallow the problematic case.

In real life, examples of this sort are very rare. Because the message includes an example of the problematic context, the rule writer can decide whether to take the warning seriously or to ignore it. In our experience, if the compiler warns about an unresolved left-arrow conflict, it is nearly always right.

Why Context Subtraction Works

The context-subtraction and context-complementation idioms perform subtraction and complementation on relations, which is not generally possible. As pointed out

above on page 328, the equal-length relations denoted by **twolc** rules are a special case.

From another point of view, the transducers of **twolc** and two-level morphology in general are reduced to simple **FSMs** denoting languages that consist of strings that in turn consist of letters like a:a, b:b, y:0, 0:i, etc. The basic alphabet always consists of such feasible pairs. The zeros in such pairs are the “hard zeros” of **twolc**, as real as **b** and **c**, and not equal (inside rules) to the empty string. The transducers of **twolc** and two-level morphology are therefore really one-level networks that are cleverly interpreted as if they were transducers.

5.5.2 Epenthesis Rules

Epenthesis or insertion rules are a challenge for any grammar, and they are surprisingly hard to get right in a finite-state system (where your rules are compiled and performed literally by a computer). Consider the following bad **twolc** rule:

```
Alphabet  0:h  %+Neg:0 ;

Rules

! this is a bad epenthesis rule

"Lenition"
0:h  <=>  .#.  [ b | t | s ] _  :*  %+Neg: ;
```

The intent of this rule is to perform the following mapping:

```
Lexical:  b0riseann+Neg
Surface:  bhriseann0
```

where “briseann+Neg” is a string from the lexicon. As usual, **twolc** rules will match and impose their constraints everywhere they can. The problem is that the colon (:) matches any feasible pair in the alphabet, including 0:h, so the left-arrow portion of the rule continues to match and tries to insert an infinite number of 0:h pairs.

```
Lexical:  b0000...0000riseann+Neg
Surface:  bhhhh...hhhriseann0
```

However, the right-arrow restriction of the same rule prohibits any 0:h insertions that are not immediately after an initial consonant, and so the rule is in conflict with itself.

One way to fix this particular rule is to specify something more specific than just :* as the start of the right context, e.g.

```
"Lenition"
0:h <=> .#. [ b | t | s ] _ \0:h* %+Neg: ;
```

This new rule specifies that `0:h` is inserted after a **b**, **t** or **s** at the beginning of a word, where the right context consists of any number of symbols that *are not* **0:h** and then a lexical `+Neg`. This prevents the infinite insertion and allows the intended insertion of a single **0:h**.

A similar problem occurs whenever a left or right context is left out entirely, e.g.

```
"bad epenthesis rule"
0:i <=> c k _ ;
```

This rule is written so that any right context will match, so if the intent is to do the following

```
Upper: rack0
Lower: racki
```

`0:i` is also a valid right context for the rule, and it will match and require another insertion.

```
Upper: rack00
Lower: rackii
```

And in fact it will match repeatedly and require an infinite number of insertions, but once again the rule will be in conflict with itself.

```
Upper: rack000...00000
Lower: rackiii...iiiiii
```

The way to fix the rule is to specify a right context that blocks the infinite insertion.

```
"good epenthesis rule"
0:i <=> c k _ [ \0:i | .#. ] ;
```

A similar problem is found in **xfst** replace rules (see Section 3.5.5, page 179), where the bad epenthesis rule

```
! this is a bad xfst epenthesis rule

[] -> i || l _ r ;
```

would try to insert an infinite number of **is** in between **l** and **r** in the following example:

```
Lexical:  l000...0000r
Surface:  liii...iiiiir
```

An infinite number of empty strings exist between **l** and **r** on the lexical side, and the rule tries to map them all into **i**.

For replace rules, the solution is provided in the [...] or “dotted-bracket” operator, which constrains strings to be interpreted as having exactly one empty string between each symbol. The following replace rule has exactly one output.

```
[...] -> i | l _ r
```

```
Lexical:  l0r
Surface:  lir
```

5.5.3 Diacritics

In a **twolc** file, an optional section allows you to declare diacritical characters, e.g.

```
Diacritics    %^FOO    %^DOUBLE    %^DEL    ;
```

Of all the features of **twolc**, the Diacritics have always been the worst documented and the most poorly understood. They also complicate the methods described above to resolve rule clashes. Diacritics are now a deprecated feature of **twolc**, but they are described here for completeness.

twolc Diacritics should not be confused with the Flag Diacritics described in Chapter 8.

The Raw Facts about Diacritics

When a **twolc** grammar has Diacritics declared, the rules in the grammar are compiled in such a way that

- The Diacritics are always mapped to epsilon (zero) on the lower side, and
- The rules, by default, simply ignore the presence of the Diacritics in the lexical strings

However, if a rule explicitly “mentions” or refers to a Diacritic in its context, then that rule is compiled in such a way that it “notices” that Diacritic symbol.

The notion of Diacritics applies only in **twolc** files, and a declaration of Diacritics only affects the way in which the rules of the file are compiled into transducers. Inside the grammar as a whole, the Diacritic symbols are just symbols, typically multicharacter symbols, like any other.

The Motivation for Diacritics

To understand **twolc** diacritics, recall that the classical approach to finite-state morphology is to build a lexicon, typically using the **lexc** language, and to write a separate grammar of rules, typically using **twolc**, to map the abstract strings from the lexicon into surface strings. The rules are traditionally composed on the lower side of the lexicon such that the lower-side language of the lexicon is the upper-side language of the **twolc** rules.

In a lexicon, there is often a need to inject a feature symbol into certain strings which eventually, when the **twolc** rules are applied, either triggers the firing of a particular rule or blocks the firing of a particular rule. Such a symbol automatically becomes part of the alphabet of the lexicon, and normally all **twolc** rules would have to be written to use or at least allow the presence of such feature symbols in the strings.

However, writing all the rules to be aware of all the features is often not desirable, especially if only a couple of rules need to be aware of the feature symbols. The problem is especially acute in a relatively mature lexicon where the linguist feels a need to inject some new feature characters to capture behaviors that were not anticipated at the beginning. The injection of new feature symbols in certain strings may require the review and rewriting of all the rules in the grammar to allow the presence of the new symbols.

The motivation behind Diacritics was to let linguists inject feature symbols freely into their lexical strings (typically in a **lexc** program), but to allow them, in **twolc**, to declare such characters as Diacritics so that they would be ignored by all rules except those rules that overtly refer to them.

Noticing and Ignoring Diacritics

twolc rules are compiled, by default, in such a way that they ignore the symbols declared to be Diacritics. There are in fact two ways to override the default:

- Any rule that specifically mentions a Diacritic symbol in its context will be compiled in such a way that it notices (i.e. does not ignore) that symbol.
- Another overt way to force a particular rule to notice or pay attention to a Diacritic is to include an Attention statement immediately after the rule name.

The following example shows the use of the Attention statement, where the word *Attention* is followed by an equal sign and a list of Diacritic symbols (one or more) to be noticed.

```
Diacritics  %^MF  %^SG  %^PL  ;
```

```
Rules
```

```
"my rule"
Attention = ( %^MF )

0:x <=> l _ r ;
```

Such a rule would normally be compiled in such a way that it ignores completely the symbol `^MF`, declared in the same file to be a Diacritic; but with the overt `Attention` statement, the default is overridden and the rule will be compiled so that it treats `^MF` like any other symbol.

Diacritics and Rule Conflicts

Although the automatic resolution of left-arrow conflicts usually works correctly even when diacritics are involved, they present difficult problems if the conflict has to be resolved by the brute-force context-subtraction method. This is one of the major reasons why Diacritics are currently deprecated.

Most **twolc** writers eventually need context subtraction (see Section 5.5.1, page 356) to clean up rule conflicts in their grammars. If you feel a strong need to use context subtraction and Diacritics at the same time, you can often get away with it by overtly forcing the rules in conflict to notice all the declared Diacritics, as in the following example.

```
Alphabet A:0 ; ! a token alphabet declaration

Diacritics %^FOO ; ! rules will ignore ^FOO by default

Rules

"Rule 1"
Attention = ( %^FOO )
a:e <=> .#. ~[:* y] _ z ; ! with Diacritics declared,
! and Rule 1 forced to pay
! attention to them, the
"Rule 2" ! rules are no longer in
Attention = ( %^FOO ) ! conflict.
a:o <=> y _ z ;
```

5.5.4 Classic twolc Errors

When writing **twolc** grammars, beginners often fall victim to some classic errors. The following points are worth reviewing.

1. You should always start a **twolc** analysis by writing out numerous two-level examples with the lexical string above the surface string, and with **twolc** hard zeros inserted at appropriate places to pad out the strings to the same length. The failure to “line up your strings”, to document the lineup, and to be consistent in the lining up of your examples is a fundamental methodological

error. Two-level rules constrain the appearance of feasible pairs in two-level strings, and you cannot write a workable set of rules unless you know what you're trying to accomplish.

2. **twolc** rules map strings from an upper level to a lower level, and the upper level of the rules is typically the lower level of the lexicon. Sometimes, linguists write multiple levels of **twolc** rules, mapping through a cascade of levels. The failure to plan and understand your own levels is a fundamental methodological error. You should think of each level of intermediate strings as a language and be able to characterize what the strings look like at each level.
3. The **twolc** compiler prints out error messages and warnings when it compiles a source file. Beginners tend to ignore the messages; experts study them carefully, fixing syntactic errors and rewriting rules to resolve conflicts that are not resolved automatically by the compiler.
4. **twolc** rules are deontic statements of fact that constrain where particular feasible pairs can appear in a pair of lexical/surface strings. A simple rule states constraints on a single feasible pair like **x:y**, while a compound rule with where clauses is just an abbreviation for multiple single rules that constrain a single feasible pair.

```
"unique rule name"
r:l <=> i _ .#. ;

! A typical rule constrains where a feasible pair
! like r:l can appear in a valid pair of lexical-
! surface strings
```

Attempts to make **twolc** rules map strings of symbols into strings of symbols, which is possible in replace rules, is a common error that betrays a poor grasp of the syntax and semantics of **twolc** rules. The following **twolc** rule, which might appear to map +Participle into the symbols **i**, **n** and **g**, will in fact implicitly declare a single multicharacter symbol named **ing** and constrain where the feasible pair **%+Participle:ing** can appear.

```
! declaring multicharacter symbols like ing is
! almost always an error

"bad ing rule"
%+Participle:ing <=> l _ r ;
```

In **twolc**, multiple letters written together, such as **%+Noun** or **%+Pl** or **ing**, are automatically interpreted, and implicitly declared, as single symbols with

multicharacter print names. Where these names contain special characters like the plus sign or the circumflex, the special characters must be literalized with the preceding percent sign.¹⁴ When the intent is to represent a genuine string of concatenated alphabetical characters such as **i** concatenated to **n** concatenated to **g**, it is almost always a mistake to use a single multicharacter symbol such as **ing** in **twolc**.

5. **twolc** always works within a fixed alphabet of feasible pairs, and in the earliest versions of the language the linguist was forced to declare all the feasible pairs overtly in the Alphabet section. This was tedious and led to mysterious errors when the linguist forgot to declare simple identity pairs such as **t:t** or **u:u**.

In an attempt to remedy this problem, the current versions of **twolc** automatically assume that identity pairs such as **a:a**, **b:b**, **t:t**, etc. are in the alphabet, *unless* the overtly “mentioned” symbols suggest otherwise. For example, if the user overtly declares the pair **a:e** or writes a rule to control **a:e**, or even writes a rule where **a:e** appears in the context, then the compiler will assume that **a** can appear only on the lexical side and that **e** can appear only on the surface side. That is, “mentioning” **a:e** will override the default assumption that **a:a** and **e:e** are in the alphabet; if the linguist wants to retain **a:a** and **e:e** after declaring **a:e**, then they will have to be declared overtly in the Alphabet section.

It is an open question whether the new alphabetic assumptions are any better than the old ones. In any case, the failure to understand what the alphabet is leads to many mysterious errors for beginners. Be aware that alphabetic symbols can be declared overtly, in the Alphabet section, or implicitly via their use in rules.

6. Often a linguist will declare feasible pairs like **%+Pl:s** and **s:s**, intending to limit **%+Pl** only to the upper-side rule language. Then, unwittingly, the linguist will write a rule like the following,

```
0:e <=> z: _ %+Pl ;
```

intending to insert a surface **e** into plurals like the Spanish “vez+Pl: veces”. However, the right context of the rule, written simply as **%+Pl**, is automatically interpreted as **%+Pl: %+Pl**, and this feasible pair is then added automatically to the alphabet. What the linguist should have written is one of the following:

```
0:e <=> z: _ %+Pl: ;
```

¹⁴Note that special characters in **twolc** cannot be literalized by surrounding them in double quotes, which is possible in **xfst** regular expressions.


```
! or

0:e <=> z: _ %+P1:s ;
```

This common error does not show up in error messages because the compiler has no way of knowing whether you really mean to declare `%+P1:%+P1` or not. The typical sign of this problem is when you generate, via **lex-test**, a string like “vez+P1” and you get multiple output strings like “vece+P1”, “vezs” and “vez+P1” instead of the single solution “veces” that was expected. Experts learn to expect and check for problems in the **twolc** alphabet.

7. Finally, the little where clauses that define rule-scoped variables cause their share of problems. Linguists often write a rule like the following

```
"rule with a where clause"
XX:YY <=> l _ r ;
           where XX in ( i e )
                 YY in ( u o ) matched;
```

and then try to add a new context on the end of the rule, e.g.

```
! the where clause must appear after
! all the contexts
```

```
"rule with a where clause"
XX:YY <=> l _ r ;
           where XX in ( i e )
                 YY in ( u o ) matched;
m _ n ;    ! ERROR, ERROR
```

However, if a where clause appears, it must be after the very last context. Unfortunately, the **twolc** rule compiler does not give an error message but simply stops parsing at the added context and ignores the remainder of the file. Once again, it is important to look carefully at the messages returned by the compiler, including the listing of the rules that were successfully compiled.

5.6 Final Reflections on Two-Level Rules

5.6.1 Upward-Oriented Two-Level Rules

It will be noticed (see Table 5.1, page 326), that the right-arrow and left-arrow restrictions of two-level rules are not symmetrical. The right-arrow => restriction, as in

$$a:b \Rightarrow l _ r ;$$

states that the symbol *pair* **a:b** is restricted to appear in the indicated context; the rule will therefore block the appearance of the pair **a:b** in any other context. Notice that there is no directionality in the right-arrow restriction—it is the pair **a:b** that is restricted to appear in a certain context or contexts.

In contrast, the left-arrow \Leftarrow restriction has a clear downward orientation.

$$a:b \Leftarrow l _ r ;$$

It states that if the lexical symbol **a** occurs in the indicated context, then this **a** must be realized on the surface as **b**. Any other realization of lexical **a** in the indicated context or contexts is blocked by the rule. Significantly, the rule does *not* say that a surface **b** in the indicated context must be a realization of lexical **a**; the **twolc** left-arrow restriction is downward oriented, from lexical to surface.

The double arrow \Leftrightarrow straightforwardly combines the non-directional right-arrow constraints and the downward-oriented left-arrow constraints.

The downward-orientation of the \Leftarrow restriction suggests that one might define a new kind of two-level rule with an upward-oriented semantics. Let the left arrow with a single bar \Leftarrow be a new operator such that the rule

$$a:b \Leftarrow l _ r$$

states that if the surface symbol **b** appears in the indicated context, then it must be mapped to **a** on the lexical side; the rule would therefore block any other lexical source for surface **b** in the indicated context.

Such upward-oriented rules can certainly be compiled by hand and included in a set of two-level rules, but to our knowledge such upward-oriented two-level rules have never been supported by automatic rule compilers or seriously investigated for their usefulness.

5.6.2 Comparing Sequential and Parallel Approaches

The application of a set of replace rules to an input string involves a cascade of transductions, that is, a sequence of compositions that yields a relation mapping the input string to one or more surface realizations. The application of a set of **twolc** rules involves a combination of intersection and composition.

A set of rules of either type can in principle always be combined into an equivalent single transducer. Replace rules are merged by composition, **twolc** rules by intersection. The final outcome can be the same, as the following example shows.

Yokuts vowels (Kisseberth, 1969) are subject to three types of alternations. (The period, \cdot , is used here to indicate that the preceding vowel is long.)

- Underspecified suffix vowels are rounded in the presence of a round stem vowel of the same height: $dub+hIn \rightarrow dubhun$, $bok'+Al \rightarrow bok'ol$.

- Long high vowels are lowered: $?u.t+It \rightarrow ?o.tut$, $mi.k+It \rightarrow me.kit$.
- Vowels are shortened in closed syllables: $sa.p \rightarrow sap$, $go.b+hIn \rightarrow gobhin$.

Because of examples such as $?u.t+hIn \rightarrow ?othun$, the rules must be applied in the order shown. Rounding must precede lowering because the suffix vowel in $?u.t+hIn$ emerges as u . Shortening must follow lowering because the stem vowel in $?u.t+hIn$ would otherwise remain high giving $?uthun$ rather than $?othun$ as the final output.

Kisseberth's analysis of this data can be formalized straightforwardly as regular replace expressions in **xfst**.

```
define Cons [b|d|f|g|h|k|k'|l|m|n|p|q|r|s|t|v|w|%?];
define Rounding [I -> u | u $%+ _ , ,
                 A -> o | o $%+ _ ];
define Lowering [i -> e, u -> o | _ % . ];
define Shortening [% . -> 0 | _ Cons [Cons | .#.]];
define Defaults A -> a, I -> i, %+ -> 0;
```

The composition [Rounding .o. Lowering .o. Shortening .o. Defaults] yields a 63-state single transducer that maps Yakut lexical forms to surface forms, and vice versa.

The same data can also be analyzed in **twolc** terms. The **twolc** Shortening and Lowering rules are simple notational variants of their **xfst** counterparts. The **twolc** Rounding rule, however, is different from its **xfst** counterpart in one crucial respect: the left context of the alternation [Vz: \$%+:] requires a rounded lexical vowel without specifying its surface realization. Thus the lowering of the stem vowel in $?u.t+hIn$ does not prevent it from triggering the rounding of the suffix vowel. This is one of the many cases in which *under-specification* in two-level rules does the same work as rule ordering in the old Chomsky-Halle phonological model.

Alphabet

```
%+:0 %.:0 % . i i:e u u:o o e i I:u I:i A:a A:o a
b d f g h k k' l m n p q r s t v w %? ;
```

Sets

```
Cons = b d f g h k k' l m n p q r s t v w %? ;
```

Rules

"Rounding"

```
Vx:Vy <=> Vz: $%+: _ ; where Vx in (I A)
                          Vy in (u o)
                          Vz in (u o)
                          matched;
```

"Lowering"

```
i:e | u:o <=> _ % . ;
```

"Shortening"

```
%.:0 <=> _ Cons [Cons | .#.];
```

The intersection of the **twolc** rules for Yakut also yields a 63-state transducer that is virtually identical to its **xfst** counterpart. The only difference is that the two rule systems make a different prediction about the realization of hypothetical Yakut forms such as *?u.t+hI.n* in which the underspecified suffix vowel is long. In this case, the **xfst** Yakut rules predict that it would undergo all three processes yielding *?othon* as the final output whereas the **twolc** Yakut rules produce *?othun* in this case. With a little tinkering, it would be easy to make two rule systems completely equivalent, if one knew what the right outcome would be in that unattested case.

From a mathematical and computational point of view, **twolc** and **xfst** rule systems are equivalent in the sense that they describe regular relations. The sequential replace rules and the parallel **twolc** rules decompose, in a different way, a complex relation between lexical and surface forms into smaller subrelations that we can understand and manipulate. They represent different intuitions of what the best way is to accomplish the task. In practice, the two approaches are often combined, for example, by composing a set of intersected **twolc** rules with networks that represent constraints, replace rules, or another set of intersected **twolc** rules.

Depending on the application, one approach may have an advantage over the other. A cascade of replace rules is well-suited for cases in which an upper language string is typically very different from its lower language counterpart. In such cases we are often interested only in string-to-string mappings and do not care about how the strings are exactly aligned with respect to each other. But if it is important to describe the relation in terms of exact symbol-to-symbol correspondences, **twolc** rules may be a more natural choice because that is precisely what they were designed to do.

Part III

Finite-State Systems

Chapter 6

Planning and Managing Finite-State Projects

Contents

6.1	Infrastructure	377
6.1.1	Operating System	377
6.1.2	Hardware	377
6.1.3	Software	377
	License to Xerox Finite-State Software	377
	Protecting Your Source Code	378
	Backups	378
	Version Control	378
	Makefiles and System Charts	379
	Makefiles	379
	System Charts	379
6.2	Linguistic Planning	380
6.2.1	Formal Linguistics	380
6.2.2	Programming Decisions	381
	Tool Choice	381
	Modularity	381
	Choosing Lexical Baseforms	382
	Multicharacter-Symbol Tags and Tag Orders	383
	Choosing Tags	383
	Tag String Orders	383
	The Lexical Language is Defined by the Linguist	383
6.2.3	Planning vs. Discovery	384
6.2.4	Planning for Flexibility	384
	Thinking about Multiple Final Applications	384

One Core, Many Modifications	385
Dialects	385
Spelling Reforms	387
Vulgar, Slang, Substandard Words	388
Handling Degraded Spelling	389
Specialized Extra Words	389
6.3 Composition is Our Friend	390
6.3.1 Modifying Strings	390
6.3.2 Extracting Subsets of a Transducer	392
6.3.3 Filtering out Morphotactic Overgeneration	394
6.4 Priority Union for Handling Irregular Forms	397
6.4.1 Irregular Forms	397
6.4.2 Handling Irregularities in lexc and Rules	398
6.4.3 Handling Exceptions via Union and Composition	401
Handling Extra Plurals	401
Handling Overriding Plurals with Composition, then Union	404
Handling Overriding Plurals via Priority Union	406
6.5 Conclusions	407
6.5.1 Take Software Engineering Seriously	407
6.5.2 Take Linguistic Planning Seriously	407
Handling Irregular Forms	407
Customization and Multiple Use	407

Writing a finite-state system such as a morphological analyzer demands the same planning and discipline as for any large software-development project. Every experienced software developer can tell horror stories about inadequate hardware, code being lost and corrupted, and the nightmare of maintaining poorly designed and undocumented systems.

From the software-engineering point of view, the finite-state developer faces a bewildering array of choices in how the project is divided into modules and how the work should be divided up among **lexc**, **xfst**, and perhaps even **twolc**. From the strategic point of view, it may appear that choices must be made concerning orthography, dialect, register and even political correctness; but there are techniques that allow a single flexible core system to be modified via finite-state operations to serve multiple ends. Finally, the expert use of finite-state operations like union, composition and priority union can simplify grammars and allow cleaner handling of grammatical irregularity.

This chapter will discuss hardware and software infrastructure, the kinds of software and linguistic planning that should be done *before* coding, and some expert idioms that keep your grammatical descriptions simpler while making them maximally flexible.

6.1 Infrastructure

6.1.1 Operating System

The **Xerox** Finite-State Tools licensed with this book are compiled for the Solaris, Linux (both Intel and PowerPC), Windows (both NT and 2000) and Macintosh OS X operating systems.

6.1.2 Hardware

One can learn and experiment with the **Xerox** Finite-State Calculus using almost any modern computer that runs one of the supported operating systems. However, large-scale grammars can easily compile into finite-state networks that tax memory resources.

The problematic morphological phenomena that often result in large networks include productive compounding (as in German and Dutch), infixation (as in Tagalog), interdigitation (as in Hebrew and Arabic) and separated dependencies between morphemes in a word. The intersection and composition operations, in general, can result in huge networks or require large amounts of intermediate processing memory before minimization can be invoked. The **Xerox Research Centre Europe** maintains some huge workstations, including a SUN Ultra with 2 GBytes of RAM, to handle the most intensive finite-state calculations for commercial systems.

Having sufficient RAM is often a key to successful large-scale development. The **Xerox** finite-state algorithms usually run very fast in RAM, but performance degrades sharply if RAM becomes exhausted and the algorithm must start accessing disk memory (known as “hitting the disk”). When the RAM threshold is crossed, sometimes in the middle of a development project, compilations that once took minutes may suddenly start taking hours. While it may be possible to restructure the offending compilation in such a way as to avoid crossing the threshold, often the only practical solution is to move to a bigger machine or to add RAM to the old one.

6.1.3 Software

License to Xerox Finite-State Software

The **Xerox** finite-state software tools are licensed for non-commercial use under the conditions specified on the license agreement. Requests for commercial licensing are reviewed and negotiated on a case-by-case basis.¹

¹See <http://www.fsmbook.com/> for the latest information on commercial licensing.

Protecting Your Source Code

Your **lexc**, **twolc** and **xfst** files, plus makefiles and documentation, will often represent a great deal of your work and deserve protection. The long-term success and maintainability of any significant project depends on regular backups and careful use of version control.

Backups Any mature software laboratory has a regular backup procedure allowing lost files to be restored. Files can be lost at any time through human error, sabotage, operating-system bugs, disk crashes, power outages, fires, lightning strikes, earthquakes and any number of other disasters.

Without backups, it is all too easy to lose weeks or months of work in a fraction of a second. Experienced developers have all lost work this way, and they have learned the hard way to insist on backups. Intelligent beginners learn the same lesson by example rather than through their own painful experiences.

Talk to your system support group about backups at your site. If there is no system support group, or if their backup procedures are inadequate and cannot be improved, then make your own backups. It's a nuisance, but someday you will thank yourself.

At major stable points in your development cycle, especially when you reach a contractual milestone or make any kind of delivery, the full state of your system should be archived, and copies should be stored in secure areas away from your development site, such as a safety-deposit box in a bank.

Version Control The use of a version-control system like CVS² is always recommended, and it is vital if two or more people have write-access to the same files. An inherent danger in team development is that two developers will try to edit the same file at the same time; each edits a copy in an edit buffer, and when they save the buffer to file they destroy each other's corrections and changes.

Version-control systems impose team discipline, typically requiring a developer to CHECK OUT a file or whole directory for editing, and to CHECK IN³ the file or directory when he or she is finished. During the time that a file is checked out, it remains locked and unchangeable by the rest of the team. CVS is somewhat less strict, allowing multiple researchers to check out copies of the same files, but it keeps strict track of changes and signals you when your changes need to be merged with someone else's.

Each time a file is checked in, CVS saves a record of the changes that were made. This allows CVS to backtrack and restore any previous checked-in version of a file. At major stable milestones, and especially for delivered systems, the developers should take a CVS snapshot that records the state of all the source

²<http://www.cvshome.org/>

³The terminology of *check out* and *check in* comes from libraries. When a patron checks out a book from the library, he or she has control of it until it is checked back in, after which someone else can check it out.

files under version control. There are three main scenarios where version-control systems come in handy.

- If you start on a development path, editing a file, but then change your mind and want to start over, CVS can restore a previous version of your file. It is therefore good practice to check in a stable file just *before* starting any new and dangerous editing. In general, think of version control as a way to save software versions that you might want to return to.
- When you are testing a new stable version of your overall system, you should always compare it against the previous stable version of the same system. If you faithfully took a snapshot of the previous version, CVS can restore it for your tests.
- If you have delivered a version of your project to any other organization or development team, they may come back to you months or weeks later with bug reports and requests for changes. You may need to restore a copy of the version that was delivered in order to confirm the bugs.

Experienced software developers use and appreciate version control, even if it does impose some overhead in checking out and checking in files.

Makefiles and System Charts

Typical lexical transducers are made from multiple **lexc**, **twolc** and **xfst** source files, and the resulting networks are unioned, composed and intersected in various ways to make the final network. While recompiling and re-assembling a whole system can theoretically be done by hand, in practice the only way to ensure reliability is to put all the commands in a makefile.

Makefiles Makefiles are organized in terms of goals, with the top-level goal being the production of the file that contains the final lexical transducer. A makefile typically contains many subgoals that must be produced on the way to the final goal, including the appropriate compilation of all the component files and calculation of intermediate results. In a properly written makefile, each goal is associated with all of the subgoals or source files on which it depends, and the `make` utility is intelligent enough to recompute a goal only if one of its subgoals has changed. Whenever you edit and save any source file of your system, you should be able simply to enter `make` to have all (and only) the affected parts of your overall system recompiled. See Appendix C for an introduction to makefiles.

System Charts While a properly written makefile will automate the recompilation and re-assembly of your lexical transducers, experience has shown that finite-state product developers quickly lose track of the overall organization of their system modules unless they are visualized graphically. This problem is particularly

acute if the developer can't write, or perhaps even read, the makefile himself or herself. It is therefore highly recommended that the components and operations of the makefile be plotted graphically as boxes on a chart and that the chart be taped prominently to the wall for easy reference during development. Such a chart should be as large as necessary, showing explicitly how the system modules are compiled, unioned, intersected and composed together.

The wall chart is especially valuable when tracing bugs, and, of course, when understanding and modifying the makefile itself. It is up to the developer to keep the makefile and the wall chart in sync.

6.2 Linguistic Planning

lexc, **twolc** and **xfst** are tools that allow the developer to state formal linguistic facts in notations that are already familiar to formal linguists. Thus **lexc** gives us a framework for specifying morphotactic facts in terms of sublexicons, entries and continuation classes; and **twolc** and rewrite rules provide a couple of ways to specify facts of phonological or orthographical alternation. But these tools are only frameworks for the formalization of facts; they cannot tell you what the facts are. The discipline that examines natural languages, discovering their rules and formalizing them, is FORMAL LINGUISTICS.

6.2.1 Formal Linguistics

When we refer to *linguists* in this book, we mean *formal* linguists, typically people who have been trained at the university level in formal linguistic description. In common parlance, a linguist is anyone who speaks multiple languages, i.e. a polyglot, which is something quite different. The ability to speak multiple languages, though admirable, doesn't make one a formal linguist any more than having a heart makes one a cardiologist.

A great deal of good formal linguistics was done before computerized tools were available, and a great deal of bad formal linguistics can be done even with the best computational tools. It is important to realize that coding in **lexc**, **twolc** and **xfst** is not linguistics per se and that the formalization of bad linguistic models simply results in bad programs. Formal linguistics is what it always has been, the formal study and understanding of linguistic structures; and that study and understanding should come first. After the formal linguistics is done, after a model is worked out, we implement and test the model using the finite-state tools.

The lesson is this: study the language and do some old-fashioned pure linguistic modeling before jumping into the coding. Your programs will never be better than the linguistic model behind them.

6.2.2 Programming Decisions

Experienced finite-state developers understand that most programming projects can be divided up in many ways, and this is especially the case when one has a choice of three tools, **lexc**, **xfst** and **twolc**, whose capabilities overlap. A little planning, in the form of software engineering, is always recommended.

Tool Choice

lexc, **xfst** and **twolc** are all notations that compile into finite-state networks, so a developer must often choose which tool to use to define a particular network. **lexc** is convenient for defining natural-language lexicons and morphotactics; it is optimized to handle the typically huge lexical unions efficiently, and the continuation classes provide a convenient notation for encoding general compounding and looping. **lexc** also provides convenient notations for handling irregularities, including the gross irregularities called SUPPLETIONS. Anything that can be encoded in **lexc** can theoretically be encoded in **xfst**, using the regular-expression compiler, but with some practical limitations:

1. **xfst** is *not* optimized, as **lexc** is, to handle huge unions efficiently, and
2. regular expressions that encode general compounding and looping can be very hard to write and to read

For these reasons, **lexc** is the preferred choice for defining lexicons and morphotactics.

Alternation rules, which also compile into networks, can be written in either **twolc** or in **xfst**. At **Xerox**, **twolc** has fallen out of use, with most developers preferring to write cascades of replace rules in **xfst**.

Modularity

Depending on your language, you may want to keep verbs, nouns, adjectives, etc. in separate **lexc** lexicons and compile them into networks separately. The corresponding verb, noun, adjective, and other networks may subsequently be unioned together before alternation rules are composed, or you may need to write and apply at least some rules that are specific to particular categories. For languages with productive compounding, where adjective, noun and verb roots attach to each other, all the compounding categories will have to be kept together in the same **lexc** source file.

For alternation rules, you may want to organize your rules into several layers, creating various intermediate languages between the lexical and surface levels (see Section 5.4.2). You will almost certainly want to apply little cleanup transducers to the lexical side, and perhaps to the surface side, of your final lexical transducer. It is impossible to lay down rigid rules, because languages differ so much, but some

decent advance planning can save much rewriting and reorganization of the system during development.

Whenever possible, formal linguistic studies and descriptions should be presented to other linguists for comment before you jump into the coding. Part of the plan should include a wall chart (see page 379), a graphic display of the various components of the system and how they will be combined together.

Choosing Lexical Baseforms

By convention, **Xerox** lexical transducers have lexical (upper-side) strings that consist of BASEFORMS followed by multicharacter-symbol tags. The baseform itself is the headword used conventionally when looking up the surface words in a standard printed (or perhaps online) dictionary. For example, a Spanish verb lemma consists of perhaps 300 different surface forms, but only the infinitive form, the conventional baseform, appears as a headword in dictionaries. Thus *canto* (“I sing”) is analyzed as a form of *cantar* (“to sing”).

```
Upper:   cantar+Verb+PresInd+1P+Sg
Lower:   canto
```

There is nothing necessary or obvious about using the infinitive to represent the whole lemma; it is only a lexicographical convention of Spanish, and the conventions change from language to language. Latin, for example, has a cognate infinitive *cantare*, but the headword in standard dictionaries is *canto*, the first-person singular present-indicative active form. In a typical **Xerox** Latin analyzer, we would therefore expect *cantare* and all other surface forms of the lemma to be analyzed as forms of *canto*.

```
Upper:   canto+Verb+Inf
Lower:   cantare
```

In analyzing other languages, where lexicographical conventions are mixed or non-existent, it is ultimately up to the linguist to decide what the baseform will be. Some dictionaries, such as those for Esperanto and Sanskrit, follow the admirable convention of using stems rather than whole surface words as the keys in dictionaries. Dictionaries for Semitic languages are usually organized under very abstract root keys, which are typically triplets of consonants that aren't even pronounceable by themselves.

The overall goal is to choose baseforms that will be of maximum familiarity and usefulness in future projects. Where a lexical tradition already exists, writing a **Xerox**-style system that returns the conventional keywords will facilitate looking up definitions and translations in existing lexicons. For a language that has no good tradition for choosing baseforms, it is up to you to choose them. In any case, you should have a solid view of what the baseforms will be before you start coding with tools like **lexc** and **twolc**.

Multicharacter-Symbol Tags and Tag Orders

Choosing Tags See the Appendix on tags, page 585, for detailed information on choosing appropriate tags for your language. Pure linguistic planning should clarify which distinctions are real in the morphology and, therefore, which tags will be necessary to represent those distinctions. As much as possible, the choice of a preliminary set of tags should be done before coding starts.

Tag String Orders Where multiple tags follow the baseform, the linguist must define the order in which tags will appear. Failure to define standard combinations and orders of tags will make it impossible for anyone to use your system for generation. Tag orders should be documented in a Lexical Grammar (see Section 7.4.1).

The Lexical Language is Defined by the Linguist

A transducer encodes a relation between an upper-side language and a lower-side language, where a language is a set of strings. When defining a transducer that performs morphological analysis for a natural language like French, the surface language is usually defined for the linguist by the official orthography. The upper-side or lexical language, however, must be defined by the linguist according to his or her needs and tastes.

The **Xerox** convention of defining lexical strings that consist of a baseform followed by tags may not be suitable in all cases, and indeed this convention is not always followed even at **Xerox**. Here are some cases that suggest other ways to design the lexical language:

1. If the language makes productive use of prefixation rather than suffixation, then it may be much more practical to define a lexical language where the tags are prefixed to the baseform.
2. If the language is Semitic, involving surface stems that are complex interdigitated constructs, then it may be convenient to define a lexical language that separates and labels the component root and pattern (or root, template and vocalization) on the lexical side, depending on your theory.
3. If the goal of the system is pedagogical, then it may be convenient to define a lexical language that contains the (morpho)phonological content of the affixes, not just of the baseform.

So there are no hard-and-fast rules for the design of lexical language. Expert developers understand that the lexical language is defined by the developer, and they put considerable planning into making it as useful as possible for the application or applications at hand.

6.2.3 Planning vs. Discovery

Plan Before You Code is always a good maxim, but no one can plan everything, and there comes a point where one must simply sit down and start writing **lexc**, **xfst** and perhaps **twolc** code. The rigor of formalizing your models in these programming languages will inevitably highlight possibilities and gaps that you didn't even imagine. You will find your own initial intuitions to be unreliable; and even the printed descriptions of your language will soon prove to be inaccurate and incomplete, intended as informal guidance to thinking humans rather than formal descriptions for a computer program. This is never a surprise to formal linguists, but it can come as a shock to those trained only in traditional schoolbook grammar.

Finite-state morphological analyzers can typically process thousands of words per second, analyzing huge corpora while you eat lunch, and quickly revealing the errors and omissions of your grammars and dictionaries. Building and testing a morphological analyzer can therefore be an important part of the linguistic investigation itself.

As problems arise, be prepared to refine your linguistic theories and rewrite parts of your system. Linguistic development is an endless round of observation, theorizing, formalizing and testing; and the goal, for a lexical transducer, is to create a system that correctly analyzes and generates a language that looks as much like the real natural language as possible.

The ultimate goals of any morphological analyzer are to accept and correctly analyze all valid words, and not to accept any invalid words.

6.2.4 Planning for Flexibility

First-time users of the **Xerox** Finite-State Tools usually have a single final application in mind, such as a spelling-checker, part-of-speech disambiguator, parser or even an automatic machine-translation system, and they tend to make design choices that limit the potential uses of the system. Experienced developers understand that a single lexical transducer can and should be the basis for multiple future applications, and they build this flexibility into the rules and lexicons at many levels. A sound understanding of the full power of the finite-state calculus, and especially the union and composition algorithms, is the prerequisite to planning and maintaining flexibility.

Thinking about Multiple Final Applications

The first task of finite-state developers is usually to create a morphological analyzer, i.e. a Lexical Transducer, and such systems are a key component of many other applications. For example, the extracted lower side of a lexical transducer

can serve as a spelling checker, and morphological analysis is a vital step before part-of-speech tagging, parsing, and all the various NLP systems that depend on parsing. **Xerox** tokenizers and HMM (Hidden Markov Model) taggers use a modified lexical transducer as a vital component.

Even within a particular application, such as a spelling checker, one might imagine versions that accept vulgar words, and others that do not. Some morphological analyzers might accept and analyze words from Brazilian Portuguese and Continental Portuguese, and others might specialize in one or the other. In some languages, spelling reforms are adopted from time to time, changing the surface language that needs to be analyzed and generated. The keys to flexibility are the following:

- Avoid making decisions that limit flexibility. Instead of choosing Option A or Option B, try to create a core system that supports both.
- Plant the seeds of flexibility, typically in the form of feature markings, in your source files.
- Use the finite-state calculus, and particularly union and composition, to generate multiple variations from the same source files.

One Core, Many Modifications

In supporting flexibility, the one thing you want to avoid is maintaining multiple copies of core files such as lexicons and alternation rules. You do not, for example, want to maintain separate dictionaries for American English and British English that differ only in small details. It is humanly impossible to maintain multiple large files in parallel.

Dialects Suppose, for example, that you are building a Portuguese system and that there are two major dialects: Brazilian and Continental (i.e. Portuguese as spoken in Portugal). While most written words in the language are common to the two dialects, there are some idiosyncratic and even some systematic differences. For example, the man's name *Antônio* in Brazil is spelled *António*, reflecting a significantly different pronunciation of the accented vowel, in Portugal. The two verb forms spelled *cantamos* and *cantámos* in Portugal are both spelled *cantamos* in Brazil. In addition, the rather productive diminutive suffix *-ito/-ita* used in Portugal is found only in a few fossilized examples, such as *cabrito* ("kid" or "young goat") in Brazil, where the productive diminutive suffixes are *-inho* and *-inha*. Finally, there are some gross terminological differences such as *ananás* ("pineapple") in Portugal vs. *abacaxi* in Brazil.

In such cases, one should not choose one dialect or the other, and one should definitely not maintain two separate lexicons. Instead, the proper finite-state approach is to go into the common source dictionary and place a distinctive feature

marking, e.g. \hat{C} , on the lexical side of all words that are exclusively used in Continental Portuguese. A parallel feature mark, like \hat{B} , is placed on the lexical side of all words that are exclusively used in Brazilian Portuguese.

```
! Marking dialectal vocabulary with multicharacter
! 'features'
```

```
Multichar_Symbols ^C ^B
```

```
LEXICON Root
      Nouns ;
```

```
LEXICON Nouns
gato      Nm ;    ! common word for 'cat'
cachorro  Nm ;    ! common word for 'dog'

ananás    NmC ;   ! Continental for 'pineapple'
abacaxi   NmB ;   ! Brazilian for 'pineapple'

eléctrico NmC ;   ! Continental for 'streetcar'
bonde     NmB ;   ! Brazilian for 'streetcar'
```

```
LEXICON NmC
^C:0      Nm ;
```

```
LEXICON NmB
^B:0      Nm ;
```

A non-final Common Core version of the lexical transducer will then have a majority of lexical strings containing no \hat{B} or \hat{C} feature, some strings marked with \hat{B} , and some with \hat{C} . To create a version of the system that accepts both Brazilian and Continental words, all one needs to do is to start with the Common Core network and map both \hat{C} and \hat{B} upward to the empty string as shown in Figure 6.1.

```
[ ] <- [ %^C | %^B ]
.o.
CommonCoreSystem
```

Figure 6.1: Leave Both Brazilian and Continental Words

Similarly, to create a purely Brazilian system that contains no exclusively Continental words, we map \hat{B} to the empty string and \hat{C} to the null language (which eliminates all the Continental paths) as in Figure 6.2. (The null language is that language that contains no strings at all, not even the empty string. The null language can be notated in an infinite number ways, including $\sim\$\square$, $\sim[?^*]$, $[a - a]$, etc.)

```

~$[] <- %^C
.ο.
[] <- %^B
.ο.
CommonCoreSystem

```

Figure 6.2: Keep Brazilian Words, Kill Continental Words

Finally, to create a purely Continental system that contains no exclusively Brazilian words, we do the opposite, as shown in Figure 6.3. The order of the compositions in Figures 6.2 and 6.3 is not significant for these examples.

```

[] <- %^C
.ο.
~$[] <- %^B
.ο.
CommonCoreSystem

```

Figure 6.3: Kill Brazilian Words, Keep Continental Words

Spelling Reforms Spelling reforms, like dialectal differences, are problems that plague all developers in natural-language processing. Beginning developers too often feel that they have to make a choice, to accept only the old orthography or only the new orthography. The better solution, of course, is to accept both, or either, in variants of the same core system. Spelling reforms are often announced and then abandoned, or new and old forms may exist side-by-side for years; in any case, large corpora of text in the old orthography may exist for years and still need to be processed. Creative flexibility is highly encouraged when dealing with spelling reforms.

The basic mechanism is the same as for dialectal differences. One feature, such as $\wedge N$ (for “new”) is placed on the upper side of new word forms, and something like $\wedge O$ (for “old”) is placed on forms destined someday to disappear. Then you use simple composition, as in the dialect examples above, to allow or delete new and old orthographical forms in variations of the final lexical transducer.

In any apparent choice between two desirable alternatives, always try to have it three ways: either one, or the other, or both.

Spelling differences can straddle the line between dialects and spelling reforms. In Brazil, there was a traditional distinction between the spellings *qui* representing /ki/ and *qüi* representing /kwi/; a Brazilian or student of Brazilian could therefore look at the written word *tranqüilo* (“tranquil”) and know how it was pronounced. The pronunciation of the word is the same in Portugal, but the *ü* letter is never used in the orthography, making written *tranquilo* phonetically ambiguous, at least to a foreigner trying to learn the language. The solution in such cases is always to adopt the spelling convention with the maximum number of distinctions for the common core lexicon, making the spelling *tranqüilo*, with the *ü*, the baseform. It’s always easy to collapse the orthographical *u/ü* distinction for Continental Portuguese, as in Figure 6.4, but adding distinctions post hoc requires going in and editing the core lexicons.

```

u <- ü
.○.
CommonCoreSystem
.○.
ü -> u

```

Figure 6.4: Collapsing all *ü*s to *u*s for Continental Portuguese

In recent years, Brazil and Portugal have been trying to make their orthographies more similar, and one element of the proposal was for Brazil to abandon the use of *ü*. The composition in Figure 6.4 will therefore serve for the New Brazilian orthography as well. Yet we still do not want to edit the source files; we may still, for years to come, want to be able to create versions of our system that accept only Traditional Brazilian orthography.

Vulgar, Slang, Substandard Words As with dialect words, it is sometimes desirable to produce variations of a system that contain no vulgar, slang, substandard or other politically incorrect words. This may be the case, for example, in a spelling checker that returns suggested corrections. Such words should also be marked with features in the core lexicons, perhaps

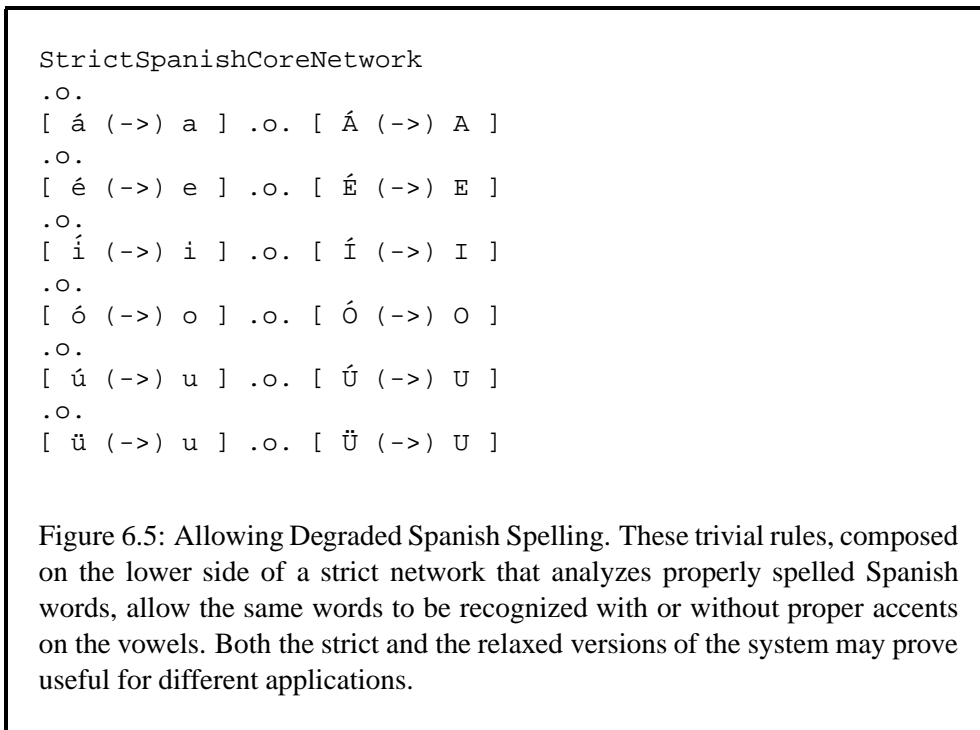
```

^V      for Vulgar
^S      for Slang
^D      for Substandard/Deprecated

```

Composition can once again be used to create many customized final versions without any re-editing of the source files.

Handling Degraded Spelling In some applications, such as parsing email messages or text retrieval, it is often desirable to create a version of a morphological analyzer that accepts words even when they are incorrectly accented. Once again, the trick is first to create a strict core system that has a lower-side language consisting of only properly spelled words, and then use composition to create versions that accept degraded spellings. For example, it is very common for Spanish and Portuguese email writers to dispense with accents, degrading letters like *é* to *e* and *ô* to *o*. The best way to handle such deaccentuation is to create a permissive version of your basic strict system by composing a RELAXING GRAMMAR on the bottom of the common core system. The relaxing grammar for Spanish in Figure 6.5 optionally allows accented letters to be unaccented on the new surface level.



In another useful but technically trivial example, German spelling has long tolerated the convention that *ü* can be spelled *ue*, *ö* can be spelled *oe*, and *ß* can be spelled as *ss* if the correct letters are not available. Starting with a strict German transducer, with *ü*, *ö* and *ß* on the lower side, the rules shown in Figure 6.6 will create a more permissive version.

Specialized Extra Words Often one customer will want special strings, such as proprietary part numbers, email addresses, internal telephone numbers or even

```

StrictGermanCoreNetwork
.ö.
[ ü (->) u e ]
.ö.
[ ö (->) o e ]
.ö.
[ ß (->) s s ]

```

Figure 6.6: Allowing Relaxed German Spelling. These trivial rules, composed on the lower side of a strict network that analyzes properly spelled German words, allow the resulting network to accept conventional respellings.

common misspellings, added to the morphology system; but other customers may not need or want such words at all. The solution in such cases is to build a common core system of words that everyone is likely to want, and then write separate grammars of specialized words. These separate grammars can then be selectively unioned with the strict core system to make customized versions for particular customers.

To take a concrete Spanish example, the plural word for “countries” is formally *países*, with an acute accent on the *i*, but even some otherwise careful newspapers omit the accent in this one case. If the correctly spelled word analyzes as

```
país+Noun+Masc+Pl
```

then the strict core system can be loosened up for this one word with the following trivial union.

```

StrictSpanishCoreNetwork |
[ [ {país} %+Noun %+Masc %+Pl ] .x. {paises} ]

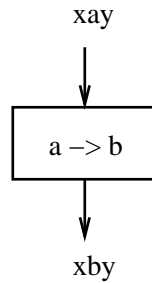
```

6.3 Composition is Our Friend

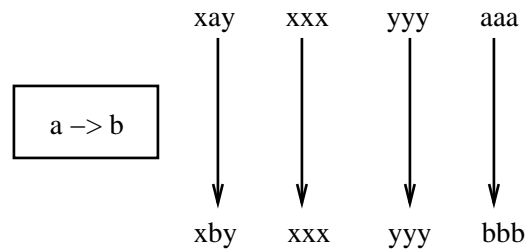
As is clear from the examples shown above, composition is a powerful tool for customizing lexical transducers, allowing a single core system to serve multiple ends. In this section, we present a summary of useful things you can do with transducers and composition.

6.3.1 Modifying Strings

We usually picture transducers as little abstract machines that map one string into another string or strings. For example, from a generation point of view, consider the following mapping

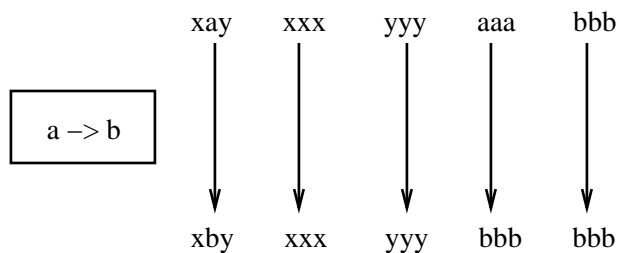


which changes “xay” (perhaps a word in our dictionary) into “xby”. In such cases, we usually do not think of the rule as changing the size of original lexical language, but just modifying one or more strings in some way. If we have the following four strings in the lexical language, they would be mapped into surface strings (in vertical correspondence) as shown.



Note that where a rule doesn’t apply, as in the cases of “xxx” and “yyy”, the rule simply maps the input to the output with no change.

In some cases, by accident, rules can effectively collapse two strings in the mapping. If, for example, the lexical language also contains “bbb”, we would see the following:



where the generation of lexical “aaa” and lexical “bbb” both appear on the surface as “bbb”. The resulting surface “bbb” is therefore ambiguous, and the lower language has one fewer distinct strings than the upper language. Such ambiguities are often seen in natural language. Consider the Spanish noun *canto* meaning

“song” or “chant”. It happens that *canto* is also “I sing”, one of the many conjugated forms of the verb *cantar* (“to sing”). Suppose that the initial **lexc** lexicon specifies the following two string pairs

```
Upper:   canto+Noun+Masc+Sg
Lower:   canto+Noun+Masc+Sg

Upper:   cantar+V1+PresInd+1P+Sg
Lower:   cantar+V1+PresInd+1P+Sg
```

and that we define the following cascade of rules to be composed on the bottom of the lexicon.

```
[ a r -> [ ] | | _ %+V1 ]
.o.
[ %+V1 %+PresInd %+1P %+Sg -> o ]
.o.
[ %+V1 %+PresInd %+2P %+Sg -> a s ]
.o.
[ %+V1 %+PresInd %+3P %+Sg -> a ]
.o.
...           ! many more rules
.o.
[ %+Noun %+Masc %+Sg -> [ ] ] ;
```

If you trace the generation by hand (or using **xfst**), you will find that both surface strings end up as “canto”.⁴

When used in the way just described, the application (composition) of rules to the lexicon tends not to modify the size of the resulting languages very much. The assumption is that the lexicon is generating valid abstract strings, and that it is the job of the rules simply to modify them to produce valid surface strings, some of which will be ambiguous.

6.3.2 Extracting Subsets of a Transducer

Composition can be used to extract subsets of paths from a transducer or language, and this is often done during testing. Assuming we have a large transducer in file `Esperanto.fst` that covers nouns, adjectives and verbs, and assuming that the tags `+Noun`, `+Adj` and `+Verb` mark the appropriate words on the upper side, we can read in `Esperanto.fst` and isolate just the nouns with the following regular expression:

⁴Beware: you should test such bare cascades of rules using **apply down**; **apply up** may yield odd results or even segmentation faults until a restricting lexicon itself is composed on top of the rules.


```
xfst[0]: read regex
$[%+Noun]
.o.
@"Esperanto.fst" ;
```

The regular-expression notation `@"Esperanto.fst"` tells the **xfst regex** compiler to read in the network stored in the binary file `Esperanto.fst`. The double quotes are necessary if the filename contains punctuation characters, as in this case. The expression `$[%+Noun]` denotes the language consisting of all and only strings that contain the symbol `+Noun`. When composed on the top of `Esperanto.fst`, it will match all the noun strings (because they have a `+Noun` symbol in them) and will fail to match any of the verb and adjective strings (precisely because they lack a `+Noun` symbol). The result of the composition is that subset of `Esperanto.fst` that contains just noun strings; all the adjectives and verbs and anything else without a `+Noun` symbol on the upper side are deleted in the composition.

Conversely, we can filter out the nouns, leaving everything else, by composing the complement of `$[%+Noun]`, i.e. `~$[%+Noun]`, on top of the lexicon:

```
xfst[0]: read regex
~$[%+Noun]
.o.
@"Esperanto.fst" ;
```

The expression `~$[%+Noun]` compiles into a simple-language network that accepts all and only strings which do not contain `+Noun`.

Multiple levels of regular expressions or rules can be composed on either side of a transducer as appropriate, but it is up to the developer to keep track of what kind of strings are on each side. If you have a transducer saved as `myfile.fst`, with the surface strings on the lower side, as usual, you can limit it to contain only those paths with surface strings that end in `ly` with the following composition:

```
xfst[0]: read regex
@"myfile.fst"
.o.
[ ?* l y ] ;
```

Notice that such a surface filter needs to be composed on the lower side of the lexicon transducer, where the surface strings are visible. You can limit the same grammar to cover only surface strings that begin with *re*:

```
xfst[0]: read regex
@"myfile.fst"
.o.
[ r e ?* ] ;
```

Note that in general regular expressions, including examples such as $[?^* l y]$ and $[r e ?^*]$, the beginning and the end of the denoted strings are implicitly specified. $[?^* l y]$ denotes the language of all strings that end with *ly*; $[r e ?^*]$ denotes the language of all strings that begin with *re*. The $. \# .$ notation, is *not* appropriate here; $. \# .$ can appear only in restriction contexts and in replace-rule contexts (see pages 49 and 140).

You can extract all the words that, on the surface, begin with *re* and end with *ly* with the following:

```
xfst[0]: read regex
@"myfile.fst"
.○.
[ r e ?* l y ] ;
```

You can limit the system to contain only adjectives ending (on the surface) with *ly* with the following:

```
xfst[0]: read regex
$[%+Adj]
.○.
@"myfile.fst"
.○.
[ ?* l y ] ;
```

If *myfile.fst* has good coverage of English adjectives, this final exercise will isolate cases like *friendly*, *comely*, *homely* and *cowardly*.

Note that tags, by **Xerox** convention, are found on the upper side of a lexical transducer, so the restriction $\$[%+Adj]$ must be composed on the top of *myfile.fst*, while the *ly* requirement is a restriction on surface strings, and so that restriction must be composed on the bottom where the surface strings are. As always, it is vitally important to keep straight which side is up and which side is down, and what the strings look like at each level, when performing composition.

6.3.3 Filtering out Morphotactic Overgeneration

It is also possible, and often useful, to use composition to restrict an overgenerating lexicon. Many morphotactic restrictions found in real language are awkward to impose within **lexc** (or any other formalism for specifying finite-state networks), and in such cases it may be useful to define, on purpose, an initial lexicon that overgenerates. As a simple example, consider the Esperanto Animal Nouns, where

readers were urged (see Section 6, page 231) not to worry about restricting multiple occurrences of the feminine (+Fem) suffix *in*, the augmentative (+Aug) suffix *eg* or the diminutive (+Dim) suffix *et*.

Here's one purposely overgenerating **lexc** description:

```
Multichar_Symbols +Noun +Aug +Dim +Fem +Sg +Pl +Acc
```

```
LEXICON Root
      Nouns ;
```

```
LEXICON Nouns
hund      N ;           ! dog
kat       N ;           ! cat
elefant  N ;           ! elephant
```

```
LEXICON N
+Aug:eg   N ;           ! looping site!
+Dim:et   N ;
+Fem:in   N ;
          NSuff ; ! escape from the loop
```

```
LEXICON NSuff
+Noun:o   Number ;
```

```
LEXICON Number
+Sg:0     Case ;
+Pl:j     Case ;
```

```
LEXICON Case
#         # ;
+Acc:n   # ;
```

Despite the instructions, many readers do worry a lot about the fact that this grammar will happily overgenerate strings like

```
hundegego
hundininoj
elefantineginetojn
```

and they try to incorporate in the **lexc** description various restrictions that may or not reflect realities in Esperanto grammar. In any case, let us assume for the purposes of this example that the three suffixes *et*, *eg* and *in* can all co-occur in the same word, in any order, but that a maximum of only one of each can appear in a valid word. Let us also assume that we have compiled our overgenerating **lexc** grammar and stored the results in `esp-lex.fst`. If you actually compile this grammar, **lexc** will inform you that it contains 18 states, 23 arcs and is CIRCULAR; this means that the grammar produces an infinite number of strings. The circularity of course results from the loop in LEXICON N.

Here is a step-by-step reasoning to a solution that removes the overgeneration:

- We want to restrict our lexicon to contain only those string pairs where the string on the lexical side contains zero or one appearances of +Dim, zero or one appearances of +Aug, and zero or one appearances of +Fem. We will let these three suffixes occur in any order, but we never want to see two or more of the same suffix in a single string.
- Another way to look at the problem is to realize that any lexical string that contains two or more +Aug symbols is bad. We can formalize these bad strings as

$$\$(\%+Aug\ ?^*\ \%+Aug] \quad ! \text{ some bad lexical strings}$$

Read this as “all strings that contain one +Aug followed at any distance by another +Aug”. This is a finite-state characterization of what we do not want: having two or more +Aug symbols in the same word.

- Similarly, a lexical string that contains two or more +Dim symbols, or two or more +Fem symbols, is also bad.

$$\$(\%+Dim\ ?^*\ \%+Dim] \quad ! \text{ more bad lexical strings}$$

$$\$(\%+Fem\ ?^*\ \%+Fem] \quad ! \text{ yet more bad lexical strings}$$

- A single expression that matches all the bad strings is the following:

$$\begin{aligned} & [\$(\%+Aug\ ?^*\ \%+Aug] \\ & | \$(\%+Dim\ ?^*\ \%+Dim] \\ & | \$(\%+Fem\ ?^*\ \%+Fem] \end{aligned}$$

We want to remove every string pair whose lexical side is in this language from the overgenerating transducer created by **lexc**. We cannot subtract transducers, because subtraction is not a valid operation on transducers, but we can accomplish our goal by creative composition.

- The complement of the bad strings denotes a language consisting of good strings:

$$\begin{aligned} \sim & [\$(\%+Aug\ ?^*\ \%+Aug] \\ & | \$(\%+Dim\ ?^*\ \%+Dim] \\ & | \$(\%+Fem\ ?^*\ \%+Fem] \end{aligned}$$

This expression will match all strings *except* the bad ones we want to eliminate.

- And when we compose this net *on top of* the overgenerating lexicon (remember that the tags are on the top), the composition will impose the restrictions we desire. The unmatched strings, which are bad, are simply eliminated in the composition.

```
xfst[0]: read regex
~[ $[%+Aug ?* %+Aug]
  | $[%+Dim ?* %+Dim]
  | $[%+Fem ?* %+Fem] ]
.ο.
@"esp-lex.fst" ;
```

When you compile this expression, **xfst** tells you the result contains 34 states, 48 arcs, and 192 words. As the original `esp-lex.fst` was circular, it contained an infinite number of words. By imposing the restriction, we have reduced the number of words covered and, in doing so, have increased the memory size (the number of states and arcs) of the network. Imposing restrictions on a transducer often results in increased memory. In the worst cases, composition restrictions on a transducer can cause it to “blow up” in size.

The use of composed filters to eliminate overgeneration is often much cleaner than trying to impose the same restrictions in **lexc**. For an alternative expert method to limit morphotactic overgeneration, see Chapter 8 on Flag Diacritics.

6.4 Priority Union for Handling Irregular Forms

6.4.1 Irregular Forms

When writing morphological analyzers for natural languages, one is often faced with a finite number of idiosyncratic irregular forms that simply do not follow the usual rules. English noun plurals provide convenient examples, and we shall see that it is sometimes convenient to add exceptional plurals to the productive plurals in a transducer, while other times it is convenient to override productive, but unacceptable, plurals with irregular ones. The adding of new paths to a transducer is easily accomplished using the union algorithm, which is already familiar to us. The overriding of paths in one transducer with paths from another is accomplished elegantly using the PRIORITY UNION algorithm, which will be illustrated below.

Looking at the data, we note that some English nouns like *money* have a normal productive plural plus one or more exceptional plurals as shown in Table 6.1. In these cases the final morphological analyzer must recognize the irregular forms as well as the regular ones. We will refer to these irregular forms as EXTRA PLURALS.

Noun	Regular	Irregular
money	moneys	monies
octopus	octopuses	octopi
bus	buses	busses
cherub	cherubs	cherubim
fish	fishes	fish

Table 6.1: Some English Nouns with Extra Irregular Plurals

Noun	Regular	Irregular
sheep	*sheeps	sheep
deer	*deers	deer
kibbutz	*kibbutzes	kibbutzim
automaton	*automatons	automata

Table 6.2: Some Irregular English Plurals that Override the Regular Forms

Some nouns like *sheep* and *deer* do not change in the plural, and other nouns like *kibbutz* have been half-borrowed into English, keeping only their native plurals, which are irregular from the English point of view. In such cases, illustrated in Table 6.2, the final morphological analyzer should not recognize the regular plurals but only the irregular ones. We will refer to these irregular forms as **OVERRIDING PLURALS**.

We will now examine three ways that these irregular plurals can be handled, concluding with the technique of using priority union to override undesired forms with desired forms. This new technique is especially appropriate and elegant when the exceptions are, as with these English plurals, finite and genuinely idiosyncratic.

6.4.2 Handling Irregularities in *lexc* and Rules

It is possible, but usually awkward, to handle irregular English plurals by encoding them specially in *lexc* and perhaps writing extra alternation rules for them. Thus one could imagine writing a *lexc* grammar like the one in Figure 6.7, where each of the irregular nouns is given a customized continuation class. For example, the continuation class *Nsame* for *sheep* and *deer* is easily defined, as is the class *Nim*

```

Multichar_Symbols +Noun +Sg +Pl

LEXICON Root
      Nouns ;

LEXICON Nouns
dog      Nreg ;      ! regular nouns, take -s plural
cat      Nreg ;
dish     Nreg ;      ! regular noun, takes -es plural
                        ! (handled by productive rule)

money    NregIes ;   ! noun takes -s plural plus -ies
octopus  NregI ;     ! noun takes -s plural plus -i
bus      NregSes ;   ! noun takes -es plural plus -ses
cherub   NregIm ;    ! noun takes -s plural plus -im
sheep    Nsame ;     ! no change in the plural
deer     Nsame ;
kibbutz  Nim ;       ! -im plural only
automaton Na ;       ! -a plural only
fish     NregSame    ! plurals fish and fishes

```

Figure 6.7: Trying to Handle Irregular Plurals in **lexc** with Customized Continuation Classes

for *kibbutz*.

```

LEXICON Nreg
+Noun+Sg:0      # ;
+Noun+Pl:s      # ;

LEXICON Nsame
+Noun+Sg:0      # ;      ! deer
+Noun+Pl:0      # ;      ! deer

LEXICON Nim
+Noun+Sg:0      # ;      ! kibbutz
+Noun+Pl:im     # ;      ! kibbutzim

```

The more interesting NregIm class for *cherub* could lead to a sublexicon that in turn leads to the normal single and plural endings and then adds the extra *-im* ending. A similar solution is possible for NregSes and NregSame (for *fish*).

```

LEXICON NregSes
      Nreg ;
+Noun+Pl:ses   # ;

LEXICON NregIm
      Nreg ; ! for regular endings: cherub/cherubs
+Noun+Pl:im    # ; ! for the extra -im plural: cherubim

LEXICON NregSame
      Nreg ; ! for the regular endings: fish/fishes
+Noun+Pl:0     # ; ! for fish as plural

```

Other cases are harder. For example, the NregIes class for *money* will need to lead to the regular endings and to *-ies*, but simply adding *-ies* will yield **moneyies*, and rules of some type will need to be applied to yield the final correct form *monies*.

```

LEXICON NregIes
      Nreg ; ! money/moneys
+Noun+Pl:ies  # ; ! *moneyies
              ! needing rules to map it to monies

```

For finite exceptional cases like these English plurals, trying to handle them in **lexc** leads to a proliferation of continuation classes, and one must often write ad hoc rules to handle a single root or a small handful of obviously exceptional roots. In general, one would prefer to keep the **lexc** grammar simpler and to reserve the writing of rules for more productive phenomena.


```

Multichar_Symbols  +Noun +Sg +Pl

LEXICON Root
      Nouns ;

LEXICON Nouns
cat      Nreg ;
dog      Nreg ;
dish     Nreg ;
money    Nreg ;
octopus  Nreg ;
cherub   Nreg ;
fish     Nreg ;

```

Figure 6.8: A **lexc** Grammar that Generates only Regular Plurals

6.4.3 Handling Exceptions via Union and Composition

Handling Extra Plurals

Recall that we made a distinction in Section 6.4.1 between *extra* plurals, which need to be recognized in addition to the regular plurals, and *overriding* plurals, which need to override or supplant the regular plurals.

In the finite-state world, the notion of adding-to is easily accomplished via the union algorithm, so we'll start with the handling of extra plurals. We can start with a **lexc** grammar that undergenerates, producing only the regular plurals for nouns like *money*, *cherub*, *octopus* and *fish* as shown in Figure 6.8. All that remains is to union in the extra irregular plurals, which are easily defined by hand; sometimes this is referred to as adding forms by “brute force”. This can be done in **lexc** itself as shown in Figure 6.9.

Recall that in **lexc**, the entries in any LEXICON are simply unioned together, so this grammar will union together all the paths built by following the Nreg continuation and all the extra hand-defined plurals. The proliferation of continuation classes is avoided, as is the writing of ad hoc alternation rules.

There are many ways to accomplish the same task. In **lexc**, the extra plurals could be grouped into a separate sublexicon, as in Figure 6.10. The irregular forms could also be defined, again by hand, in a completely separate **lexc** program, compiled into a separate network, and then the main network and the network of extra noun plurals could simply be unioned together in **xfst**. The extra plurals could also be defined directly in **xfst** itself and then unioned with the undergenerating lexicon from **lexc**. The **xfst** script in Figure 6.11 assumes that the **lexc** grammar has been compiled and that the network is stored in file `lex.fst`.

```

Multichar_Symbols +Noun +Sg +Pl

LEXICON Root
      Nouns ;

LEXICON Nouns
cat      Nreg ;
dog      Nreg ;
dish     Nreg ;

money    Nreg ;
money+Noun+Pl:monies # ; ! extra plural

octopus  Nreg ;
octopus+Noun+Pl:octopi # ; ! extra plural

cherub   Nreg ;
cherub+Noun+Pl:cherubim # ; ! extra plural

fish     Nreg ;
fish+Noun+Pl:fish # ; ! extra plural

```

Figure 6.9: Unioning in Irregular Extra Plurals by Brute Force

```

Multichar_Symbols  +Noun +Sg +Pl

LEXICON Root
    Nouns ;
    ExtraIrregularPlurals ;

LEXICON Nouns
cat      Nreg ;
dog      Nreg ;
dish     Nreg ;
money    Nreg ;
octopus  Nreg ;
cherub   Nreg ;
fish     Nreg ;

LEXICON ExtraIrregularPlurals
money+Noun+Pl:monies      # ;
octopus+Noun+Pl:octopi   # ;
cherub+Noun+Pl:cherubim # ;
fish+Noun+Pl:fish        # ;

```

Figure 6.10: Unioning in Extra Plurals in a Separate Sublexicon

```

clear stack

define ExtraIrregularPlurals [ {money}:{monies}
| {octopus}:{octopi}
| {cherub}:{cherubim}
| {fish}
] %+Noun:0  %+Pl:0
;

read regex  ExtraIrregularPlurals | @"lex.fst" ;
save stack augmentedlex.fst

```

Figure 6.11: Unioning in Extra Plurals in **xfst**. The network that handles only regular plurals is stored in `lex.fst`. The script unions the extra plurals with `lex.fst` and saves the result as `augmentedlex.fst`.

Handling Overriding Plurals with Composition, then Union

For handling the overriding plurals, we can

1. first define a **lexc** grammar that overgenerates regular plurals
2. use composition to filter the overgenerated plurals from the network, and then
3. use union to add the overriding plurals

As we shall see in the next section, this idiom is helpfully encapsulated in a single finite-state algorithm called priority union.

```

Multichar_Symbols  +Noun +Sg +Pl

LEXICON Root
      Nouns ;

LEXICON Nouns
cat      Nreg ;
dog      Nreg ;
dish     Nreg ;
money    Nreg ; ! undergenerates; no monies pl.
octopus  Nreg ; !                no octopi pl.
cherub   Nreg ; !                no cherubim pl.
fish     Nreg ; !                no fish pl.

sheep    Nreg ; ! overgenerates *sheeps
deer     Nreg ; !                *deers
kibbutz  Nreg ; !                *kibbutzes
automaton Nreg ; !                *automatons

```

Figure 6.12: A **lexc** Grammar that Both Undergenerates and Overgenerates Plurals

To illustrate the idiom step by step, assume that we start with the **lexc** grammar in Figure 6.12 that overgenerates regular plurals for *sheep*, *deer*, *kibbutz* and *automaton*. It also undergenerates by not handling the irregular plurals for these roots and for words like *money* that have both regular and irregular plurals.

In a separate **xfst** grammar, we can define a set of overriding plurals in addition to the set of extra plurals. The script in Figure 6.13 assumes that the network from the **lexc** grammar is stored in file `lex.fst`. Follow the comments in the script to see how the incorrect regular plurals are first removed by an upper-side filtering composition before the irregular plurals are unioned into the result.

```

clear stack

define ExtraIrregularPlurals [ {money}:{monies}
| {octopus}:{octopi}
| {cherub}:{cherubim}
| {fish}
] %+Noun:0  %+Pl:0
;

define OverridingIrregularPlurals [ {sheep}
| {deer}
| {kibbutz}:{kibbutzim}
| {automaton}:{automata}
]  %+Noun:0  %+Pl:0
;

# now set Filter to the upper-side language of
# OverridingIrregularPlurals; it contains strings like
# sheep+Noun+Pl and automaton+Noun+Pl

define Filter  OverridingIrregularPlurals.u ;

# read lex.fst from file, and compose on top of it
# the complement of the Filter.  This filters out
# the paths that have sheep+Noun+Pl, deer+Noun+Pl,
# kibbutz+Noun+Pl and automaton+Noun+Pl on the
# upper side.  The operation therefore removes the
# overgenerated regular plural paths.

define FilteredLex  ~Filter .o. @"lex.fst" ;

# now simply union in the extra plurals and
# the overriding plurals to create the
# final result

read regex
FilteredLex |
ExtraIrregularPlurals |
OverridingIrregularPlurals ;

save stack augmentedlex.fst

```

Figure 6.13: Overriding via a Composed Filter and Union

Handling Overriding Plurals via Priority Union

```

clear stack

define ExtraIrregularPlurals [ {money}:{monies}
| {octopus}:{octopi}
| {cherub}:{cherubim}
| {fish}
] %Noun:0 %Pl:0
;

define OverridingIrregularPlurals [ {sheep}
| {deer}
| {kibbutz}:{kibbutzim}
| {automaton}:{automata}
] %Noun:0 %Pl:0
;

# use of upper-side priority union .P.
# overrides bad plurals with good plurals

read regex
[ OverridingIrregularPlurals .P. @"lex.fst" ] |
ExtraIrregularPlurals ;

save stack augmentedlex.fst

```

Figure 6.14: Simplified Overriding Using the Upper-Side Priority-Union Operator .P.

As illustrated in the previous section, irregular plurals can override unwanted regular plurals via an idiom that combines upper-side filtering and union. This idiom is so useful that it has been packaged in a single finite-state algorithm called PRIORITY UNION, or more precisely, *upper-side* priority union. In a regular expression, upper-side priority union is notated with the operator `.P.` as in the script in Figure 6.14. Note that the `.P.` operation is ordered, with the left-side network operand having the priority. For the operation `L .P. R`, the result is a union of `L` and `R`, except that whenever `L` and `R` have the same string on the upper side, the path in `L` overrides (has precedence over) the path in `R`.

For completeness, the regular-expression language also includes the operator `.p.`, for lower-side priority union, but the `.P.` operator is more useful in practice given the **Xerox** convention of putting baseforms and tags on the upper side of a transducer.

6.5 Conclusions

6.5.1 Take Software Engineering Seriously

The building of a finite-state morphological analyzer is a significant software project that deserves careful planning. Developers should make sure that they have adequate hardware, up-to-date versions of the finite-state software, a version-control system, backups, and the ability to automate complex compilations using makefiles. This may require cooperation among linguists and computer scientists.

6.5.2 Take Linguistic Planning Seriously

The bulk of linguistic planning and modeling should be done before the coding starts, although the rigor of writing an automatic morphological analyzer is also a discovery process that can lead to many insights. The overall plan of the system, showing the division into lexicon and rule modules, and the way that various transducers combine together into the final lexical transducer(s), should be formalized in makefiles and made visual in wall charts for easy reference. From the beginning, developers should have a clearly defined strategy for handling irregular forms and for supporting multiple final applications.

Handling Irregular Forms

When making decisions concerning the handling of morphological phenomena, the principle of CREATIVE LAZINESS should apply. For morphotactic and morphophonological phenomena that are truly productive, it is best to handle them using **lexc** and alternation rules. Writing rules is the “lazy” way to handle lots of similar examples that would be tedious to handle by brute force. But for finite, idiosyncratic irregularities like the English plurals shown above, it is often cleaner and easier to create an initial lexicon transducer that overgenerates and undergenerates, and then fix it using union, composition, and perhaps priority union. Handling a very finite number of exceptions via brute force is easier and lazier than writing a lot of ad hoc rules, most of which are, in any case, suspicious from any phonological point of view. These tricks facilitate the lexicography and keep the grammars smaller and easier to maintain.

Customization and Multiple Use

Expert finite-state developers plan from the beginning for flexibility and adaptability. Starting with a common core network, which may by itself be useless for any particular application, the goal is to be able to modify that core network via finite-state operations, notably composition, union and priority union, to create a multitude of variant systems that support

- Multiple final applications, such as morphological analyzers, automatic dictionary lookup systems, disambiguators, parsers, machine-translation systems, etc. (see page 384)
- Multiple dialects of the same language (see page 385)
- Multiple orthographies (e.g. after a spelling reform, see page 387)
- Multiple registers, including or excluding vulgar, slang and substandard words (see page 388)
- Multiple levels of strictness, e.g. properly accented vs. deaccented Spanish (see page 389)

The key in this customization technique, as it is in the filtering out of morphotactic generation (Section 6.3.3) is to understand that a finite-state network is an object that can be modified by finite-state operations, without having to return and edit the source files that created it in the first place.

Chapter 7

Testing and Debugging

Contents

7.1	The Challenge of Testing and Debugging	410
7.2	Testing on Real Corpora	410
7.2.1	Hand-Testing vs. Automated Testing	410
7.2.2	Testing for Failures	411
	Collecting Words that Aren't Analyzed	411
	Testing for Failures with a Guesser	416
7.2.3	Positive Testing	416
7.3	Checking the Alphabet	418
7.3.1	The Gap in Symbol-Parsing Assumptions	418
7.3.2	Failure to Declare Multichar_Symbols in lexc	419
7.3.3	Inadvertent Declarations of Multicharacter Symbols in xfst and twolc	420
7.3.4	When to Suspect an Alphabet Problem	421
7.3.5	Idioms for Checking the Alphabet	423
	Printing Out the Sigma and Labels	423
	Setting print-space ON	425
7.4	Testing with Subtraction	427
7.4.1	Testing the Lexical Language	427
7.4.2	Testing the Surface Language	430
	Testing Against Wordlists	431
	Testing Against Previous Versions	433
	Testing Against External Lexicons	435

7.1 The Challenge of Testing and Debugging

The testing of any sizable natural-language processing system is notoriously difficult. A full-scale morphological analyzer will typically grow to contain tens of thousands of baseforms and millions of surface forms. An analyzer that handles productive compounding will in fact handle an infinite number of surface forms. When you edit your lexicons or rules and recompile the system, how can you tell what was added, lost or perhaps broken in the process?

Fortunately, the finite-state calculus provides many powerful ways to test a system against large corpora, wordlists, other lexicons, Lexical Grammars, and previous versions of the system itself. Networks can also be examined in various ways, in particular to print lists of their alphabets or labels, to flush out typical programming errors. This chapter describes the idioms for testing and debugging finite-state morphology systems using the Finite-State Calculus itself.

7.2 Testing on Real Corpora

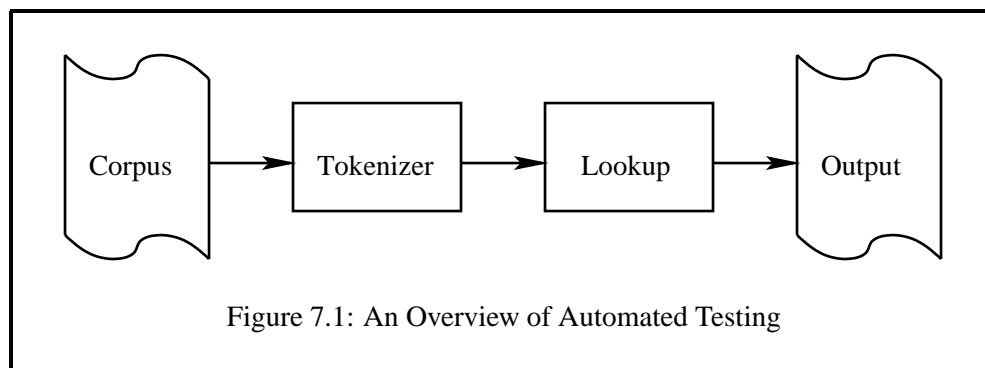
7.2.1 Hand-Testing vs. Automated Testing

In the days before computers, or at least before computational linguistics became practical, linguists with formal instincts and training would often produce elaborate grammars on paper. While much excellent linguistics was done in this way, *faute de mieux*, all testing had to be done by hand; and rare is the human being with the patience and concentration necessary to run even a hundred examples reliably through a complicated grammar, let alone thousands. It is not surprising that most paper grammars are poorly tested and inaccurate.

With **Xerox** finite-state compilers and interfaces, grammars are compiled and combined into highly efficient Lexical Transducers that typically analyze real text at rates of thousands of words per second. However, the testing facilities built into the development tools themselves are quite limited, intended mainly for quick manual checking of a few words here and there during development. Thus **lexc** offers the **lookup** and **lookdown** commands (page 216), which operate on just a single token (string) at a time. The **apply up** and **apply down** (page 89) commands in **xfst** can handle single tokens, and even accept input from a wordlist file that has one token per line; but neither tool offers a way to output to file, or to pipe the output of morphological analysis to a subsequent process like a disambiguator or parser.

For serious testing, we need to automate the process, applying our lexical transducers to corpora containing hundreds, thousands or even millions of words, and outputting successful analyses and FAILURES (not-found words) to various output files for later review. By CORPUS, plural CORPORA, we mean any files containing normal running text.

The testing procedure, shown in overview in Figure 7.1, is to pipe the corpus to a TOKENIZER utility that breaks it up into individual TOKENS, which are typi-



cally orthographical words, and which are in turn piped to a LOOKUP utility that tries to analyze them. The software available with this book includes command-line utilities called **tokenize** and **lookup** which we will examine more closely in Chapter 10. The use of **tokenize**, in particular, requires the definition of a special language-specific tokenizing transducer that we don't want to deal with just yet.

Whereas **lexc**, **xfst** and **twolc** are DEVELOPMENT TOOLS for creating finite-state networks, **tokenize** and **lookup** are RUNTIME APPLICATIONS that apply your transducers to do serious testing and practical natural-language processing.

The runtime application called **lookup** should not be confused with the **lookup** command that is found in the **lexc** interface.

In this chapter, we will provide some useful testing idioms that can be copied and used without necessarily understanding all the details and options.

7.2.2 Testing for Failures

Morphological analyzers are never complete or perfect, and we need various ways to test them and find the problems.

Collecting Words that Aren't Analyzed

When testing your morphological analyzer on a large corpus, it is usually practical to collect and look only at the failures, that is, at the input words for which the analyzer fails to find any analysis at all. In this section, we will present some useful idioms for collecting and sorting such failures or "sins of omission". In Section 7.2.3 we will look at more positive testing that discovers "sins of commission"

and partial analysis failures.

Failures can result from mistakes in your **xfst** or **twolc** rules, or mistakes in your **lexc** morphotactic description; other failures result from the simple fact that you haven't yet added the appropriate baseforms to the **lexc** lexicon. After the basic outlines of your analyzer are written and working, subsequent development becomes a never-ending round of addition and refinement of your rules and lexicons, trying to make the language accepted by the Lexical Transducer correspond to the target natural language as closely as possible.

Assuming that your corpus is stored in file `mycorpus.txt` and that your morphological analyzer is stored in `mylanguage.fst`, the following testing idiom generates an output file of not-found words called `failures.all`.¹

```
unix> cat mycorpus.txt | \
tr -sc "[:alpha:]" "[\n*]" | \
lookup mylanguage.fst | \
grep '+?' | \
gawk '{print $1}' > failures.all
```

This somewhat intimidating command uses the fairly standard Unix command-line utilities **cat**, **tr**, **grep** and **gawk**; and it requires that the **lookup** utility, included in the **Xerox** software licensed with this book, be installed in a directory that is in your searchpath. Consult a local guru if you need help installing programs or modifying the command to work in your particular operating system.

Without going into too much detail, the command operates as follows:

1. The text in the corpus is piped to the **tr** utility.
2. **tr** translates all non-alphabetic characters into newlines, squeezing multiple newlines into a single newline, and pipes its output to **lookup**
3. **lookup** uses `mylanguage.fst` to look up each word, piping its output (in its own idiosyncratic format) to **grep**
4. **grep** acts as a filter, searching for and outputting only those lines containing `+?`, which is what **lookup** outputs as the pseudo-solution for a word when no solution is found
5. and finally **gawk** prints out the first field of the **lookup** output, which is the not-found word itself.

The result is output to a file, `failures.all`, containing all the words from `mycorpus.txt`, for which no solution at all was found by `mylanguage.fst`.

¹The backslashes in the example precede newlines and effectively tell the operating system to treat the command as if it were all typed on a single long line. You can literally type the command on a single line if you wish, but in that case the backslashes should not be typed.

Finite-state morphological analyzers are so fast that it is completely practical to test repeatedly against corpora containing millions of words. But even if you are saving only the failures, this can still result in unmanageably large output files, usually including many duplicates. Developers need some way to collapse the duplicates and identify which failures are the most important to fix first.

The simplest solution is to sort the collected failures by frequency of occurrence. If the failures are stored in `failures.all`, the following Unix idiom will produce `failures.sorted`, with the most frequent failures sorted to the top of the file.

```
cat failures.all | sort | uniq -c | sort -rnb > failures.sorted
```

The first `sort` will arrange all the not-found words alphabetically, which will have the effect of putting all multiple copies of the same word together in the output. If you are analyzing an English corpus and the word *canton* is missing three times, and the word *dahlia* is missing four times, the output of this first sort would look like this:

```
...
canton
canton
canton
dahlia
dahlia
dahlia
dahlia
...
```

These results are then piped to the **uniq** utility with the **-c** flag, which collapses multiple adjacent lines containing the same word into a single line with a count of the original number of lines, e.g.

```
3      canton
4      dahlia
```

These results are piped in turn to the **sort** utility, but this time with the flags **-rnb**, which cause the lines to be sorted in reverse numerical order, based on the first numerical field.

```
4      dahlia
3      canton
```

The most frequent not-found words end up at the top of the `failures.sorted` file. You can then examine the output file and concentrate on the top 100 or so most frequently missed words, either adding the necessary baseforms to the lexicon or

fixing the **lexc** description or the rules that prevent the forms from being analyzed. If the not-found words contain misspellings, then you may simply want to correct the test corpus. The addition of one new baseform will often resolve multiple missing surface words, so after fixing the top 100 missing words the whole test is usually best repeated to find the next 100 most important missing words. Retesting the whole corpus at reasonable intervals is also a valuable form of REGRESSION TESTING, highlighting errors that you inadvertently introduce while trying to fix your lexicons and rules.

Use the testing idioms to run even huge corpora through your morphological analyzer, looking for failures. Re-running the corpus at reasonable intervals is a useful form of regression testing.

If you know what you're doing, such commands are infinitely customizable to meet your particular needs. For example, the idiom shown above uses the **tr** utility to replace all non-alphabetic characters with the newline character, where alphabetic characters are designated by the built-in `:alpha:` notation, which is locale-specific. If necessary or desired, you can also specify the set of alphabetic letters explicitly by enumeration, as in the following example, which would be suitable for a language where the alphabet contains **A** to **Z**, **Ñ**, **Ä**, **Ï**, **Ü** (all in both cases) and the apostrophe `'`.

```
cat corpus | \
tr -sc "[A-Z]ÑÄÏÜ[a-z]'ñäïü" "[\n*]" | \
lookup lexical_transducer | \
grep '+?' | \
gawk '{print $1}' > failures_file
```

The essence of tokenization is simply to insert a newline character between tokens, and that can also be done using Perl or a similar scripting language. The Perl script in Figure 7.2, which requires Perl version 5.005.03 or higher, does a reasonable job of tokenization for English text, and any Perl hacker can customize it for a particular language. Unlike the **tr**-based tokenizers shown above, this solution separates and retains the punctuation characters as tokens. The script reads from standard input and writes to the standard output; so if it is stored in `tokenize.pl`

```
#!/usr/bin/perl -w
# requires Perl version 5.005_03 or higher
# you may need to edit the path above to point
#   to the Perl executable in your environment

while (<STDIN>) {
    chop ;
    s/^\s+// ;
    s/\s+$// ;
    /^$/ && next ;    # ignore blank lines

    # separate left punc(s) from beg of word
    s/^(["']+)(?=\w)/$1\n/g ;
    s/(?<=\W)(["']+)(?=\w)/$1\n/g ;

    # separate right punc(s) from end of word
    s/(?<=\w)(["',:;!]+)(?=\W)/\n$1/g ;
    s/(?<=\w)(["',:;!]+)$/\n$1/g ;

    # sep strings of multiple punc
    s/(?<=[^\w\s])([^\w\s])/\n$1/g ;

    # break on whitespace
    s/\s+/\n/g ;
    print $_, "\n" ;
}
```

Figure 7.2: A Perl Script that Tokenizes English Text. This script reads from standard input, writes to standard output, and preserves punctuation marks as tokens.

and made an executable file,² it can replace **tr** in the pipe.

```
cat corpus | \
tokenize.pl | \
lookup lexical_transducer | \
grep '+?' | \
gawk '{print $1}' > failures_file
```

The idioms for failure collection and sorting can of course be collapsed together into a single command, which is best typed into a file, edited to match your filenames, and run using the Unix utility **source**.

```
cat corpus | \
tr -sc "[:alpha:]" "[\n*]" | \
lookup lexical_transducer | \
grep '+?' | \
gawk '{print $1}' | \
sort | uniq -c | sort -rnb > sorted_failures
```

Because the sorting allows the linguist to concentrate on the most important missing words, this method is practical even with huge corpora, and new texts can be added to the test corpus at any time. Eventually, after many rounds of testing and lexical work, when the system is finding solutions for over 97% of the corpus words, the most frequently missed words will tend to be typographical errors, foreign words, proper names, and various rare nouns and adjectives, with most of the missing words occurring only once or twice even in a very large corpus. After this point, in any natural-language-processing system, it becomes increasingly difficult to improve the coverage significantly.

Testing for Failures with a Guesser

Another way to test for failures is to construct both a normal morphological analyzer and an auxiliary analyzer called a **GUESSER**. Guessers are built around a definition of phonologically possible roots, rather than enumerated known roots, and they are particularly valuable for testing in the early stages of a project, when the dictionary of known roots is small. The use of guessers, which requires a deeper understanding of the **lookup** application than we can approach here, is explained in Section 10.5.4.

lab:guessers

7.2.3 Positive Testing

As useful as the saving, sorting and repairing of failures is, it does not help to discover cases where the Lexical Transducer analyzes words incorrectly, i.e. over-

²The Unix command to make it executable is `chmod 755 tokenize.pl`. See your local guru for details.


```
Lexical: spit+Verb+PresPart
Surface: spiting
```

Figure 7.3: A Sin of Commission, Analyzing *spiting* Erroneously as the Present Participle of *spit*.

recognition or SINS OF COMMISSION, or where the analyzer finds some but not all of the valid analyses for a word, which we might term incomplete analysis or SINS OF PARTIAL OMISSION.

A typical sin of commission would be the English *spiting* being analyzed incorrectly as the present participle of the verb *spit* as in Figure 7.3. In reality, *spiting* is a form of the verb *spite*; the verb *spite* would properly have the present participle *spitting*, with two *ts*. The incorrect analysis of *spiting* as a form of *spit* will need to be fixed by the linguist, but the biggest problem is not usually fixing such problems but finding them in the first place. When any natural-language processing system contains tens of thousands of baseforms and analyzes and generates millions of surface forms, finding a *spit:spiting* problem is like searching for the proverbial needle in a haystack.

Similarly, sins of partial omission or incomplete analysis occur when a surface word is ambiguous, and the system does not find all of its valid readings. Consider the English words *table* and *tables*, which have both noun and verb readings. If the lexicon contains the noun baseform *table* but happens not to contain the verb baseform *table*, then even in an otherwise perfect system only the noun solutions will be found:

```
Lexical: table+Noun+Sg      table+Noun+Pl
Surface: table              tables
```

The saving and sorting of failures will of course not identify such cases where a correct but incomplete analysis is being performed.

While there is no perfect solution to the problem of flushing out sins of commission and partial omission, one useful daily exercise is the following:

1. During development, pick out a new paragraph or two of text every day and type it or store it in a separate file. The text should come from a variety of sources, for example the latest magazines, newspapers, a book you happen to be reading, a message that someone posted on the Internet, etc. In general, collect as much new text as you can find for the large test corpus, but choose just one or two random paragraphs of text for this daily exercise.
2. Run the paragraphs through the tokenizer and lookup, and then examine *all* the output by hand to see if the analyses are both correct and complete.

3. Fix the lexicons and rules as appropriate.

A useful idiom for such positive testing is the following, where the input file contains a smallish text.

```
cat corpus | \
tr -sc "[:alpha:]" "[\n*]" | \
lookup lexical_transducer > output_file
```

The output file can then be examined manually using a text editor. If you have the luxury of independent testers, make it part of their job to do such POSITIVE TESTING of real running text every day, entering manageably small texts and looking at absolutely everything that the system returns. It is a necessary balance to the NEGATIVE TESTING that looks only for failures.

```
The senator promised to table the bill
DET NOUN VERB PREP NOUN DET NOUN
```

Figure 7.4: Tagging Error Based on Incomplete Morphological Analysis

Inevitably, sins of partial omission will continue to present themselves for months and years to come. Sometimes they become obvious only after a part-of-speech disambiguator (“tagger”) is written, and the tagger is found to make mistakes as in Figure 7.4. Here we have a sentence using *table* as a verb, but the tagger is labeling it incorrectly as a *NOUN*. When tracking down such problems, it is often discovered that the morphological analysis simply lacks the verb baseform *table* and so is presenting the tagger with an unambiguous noun solution.

7.3 Checking the Alphabet

7.3.1 The Gap in Symbol-Parsing Assumptions

During development, a Lexical Transducer will often be found to contain mistakes in its alphabet, either missing symbols or strange, undesired symbols that are introduced by errors in the various source files. You should “check the alphabet” regularly to find such mistakes, which come from the following sources:

1. **xfst** and **twolc** make the assumption that any string of characters written together, e.g. *abc*, represents a single multicharacter symbol. **lexc**, for good lexicographical reasons, is the exception in making the opposite “exploding” assumption, i.e. that *abc* represents three separate symbols concatenated together; the only way to override the **lexc** exploding assumption is to declare written sequences of symbols as `Multichar_Symbols` to be kept together.

2. **xfst** and **twolc** treat a number of characters as special, including + (for Kleene Plus), * (for Kleene Star), | (alternation), & (intersection), ^ (iteration), etc. In **lexc**, on the other hand, such symbols are not generally special, unless they occur inside angle brackets, e.g. < a b* c >, where the formalism and so the special characters of regular expressions apply.

Even though there are good reasons for the different assumptions, the linguist who switches back and forth from **twolc** to **lexc** to regular expressions in **xfst** will often have trouble keeping the varying assumptions straight. Even within **lexc** files, the linguist has to juggle different assumptions inside and outside of angle brackets. These conflicting assumptions about how to parse typed strings into symbols are the root cause of most alphabet problems in finite-state systems.

```

LEXICON Adjectives
dark           Adj ;
quick         Adj ;
heavy         Adj ;

LEXICON Adj
+Adj:0        # ;

```

Figure 7.5: +Adj Intended to be a Multicharacter Tag

7.3.2 Failure to Declare Multichar_Symbols in lexc

The most common alphabet problem in **lexc** comes from the failure to declare all the intended Multichar_Symbols. If you write LEXICONS as in Figure 7.5, intending +Adj to be a multicharacter symbol, but failing to declare it in the Multichar_Symbols statement, **lexc** will happily and silently explode the string +Adj into four separate symbols, leading to the compilation of a **lexc** transducer having paths like the following:

```

Upper:      d a r k + A d j
Lower:      d a r k 0 0 0 0

```

If you analyze the string “dark” using this transducer (e.g. using **apply up** in **xfst**), the result string will look correct, i.e. it will be printed out superficially as “dark+Adj”, but it will in fact be wrong if you intended +Adj to be a single symbol. Note that the plus sign + is not special in **lexc**, except when it appears in a regular expression inside angle brackets; **lexc** will not treat the sequence +Adj as a multicharacter symbol unless it is declared. Failure to declare all the intended

multicharacter symbols in **lexc** is a common mistake that results in multiple normal symbols where you intended a single multicharacter symbol.

7.3.3 Inadvertent Declarations of Multicharacter Symbols in **xfst** and **twolc**

```
xfst[]: read regex [{talk} | {walk}] %+Verb:0
[ %+Bare:0 | %+3PSg:s | %+Past:ed | %+PresPart:ing ] ;
```

Figure 7.6: An **xfst** Regular Expression that Implicitly Declares *ed* and *ing* as Multicharacter Symbols. Such cases are almost always errors and can cause your lexical transducer to act mysteriously.

The opposite problem occurs in **xfst** and **twolc**, and in **lexc** regular expressions between angle brackets, when the linguist forgets that strings of characters written together are automatically parsed as names for multicharacter symbols. An **xfst** regular expression like the one in Figure 7.6 implicitly declares *ed* and *ing* as multicharacter symbols simply because they are written together without any separating spaces. **xfst** Replace Rules are also regular expressions, and the rule in Figure 7.7 implicitly declares *amus* as a multicharacter symbol.

```
xfst[]: define Rule1 %+Verb
%+Pres %+Indic %+Act %+1P %+Pl -> amus
```

Figure 7.7: An **xfst** Replace Rule that Inadvertently Declares the Multicharacter Symbol *amus*

In the **lexc** example shown in Figure 7.8, the sequences *ed* and *ing* appearing in regular expressions, inside **lexc** angle brackets, are similarly treated as single symbols and are automatically added to the alphabet of the resulting network. The grammars in both Figure 7.6 and Figure 7.8 will result in transducer symbol pairings like the following

```
Upper:      t a l k +Verb +PresPart
Lower:      t a l k 0      ing
```

where, again, *ing* is being stored and manipulated as a single symbol. Having multicharacter symbols like *ed*, *ing* and *amus* in your Lexical Transducer is almost always a Bad Thing, especially if subsequently composed rules are looking for the

```

Multichar_Symbols +Verb +Bare +3PSg +Past +PresPart

LEXICON Root
      Verbs ;

LEXICON Verbs
talk   V ;
walk   V ;

LEXICON V
+Verb:0 VEndings ;

! bad definition of verb endings

LEXICON VEndings
< %+Bare:0 > # ;
< %+3PSg:s > # ;
< %+Past:ed > # ; ! Error
< %+PresPart:ing > # ; ! Error

```

Figure 7.8: Unintentional Declaration of the Multicharacter Symbols *ed* and *ing* inside **lexc** Angle-Bracket Entries. Such declarations are almost always errors. The presence of such unintended multicharacter symbols in the alphabet of a network can often lead to mysterious problems in analysis and the application of rules.

symbol sequences [e d], [i n g] and [a m u s] that just aren't there.

7.3.4 When to Suspect an Alphabet Problem

You should *always* suspect alphabet problems. Alphabet problems are so common that experienced developers learn to check for them regularly, and idioms for this purpose are presented in the next section. The overt sign of an alphabet problem is when it looks like a rule should fire, but doesn't, or when it looks like a word should be analyzed, but isn't. The underlying problem is often that the alphabet is corrupted, causing your strings to be divided into symbols in unexpected ways.

For example, consider the **xfst** session in Figure 7.9 that builds a transducer for five regular English verbs and then tests it using **apply up**. The regular expression that defines the network inadvertently declares *ed* and *ing* as multicharacter symbols, adding them to the alphabet of the network.

The application routines (such as **apply up** and **apply down**) need to parse input strings into symbols, including multicharacter symbols, before matching them

against transducer paths. This symbol tokenization is always done relative to the sigma of the network being applied, and in cases of ambiguity they always give precedence to longer symbol matches. Thus if **e**, **d** and **ed** are in the alphabet, and the input is a string like “*edits*”, the input string is broken up into symbols as *ed-i-t-s* and is not found because no strings in the language begin with the multi-character symbol *ed*.

```

xfst[n]: clear stack
xfst[0]: read regex [{talk}|{walk}|{bark}|{snort}|{edit}]
%+Verb:0
[ %+Bare:0 | %+3PSg:s | %+Past:ed | %+PresPart:ing ] ;
xfst[1]: apply up talk
talk+Verb+Bare
xfst[1]: apply up walks
walk+Verb+3PSg
xfst[1]: apply up barked
bark+Verb+Past
xfst[1]: apply up snorting
snort+Verb+PresPart
xfst[1]: apply up edit
xfst[1]: apply up edits
xfst[1]: apply up edited
xfst[1]: apply up editing

```

Figure 7.9: A Typical Alphabet Problem. Because *ed* is inadvertently declared as a multicharacter symbol, input strings like “*edit*”, which contain *ed*, cannot be found. When a word looks like it should be analyzed, but mysteriously isn’t found, it is often a clue that the alphabet is corrupted with unintended multicharacter symbols.

Another not immediately obvious problem with the transducer in Figure 7.9 is that “*snorting*” is analyzed, but probably not as intended. Because *ing* is a multicharacter symbol in the alphabet, the input string “*snorting*” is broken up into symbols as *s-n-o-r-t-ing*; it is found because the lower-side language contains this string with the symbol *ing* on the end. Beginning developers often think that they can get away with a few multicharacter symbols of this type. However, when the verb stems *ring* and *ping* are later added to the lexicon, the presence of the multicharacter symbol *ing* in the alphabet would prevent any form of “*ping*” or “*ring*” from being analyzed. Inappropriate multicharacter symbols usually come back to haunt you.

The other classic manifestation of an alphabet problem is when a rule looks like it should apply to and modify a network, but doesn’t. Suppose that invoking **print random-lower** in **xfst** shows that the lower side of a network contains strings

such as “ringort” and that the rule

```
g -> 0 || n _ [o | a]
```

is subsequently applied to the lower side, with the intention of effecting the following mapping:

```
Upper: ringort
Lower: rinort
```

If it definitely looks like a rule should fire, but doesn’t, then it may be the case that the input string, printed out by default as “ringort”, might in fact consist of the symbols

```
r i n g o r t
```

In such a case, the rule cannot apply because there is no **g** symbol in the string.

It is almost always a mistake to declare multicharacter symbols like *ed* and *ing* that look like concatenations of normal alphabetic characters.³ Following the **Xerox** convention, which includes a punctuation mark like the plus sign (+) or the circumflex (^) in every multicharacter symbol can help avoid and correct errors.

7.3.5 Idioms for Checking the Alphabet

Learn to suspect and test regularly for alphabet problems. In particular, if it appears that a rule should be applying, but doesn’t, or that an input string should be accepted, but isn’t, the very first thing to do is to “check the alphabet”.

Printing Out the Sigma and Labels

The way to test a whole network for alphabet problems is simply to load it onto the top of an **xfst** stack and print out the labels.

```
xfst[0]: load stack MyLanguage.fst
xfst[1]: print labels
```

The result, which shows all the symbol *pairs* in the system, can be a bit confusing, so printing the **SIGMA**, i.e. the alphabet of symbol characters, is usually more readable.

```
xfst[0]: load stack MyLanguage.fst
xfst[1]: print sigma
```

³Justifiable exceptions may be appropriate for orthographies where ngrams represent single phonemes and cannot be confused with sequences of separate letters. For example, if a particular orthography uses **p**, **t** and **k** to represent unaspirated consonants and **ph**, **th** and **kh** to represent their aspirated counterparts, and if **h** is not used elsewhere in the orthography; then **ph**, **th** and **kh** are unambiguous and could be safely declared as multicharacter symbols.

To better localize alphabet problems, it is often preferable to look at the labels of the upper and lower sides separately as in Figure 7.10. Just scan the list of labels printed out; if you spot anything bizarre, like surface *ing* or *ed* being treated as single symbols, isolate the words in which they appear by composing simple filters.

```

xfst[n]: clear stack
xfst[0]: load stack MyLanguage.fst
xfst[1]: upper-side net
xfst[1]: print labels

xfst[n]: clear stack
xfst[0]: load stack MyLanguage.fst
xfst[1]: lower-side net
xfst[1]: print labels

```

Figure 7.10: Idioms to Examine the Labels on the Upper and Lower Side of a Transducer. These simple tests should be performed regularly to check the alphabet for missing and extraneous multicharacter symbols caused by errors in the source files.

If the network is found to contain an unexpected multicharacter symbol like *ing* on the lower side, the idiom in Figure 7.11 will isolate those words in which it appears, which can help you locate the error in your source files. Note that the idiom composes $\$(ing)$, which denotes the language of all strings containing the multicharacter symbol *ing*, on the lower side of the network, where the surface strings are visible.

```

xfst[0]: read regex @"MyLanguage.fst" .o. $(ing) ;
xfst[1]: print random-lower
xfst[1]: print random-upper

```

Figure 7.11: Idiom to Isolate Words Containing the Suspicious Multicharacter Symbol *ing* on the Lower Side of the Network

Once the offending strings have been identified, the fixes are usually pretty obvious: If *ing* is being treated, unexpectedly, as a multicharacter symbol, then fix the rule or other regular expression that is implicitly declaring it. Conversely, if you expect to see a particular multicharacter lexical symbol like *+Adj*, and it

doesn't appear in the alphabet of the upper-side language, then you should go back to your **lexc** description and declare it.

If you're not sure what to look for, identifying undeclared multicharacter symbols is slightly more difficult. If you follow the **Xerox** conventions for spelling multicharacter symbols and features, and if you see a separate plus sign (+) or circumflex (^) or square bracket in the alphabet of the upper-side language, then this is often a tell-tale sign that you have failed to declare either a tag like +F \circ or [+F \circ], or a feature like ^FEAT. To isolate and identify which multicharacter symbols and tags are undefined, compose a little filter on *top* of MyLanguage.fst that matches all and only strings containing the unexpected separate punctuation marks. This idiom is illustrated in Figure 7.12; the random strings printed out should provide enough information to help you track down which multicharacter symbols were not declared.

```
xfst[0]: read regex $[ %^ | %+ | %] | % ]
.0.
@"MyLanguage.fst" ;
xfst[1]: print random-upper
```

Figure 7.12: An Idiom to Help Find Undeclared Multicharacter Symbols. If you include a plus sign or a circumflex or square brackets as part of the spelling of every multicharacter symbol, then finding any of them as a separate symbol on the upper side of a network may be the sign of an intended multicharacter symbol that was not properly declared in **lexc**.

It is only a **Xerox** convention to include some kind of punctuation, e.g. a plus sign or circumflex, in the spelling of every multicharacter symbol. Another convention that has been used is to spell all tags with surrounding square brackets, e.g. [Noun]. Such conventions make multicharacter symbols stand out visually and can help to discover intended multicharacter symbols that were not properly declared.

Setting print-space ON

When working interactively with networks in **xfst**, developers often invoke **print random-upper** and **print random-lower** to see what the strings in a network look like. By default, these **print** routines print out connected strings to which the network could successfully be applied, but they do not show how those strings are

tokenized into symbols. When **print random-lower** is invoked as in the following example, no alphabet problem is apparent.

```

xfst[n]: clear stack
xfst[0]: read regex [{talk}|{walk}|{bark}|{snort}|{edit}]
%+Verb:0
[ %+Bare:0 | %+3PSg:s | %+Past:ed | %+PresPart:ing ] ;
xfst[1]: print random-lower
edits
edits
snorted
snort
bark
walk
barks
edit
snorting
walks
barked
walk
walking
talking
walked
xfst[1]:

```

If an alphabet problem is even slightly suspected, a good first step is to set the **xfst** interface variable **print-space** to **ON**. This causes the **print** routines to print a space between symbols, showing clearly which sequences are being tokenized as multicharacter symbols.

```

xfst[1]: set print-space ON
variable print-space = ON
xfst[1]: print random-lower
t a l k
b a r k e d
t a l k
e d i t e d
w a l k s
w a l k s
e d i t s
s n o r t
b a r k
w a l k
s n o r t s
b a r k
w a l k e d
t a l k s
e d i t i n g

```

With **print-space** set to **ON**, the suspicious multicharacter symbols *ing* and *ed* are hard to miss. For more information on the use of **print-space**, see Sections 8.4.2, 3.6.2 and 3.8.2.

7.4 Testing with Subtraction

A regular relation, encoded by a finite-state transducer, is a mapping between two regular languages, where (following **Xerox** conventions) the upper language is typically a set of strings that consist of a baseform and tags, and the lower language is a set of surface strings. Using the finite-state calculus, and particularly the subtraction operation in **xfst**, we can compare these languages against other languages that we or others have defined.

It should be remembered that regular relations (and the transducers that encode them) are not closed under subtraction. Regular *languages* are, however, closed under subtraction, and the following examples and idioms require extracting the upper-side or lower-side language from a transducer before subtraction is performed.

7.4.1 Testing the Lexical Language

All developers are strongly urged to plan out their analyses as much as possible before starting the coding. Alternatively, developers should feel free to experiment for a while and then restart the work after they have a better idea of the directions to take.

An important part of planning is to choose a reasonable tagset and to decide on the physical order in which co-occurring tags will appear. These choices can and should be formalized in a **LEXICAL GRAMMAR** that describes all possible legally constructed lexical strings. Lexical Grammars can be written in some cases as single monolithic regular expressions to be compiled by **read regex** inside **xfst**, but it is usually better to define them using **xfst** scripts, which are executed with the **source** utility. For a review of **xfst** scripts, see Section 3.3.3.

Let us assume that we want to define an upper-side language where the strings consist of a baseform followed by multicharacter-symbol tags. A regular expression that matches all possible baseforms will look something like the following:

```
[ a | b | c | d | e | f | g | h | i | j | k |
  l | m | n | o | p | q | r | s | t | u | v |
  w | x | y | z |
  A | B | C | D | E | F | G | H | I | J | K |
  L | M | N | O | P | Q | R | S | T | U | V |
  W | X | Y | Z ]+
```

This rather crude expression, which matches any string of one or more letters from an alphabet, should be modified, of course, to contain all and only the letters that can validly appear in the baseforms of your language. Following, i.e. concatenated after, these pseudo-baseforms are the legal strings of tags, described in a regular expression something like the following:

```
[
%+Noun ([%+Aug | %+Dim]) [%+Masc | %+Fem] [%+Sg | %+Pl]
|
%+Adj ([%+Comp | %+Sup]) [%+Masc | %+Fem] [%+Sg | %+Pl]
|
%+Verb [%+Past | %+Pres | %+Fut] [%+1P | %+2P | %+3P]
      [%+Sg | %+Pl]
]
```

In this illustration, we assume that nouns are optionally marked as augmentative or diminutive, obligatorily marked as masculine or feminine, and obligatorily marked as singular or plural via tags that appear in that order. Similar distinctions and orders are defined for adjectives and verbs. Of course, the tags, tag orders and overall tagging patterns will differ for each language, and there may be prefix tags appearing before the baseform. A real Lexical Grammar will be far more complex than the example above, perhaps extending to several pages.

```
clear stack
define InitialC [p|t|k|b|d|g|s|z|r|l|n|m|w|y] ;
define CodaC    [p|t|k|s|r|l|n|m|w|y] ;
define Vowel    [a|e|i|o|u] ;
define Syllable InitialC Vowel (CodaC) ;
define Baseform Syllable^{1,3} ;
define Nouns    %+Noun ([%+Aug | %+Dim])
[%+Masc | %+Fem] [%+Sg | %+Pl] ;
define Adjs     %+Adj ([%+Comp | %+Sup])
[%+Masc | %+Fem] [%+Sg | %+Pl] ;
define Verbs    %+Verb [%+Past | %+Pres | %+Fut]
[%+1P | %+2P | %+3P] [%+Sg | %+Pl] ;
define XolaLexGram Baseform [Nouns | Adjs | Verbs] ;
```

Figure 7.13: The **xfst** Script Xola-lexgram.script Defining a Lexical Grammar for an Imaginary Language, Including Baseforms with Constraints on Syllables, Possible Initial Consonants, and Possible Coda Consonants

The definition or “modeling” of pseudo-baseforms may also be much more constrained and sophisticated, reflecting known phonological constraints in the language. For example, Figure 7.13 shows a lexical grammar for the mythical

Xola language, in the form of an **xfst** script, wherein the baseforms consist of one to three syllables of form CV or CVC, and where there are constraints on which consonants can appear in initial and coda position.

The object of writing and maintaining a Lexical Grammar is to encourage you to define and conform to a consistent format for analysis strings; such consistency is absolutely essential to anyone, including yourself, who needs to use the Lexical Transducer to do generation.

```

xfst[n]: clear stack
xfst[0]: read regex < MyLanguage-lexgram.regex
xfst[1]: define LexGram
xfst[0]: read regex ["MyLanguage.fst"].u - LexGram ;
xfst[1]: print size
xfst[1]: print words

```

Figure 7.14: An Idiom to Check the Upper-Side Language Using a Lexical Grammar Defined in a Regular-Expression File. Note that the lexical grammar is subtracted from the extracted upper-side language of `MyLanguage.fst`. If the upper-side language is fully covered by the lexical grammar, then the network left on the stack will denote the empty language.

The Lexical Grammar is also a powerful tool for testing and debugging. Assume that your lexical transducer is written as a regular expression in file `MyLanguage.fst` and that your Lexical Grammar is in file `MyLanguage-lexgram.regex`. The idiom in Figure 7.14 subtracts the Lexical Grammar language from the upper-side language of `MyLanguage.fst`, leaving on the stack a network containing all the upper-side strings of `MyLanguage.fst` that do not conform, for whatever reason, to the Lexical Grammar as it is defined in `MyLanguage-lexgram.regex`. If the upper-side language is fully covered by the lexical grammar, then the result of the subtraction will be the empty language.

If you have defined your Lexical Grammar as an **xfst** script, as in Figure 7.13, then the idiom shown in Figure 7.15 will perform the appropriate check. Let us assume that the lexical-grammar script is stored in `Xola-lexgram.script` and that the lexical transducer to be tested is in `Xola.fst`. Running the script in Figure 7.13, using the usual **source** utility in **xfst**, causes the variable `XolaLexGram` to be defined.

However it is defined, the Lexical Grammar should produce a network that encodes a simple language, not a transducer. When that language is subtracted from the upper-side language of the lexical transducer, the result should be the empty language. In practice, this exercise often flushes out a set of lexical strings that do not conform, for whatever reason, to the lexical grammar. Any non-conforming strings should be examined manually, and any necessary corrections should be

```

xfst[n]: clear stack
xfst[0]: source Xola-lexgram.script
xfst[0]: read regex ["Xola.fst"].u - XolaLexGram ;
xfst[1]: print size
xfst[1]: print words

```

Figure 7.15: Idiom to Check the Lexical Language Using a Lexical Grammar Defined in an **xfst** Script File. In this case, the script (see Figure 7.13) computes the lexical-grammar network and defines the variable `XolaLexGram`. This `XolaLexGram` is then subtracted from the upper-side language of `Xola.fst`.

done. It may, of course, be necessary to correct your **lexc** and **xfst** files, or it may be necessary to augment and correct your Lexical Grammar, especially after you introduce new parts of speech or any changes in the tags.

The Lexical Grammar will naturally grow and need modification during development. Don't be afraid to change it as necessary, but do use it throughout development to keep your lexical-level strings as consistent and beautiful as possible. Finally, put commented examples inside the Lexical Grammar source file; this file, without modification, should serve as an important part of the final documentation of your system. Anyone wanting to use your system for generation will have to know exactly how the upper-side strings are spelled.

Do not try to subtract in the other direction, computing the Lexical Grammar minus the upper side of your lexical transducer. Because your Lexical Grammar accepts strings based on any possible baseform, its language covers a huge or even infinite number of strings, and the difference will also be huge or infinite in size.

7.4.2 Testing the Surface Language

Just as the lexical language of a transducer can be extracted and compared against a language defined in a Lexical Grammar, the surface language of a Lexical Transducer can also be extracted and compared against other suitable languages. These comparison languages can come from external wordlists, external lexicons, and previous versions of your own system. Comparison against previous versions of your own system is known as regression testing.

Testing Against Wordlists

A wordlist—a plain list of surface words, usually extracted from a corpus—is not often of huge value in computational linguistics. But many such lists have been prepared, often to support primitive spelling checkers, and they can often be acquired free from the Internet. A plain wordlist is seldom worth buying, but where one is available, it can be used profitably to test the surface side of your Lexical Transducer.

Wordlists can be viewed as attempts, very primitive and inadequate attempts, to model a language by simply enumerating all the possible strings. You can automatically create your own wordlist from any corpus, simply by tokenizing it, one word to a line, and **sort**-ing it and **uniq**-ing to remove duplicates.

```
cat corpus | \
tr -sc "[:alpha:]" "[\n*]" | \
sort | uniq > wordlist
```

One way to test our Lexical Transducers on a wordlist is to simply pipe the wordlist to the **lookup** utility, which expects its input to have one token to a line. The failures can be collected and manually reviewed as usual, and your lexicons and rules can be modified, if appropriate, to accept them.

```
cat wordlist | \
lookup lexical_transducer | \
grep '+?' | gawk '{print $1}' > failures
```

However, we can get more from a wordlist, faster, by compiling it into a network and using subtraction to compare it against the surface language of our Lexical Transducer. Consider a plain wordlist that starts like this:

```
aardvark
about
apple
day
diaper
dynasty
easy
finances
grapes
grovel
...
```

We could tediously edit the wordlist into a suitable **lexc** file or even a regular expression for compilation into a network, but the need to compile a network from a plain wordlist is common enough that **xfst** offers the **read text** utility for exactly that purpose. **read text** takes the name of a wordlist file, containing one word per

line, explodes the strings into separate symbols just like **lexc**, builds a network that accepts all and only the strings in the wordlist, and pushes that network on the **xfst** stack.

The first way, and sometimes the only practical way, to use the wordlist language is to subtract from it your surface language, i.e. to compute

```
wordlist - YourSurfaceLanguage
```

An **xfst** idiom to do this is shown in Figure 7.16. We assume again that our Lexical Transducer is in file `MyLanguage.fst` and that the wordlist is in the file `wordlist.txt`.

```
xfst[n]: clear stack
xfst[0]: read text < wordlist.txt
xfst[1]: define WordList
xfst[0]: read regex WordList - [@"MyLanguage.fst"].l ;
xfst[1]: write text > words-missing.txt
```

Figure 7.16: Idiom to Subtract the Lower-Side Language of a Network from a Wordlist Language. The remaining language, which can be printed out as a wordlist file, is the set of words in the wordlist that are not analyzed by the transducer.

However it is computed, the difference language will contain a set of surface words that are in the wordlist but are not in the surface language of the Lexical Transducer. You can use the **write text** utility to write these words out to file. These words should be reviewed carefully for possible addition to your Lexical Transducer. Because publicly available wordlists come from who-knows-where and were built from who-knows-what corpora, they are notorious for containing garbage.

Depending on the size of the wordlist—the larger the better—you can also perform the reverse subtraction and get manageable results, as shown in Figure 7.17. This subtraction will yield a list of all words that are in your surface language but which were not in the wordlist. This *may* flush out some bad strings that have crept into your surface language via errors of various sorts. But if the wordlist language is too small, or if your surface language is exceptionally large (especially if you have a compounding language with an infinite number of strings), then this second test becomes impractical.


```

xfst[n]: clear stack
xfst[0]: read text < wordlist.txt
xfst[1]: define WordList
xfst[0]: read regex [@"MyLanguage.fst"].1
- WordList ;
xfst[1]: print size
xfst[1]: write text > words-extra.txt

```

Figure 7.17: Idiom to Subtract the Wordlist Language from the Surface Language of a Lexical Transducer. For this test to be practical, the lower-side language of `MyLanguage.fst` should not be infinite or too much larger than the language of `wordlist.txt`.

A corpus of running text is generally more valuable for testing than a plain wordlist. With a corpus, you can sort the not-found words by frequency of occurrence, allowing you to concentrate on fixing the most important gaps in your system; and you can search the corpus to see the context of mysterious alleged words. With a wordlist, all the not-found words are just disembodied strings of characters, with no information as to frequency or context.

Testing Against Previous Versions

Morphology systems get big and complex, to the point where it is humanly impossible to predict all the implications of adding, deleting and changing rules or lexical entries. Luckily, the finite-state calculus gives us powerful ways of comparing the coverage of different versions of our own systems, which is the essence of regression testing.

For testing purposes and for safety, you should keep backups and work within a version-control system (see Chapter 6). From time to time, the system will reach a relatively stable state, and at such times you should take a “snapshot” of the entire status of your source files. You should *always* take a snapshot when any kind of contractual milestone is reached or a delivery is made. The version-control system will then allow you to restore that state at a later time.

Here’s a typical scenario: Assume that your system reached a relatively stable state, and that you took a snapshot of it as `Version1`. A week or a month later, you have added a few lexical items, added three rules, changed several other rules, and you wonder if you’ve broken anything in the process.

Assume that the new lexical transducer is named `new.fst`, and that you saved

a copy of the previous Version1 transducer as `old.fst`. If you forgot to save a copy, use the version-control system to reconstruct it; that's what a version-control system is for. Figure 7.18 shows an **xfst** idiom for seeing which surface words have been lost going from `old.fst` to `new.fst`.

```
xfst[n]: clear stack
xfst[0]: read regex [@"old.fst"].l - [@"new.fst"].l ;
xfst[1]: print size
xfst[1]: print random-lower
xfst[1]: write text > words-lost.txt
```

Figure 7.18: Regression Testing, Comparing Two Versions to Find Lost Surface Words

Creating the output file may not be necessary. Recall that you can use **print size** to see how many words were lost, if any. If the number is not zero, then you might want to look at a few of the words using **print random-upper** or **print random-lower**; or, if the number is very small, use **print words** to see them all displayed on the screen.

To see which surface words were added in going from `old.fst` to `new.fst`, simply do the reverse subtraction as shown in Figure 7.19.

```
xfst[n]: clear stack
xfst[0]: read regex [@"new.fst"].l - [@"old.fst"].l ;
xfst[1]: print size
xfst[1]: print random-lower
xfst[1]: write text > words-added.txt
```

Figure 7.19: Regression Testing, Comparing Two Version to Look for Added Words

Testing against your own previous system is often the only practical way to catch regressions (degradations) and introduced errors. After performing the subtractions, if all the words in the file `words-lost.txt` are in fact bad words, then the system has improved. And if all the words in `words-added.txt` are good words, then again the system has improved. But if you've lost any good words, or gained any bad words, you need to fix the current files and rerun the tests until you see all improvement and no degradation. At that point, if your new system is generally stable, it is a good idea to take a new snapshot and save the current Lexical Transducer as `old.fst` for the next round of regression testing.

Learn to use your local version control system, even if it seems like irritating overhead. The ability to resurrect the previous stable system and compare it against your new system is often the only practical way to do regression testing.

Testing Against External Lexicons

Occasionally you can buy or find externally produced LEXICONS that can also be used for testing against your system's surface language, where by lexicon we mean an on-line resource that is more informative than a bare wordlist, such as a list of baseforms with category and feature information. The more auxiliary information a lexicon contains, the more potentially valuable it is. Such externally produced lexicons can be a gold mine for testing and development.

- If you can extract or generate a list of surface words from an external lexicon, that list can be used like any other wordlist for testing against your system's surface language.
- If you can establish the equivalence between the external lexicon's categories and your own, you may be able to extract large numbers of lexical entries from the lexicon, edit them with macros, and add them semi-automatically to your own lexicons.
- If the external lexicon contains part-of-speech and feature information, you should be able to construct testing scripts that check to see if your system analyzes the words and assigns equivalent tags and features.

Comparing your system against another carefully produced lexicon is one of the few effective ways to find sins of partial omission, e.g. when your system analyzes *table* as a noun but not also as a verb. Although any lexicon will have mistakes and omissions, the chances of two independently produced lexicons having the same mistakes and errors are relatively low.

For example, a wordlist of words that are identified in the external lexicon as nouns can be compared profitably against the set of surface nouns in your system. Figure 7.20 shows the idiom for extracting the set of surface nouns (as a language) from your system, which we'll assume is stored in `MyLanguage.fst`. Let's also assume that the list of alleged nouns extracted somehow from the external lexicon is called `nounlist.txt`. The idiom effectively identifies a set of words, alleged to be nouns in another dictionary, that are not analyzed as nouns by your lexical transducer. The left-over words should be considered carefully for addition, as nouns, to your lexicons.

```

xfst[n]: clear stack
xfst[0]: define MyLanguageNouns [ ${%+Noun}
.o.
@"MyLanguage.fst"].l ;
xfst[0]: read text < nounlist.txt
xfst[1]: define NounList
xfst[1]: read regex NounList - MyLanguageNouns ;
xfst[1]: print size
xfst[1]: print random-lower
xfst[1]: write text > nouns-to-add.txt

```

Figure 7.20: Extracting Nouns from a Lexical Transducer for Comparison to a Wordlist of Nouns Extracted from Another Dictionary. The subtraction of `MyLanguageNouns` from `Nounlist` will leave on the stack a language of alleged nouns that are not analyzed as nouns by the transducer.

If the noun wordlist is comparable in size to the noun coverage of your system, then the reverse subtraction may also be possible and interesting, identifying alleged nouns analyzed by your system that are not included in the external list of nouns. This is the way to find if your transducer is generating any ill-formed nouns.

Typically, both the external wordlist and your analyzer will be in error in various ways, but the probability that both will err in exactly the same way is low. This is what makes cross-wordlist or cross-lexicon testing so valuable; it subtracts one haystack from another, leaving relatively little hay and a manageable assortment of needles to examine.

The more finely an external lexicon is coded, the better it is for testing against our finite-state systems. Two separate Spanish noun lists, for example, one of feminine nouns and one of masculine nouns, are more valuable than one list of undifferentiated Spanish nouns. The testing of a list of feminine Spanish nouns against a full Spanish lexical transducer can be done exactly like the `MyLanguageNouns` test in Figure 7.20, except that one starts by extracting just the feminine nouns from `Spanish.fst`, as in Figure 7.21. Simple comparison, via subtraction, of what your system calls feminine Spanish nouns from what another system calls feminine Spanish nouns can easily highlight gaps and mis-codings.

A decent lexicon with precise subcodings can often be split automatically, e.g. with Perl scripts, into dozens of separate wordlists for comparison against the words and analyses in your own system. Acquire any good lexicons and labeled wordlists that you can, everything from lists of people, cities and companies to full-scale lexicons from other NLP projects. A list of German compound nouns is valuable; a list of Spanish verb infinitives, with each verb marked for its conju-

```
xfst[n]: clear stack
xfst[l]: define FemNounList [ ${%+Noun ?* %+Fem}
.o.
@"MyLanguage.fst"].l ;
```

Figure 7.21: Extracting Feminine Nouns Only. This example assumes that the tags are on the upper side and that the +Noun tags precede the +Fem tags.

gation class, is a gift; and a detailed lexicon from another serious natural-language-processing project can be priceless. A carefully constructed lexicon of baseforms, with detailed marking of inflectional possibilities, conjugation types and derivational possibilities can often be converted semi-automatically into real **lexc** format. Such lexicons, usually jealously guarded by their owners, can be extremely valuable for accelerating a development project, saving you person-years of tedious lexicographical development.

Unfortunately, lexicons vary tremendously in quality, coverage, format and terminology; and they are notoriously difficult to acquire for commercial purposes. Each case is unique, usually requiring some detective work to interpret and find the information you need, and you may need to depend heavily on local system gurus to write conversion scripts. But while it is never easy, “mining” an independently produced lexicon of good quality is one of the most valuable exercises you can perform for testing.

Chapter 8

Flag Diacritics

Contents

8.1	Features in Finite-State Systems	440
8.2	What Are Flag Diacritics?	441
8.2.1	Basic Concepts	441
8.2.2	Implementation of Flag Diacritics	441
8.2.3	Some Practical Points	441
8.2.4	How Flag Diacritics Work	442
8.3	Using Flag Diacritics	443
8.3.1	U-Type or Unification Flags	443
	Unification Flag Names	443
	Why the Funny At-Sign (@) Spelling?	444
	Using Unification Flags in Arabic	444
	Using Flag Diacritics and Upper-Side Tags	450
	How Analysis and Generation Routines Interpret Flags	453
8.3.2	The Full Range of Flag-Diacritic Operators	455
	P or Positive (Re)Setting	455
	N or Negative (Re)Setting	455
	R or Require Test	456
	D or Disallow Test	456
	C or Clear Feature	457
	U or Unification Test	457
8.4	Flag Diacritics and Finite-State Algorithms	457
8.4.1	Analysis and Generation	457
	xfst apply up and apply down	458
	lexc lookup and lookdown	458
8.4.2	Seeing Flag Diacritics in Networks	459
	Checking the Alphabet	459

	Using the xfst print Commands to see Flags	459
	The obey-flags Variable	460
	The show-flags Variable	460
	The print-space Variable	460
	Flag Diacritics, Variables, and Debugging	460
8.4.3	Eliminating Flag Diacritics	461
8.4.4	Flag Diacritics and Composition	463
	lexc Composition	463
	xfst Composition	463
8.4.5	Flag Diacritics and compile-replace	465
8.4.6	Reflecting Flag Diacritics Side-to-Side	465
8.5	Examples	466
8.5.1	French Articles	466
8.5.2	Using the R-type Flag Diacritics	467
8.5.3	Using P-type Flag Diacritics for Positive (Re)Set	468
8.5.4	Using Flag Diacritics to Constrain Circumfixes	469
8.5.5	Finnish Proper Names and Derivatives	471
	The Problem	471
	A Solution	471
8.5.6	Forward-Looking Feature Requirements	474

8.1 Features in Finite-State Systems

Flag Diacritics are an extension of the **Xerox** finite-state implementation, providing feature-setting and feature-unification operations that help to keep transducers small, enforce desirable constraints, and simplify grammars. Flag Diacritics are often used to enforce separated or “long-distance” constraints on the co-occurrence of morphemes within words, constraints which are awkward to handle in regular expressions or **lexc** alone. The use of Flag Diacritics is therefore an alternative to composing such constraints into the transducer using rules or filters, a practice that can sometimes cause an explosion in the size of the resulting transducer. Flag Diacritics can also be useful for marking roots for idiosyncratic morphotactic behavior, constraining circumfixes, and generally handling other constraints that are feature-based rather than phonological.

Several implementations of finite-state morphology have added full-scale feature-unification packages that represent significant overhead and can result in unacceptably slow analysis and generation. Flag Diacritics represent a simplified, lightweight approach designed for speed.

8.2 What Are Flag Diacritics?

8.2.1 Basic Concepts

As far as **lexc**, **twolc**, **xfst** regular expressions and networks themselves are concerned, Flag Diacritics are just normal multicharacter symbols that can appear in strings. Flag Diacritics do, however, have a distinctive spelling and are treated specially by application routines that have been written to be sensitive to them: First, Flag Diacritics are treated like epsilons when a network is applied to input; they are not matched against the symbols of the input string, and they do not appear in output strings. Second, the application routines themselves, i.e. analysis and generation, notice and interpret Flag Diacritics as feature setting and feature-unification operations that constrain at runtime the possible paths followed through a network.

A network containing Flag Diacritics typically contains many illegal paths that would normally result in overgeneration and overrecognition. These illegal paths can be literally removed from the network by composing the network with finite-state filters, but only at the cost of a significant increase in the size of the resulting network (see Section 6.3.3). The use of Flag Diacritics allows the illegal paths to be blocked at runtime by the analysis and generation routines, keeping the transducer small.

8.2.2 Implementation of Flag Diacritics

This chapter explains the implementation of Flag Diacritics by Lauri Karttunen at **XRCE**, as reflected in the software available with this book.¹

8.2.3 Some Practical Points

Experience has shown that students have considerable difficulty understanding and using Flag Diacritics. Most finite-state computing has been done without any use of Flag Diacritics at all, and we have avoided all discussion of them until now. But Flag Diacritics are extremely useful in the hands of an expert developer and may be essential to keep transducers to a manageable size. The following observations may help:

1. You should definitely use Flag Diacritics if your words contain separated dependencies that are likely to lead to abnormally large networks.
2. You can include Flag Diacritics from the beginning of your project, or add them at any time, without committing your final system to this approach; it is always easy to eliminate Flag Diacritics from a network. Adding Flag

¹Developers at **PARC** and at the **Xerox** spinoff **Inxight** have their own implementations of Flag Diacritics that differ in many details.

Diacritics *post hoc* to an existing system can require non-trivial re-editing of your source files.

3. In form, a Flag Diacritic is just a multicharacter symbol. In **lexc**, they should be declared in the `Multichar_Symbols` section.
4. Inside a network, Flag Diacritics are multicharacter symbols that have a special meaning and constraining effect *only* at runtime when the network is applied using analysis and generation routines that are “flag-aware”. **xfst**, **lexc** and the command-line **lookup** utility (see Section 10.3) are flag-aware or sensitive to Flag Diacritics.
5. Flag Diacritics are inserted strategically in a network by the developer as part of the abstract spelling of strings. Typically, you want to think of a Flag Diacritic as being part of the abstract spelling of a morpheme or as a prefix or suffix to a class of morphemes.
6. For reasons that will be explained below, Flag Diacritics should typically be two-sided symbols; i.e. they should be visible on both the upper and lower sides of a path in the transducer if it is desired to have parallel restrictions in both analysis and generation.

Danger: Do not confuse Flag Diacritics with the deprecated (i.e. obsolete and highly discouraged) Diacritics of **twolc**. They are completely different.

8.2.4 How Flag Diacritics Work

When analysis and generation routines recognize and use Flag Diacritics at runtime, they are essentially cheating, adding a bit of memory to the finite-state paradigm. Normally when applying a finite-state network, the transition from one state to the next depends only on the current state and the next input symbol, and there is no stack or other memory that can be consulted to constrain which paths are followed next. But with Flag Diacritics, the application routines can “remember”, during the course of application, useful bits of information in the form of feature-value settings, and the application routines can then use that information to constrain which subsequent paths are followed.

Traversing an arc with a Flag Diacritic is like an epsilon transition but is conditional on the success or failure of an OPERATION specified by the Flag Diacritic. The result depends on the state of a feature register that is initialized in the beginning of the analysis or generation and is continuously updated along each path

that is being explored. The flag-aware application routines perform on-the-fly feature setting and unification at runtime, remembering and carrying along the unified feature sets as part of the analysis or generation process.

8.3 Using Flag Diacritics

To illustrate the use of Flag Diacritics, we will start with the simplest possible examples, using the simplest and most commonly used U-TYPE flags. Then we will continue with a description of the spelling and semantics of the whole range of Flag Diacritic subtypes.

8.3.1 U-Type or Unification Flags

Unification Flag Names

Flag Diacritics are distinguished from other multicharacter symbols by their spelling. The simplest and most commonly used form of Flag Diacritic is the **U**-type or Unification-type, spelled according to the following template:

```
@U.feature.value@
```

That is, the Flag Diacritic is spelled with an initial at-sign, the uppercase letter **U** (standing for the Unification operation), a period, a *feature* string chosen by the developer, another period, a *value* string chosen by the developer, and a terminating at-sign. The *feature* and *value* strings cannot include the dot or period character itself. Uppercasing vs. lowercasing in the strings is significant. For example:

```
@U.CASE.NOM@
@U.CASE.ACC@
@U.CASE.GEN@
@U.num.sing@
@U.num.plur@
@U.gender.masc@
@U.gender.fem@
```

The *feature* and *value* names have no inherent meaning and are chosen purely for the convenience of the developer. Intuitively, the three **CASE** examples above might be defined to mark case endings that are nominative, accusative and genitive respectively. The **num** examples might mark morphemes that are singular vs. plural; and the **gender** flags might mark morphemes that are masculine vs. feminine. Marking a morpheme with a Flag Diacritic means simply including the Flag Diacritic as an additional symbol in the spelling of that morpheme. Other varieties of Flag Diacritics besides the U-type will be discussed below.

In your grammars and in the networks themselves, Flag Diacritics are just normal multicharacter symbols. It is the application routines like **apply up** and **apply down** that recognize them, by virtue of their distinctive spelling, and treat them specially.

Why the Funny At-Sign (@) Spelling?

The spelling of Flag Diacritics with surrounding at-signs, as in @U.FOO.X@, simply mimics an earlier implementation of Flag Diacritics used at **Inxight**.²

XRCE Flag Diacritics do not need to be declared in any way other than as standard multicharacter symbols. At runtime they are recognized and used as Flag Diacritics by virtue of their distinctive spelling alone.

Danger: The at-signs (@) and periods used in the spelling of Flag Diacritics are special characters in regular expressions and inside the angle-bracket entries of **lexc**, and so they must be literalized in these environments by using prefixed %-signs or by double-quoting the entire Flag Diacritic. The spelling of Flag Diacritics in regular expressions is therefore a bit awkward, and failure to literalize the special characters is a very common error.

Using Unification Flags in Arabic

A very common case in natural-language morphotactics is that one morpheme, say a particular prefix, cannot appear in the same word with another morpheme, say a particular suffix. Such affixes cannot co-occur in the same word. The traditional and most straightforward way to use Flag Diacritics is to mark such incompatible morphemes with U-type Flag Diacritics that have the same *feature* but different, incompatible *values*.

For example, Arabic noun stems like *kitaab* (“book”) can usually take any one of six case endings:³

²In fact, it was not originally intended that these odd strings should be written or even seen by developers; rather it was assumed that developers would declare sets of incompatible features and that these would be converted automatically into the internal spellings using at-signs. The **XRCE** implementation now provides a wider range of flag operations than the **Inxight** implementation, going beyond the original constraints on mutual co-occurrence, but the at-sign spelling was retained to avoid arbitrary differences between the **Inxight** and **XRCE** implementations.

³In Arabic, the indefinite case endings are phonologically /un/, /an/ and /in/, but in the standard orthography they are written, if at all, with special diacritical symbols; for the indefinite accusative ending /an/, the standard orthography also requires the addition of an ʔalif letter in most cases. For

kitaab+u	definite nominative
kitaab+a	definite accusative
kitaab+i	definite genitive
kitaab+uN	indefinite nominative
kitaab+aN	indefinite accusative
kitaab+iN	indefinite genitive

The definite article *al-* in Arabic is a prefix that attaches orthographically to the front of a noun; if it is present, then only the definite case endings are possible. Thus the word *alkitaabu* is well-formed, but **alkitaabuN* is ill-formed. From the negative point of view, we say that the definite-article prefix *al-* is incompatible with the indefinite case suffixes. To constrain this long-distance dependency, it suffices to include a Flag Diacritic symbol like @U.ART.PRESENT@ (intended to suggest to the linguist “the definite article is present”) as part of the spelling of the *al-* morpheme and to include an incompatible Flag Diacritic like @U.ART.ABSENT@ (intended to suggest “the definite article is/must-be absent”) as a symbol in each of the indefinite case endings.

In a **lexc** source file, the entry for *al-* might look like this

```
LEXICON Article
al@U.ART.PRESENT@      Stems ;
```

while the entries for the case endings might be notated as

```
LEXICON Case
u      # ;
a      # ;
i      # ;
@U.ART.ABSENT@      IndefCase ;

LEXICON IndefCase
uN     # ;
aN     # ;
iN     # ;
```

The entire **lexc** mini-grammar could then be written as in Figure 8.1. Note that the Flag Diacritics are declared as normal multicharacter symbols. The one stem *kitaab* will be used here to represent tens of thousands of noun stems, and they are compiled into a subnetwork that we will represent as in Figure 8.2.

The **FST** compiled from this grammar, as shown in Figure 8.3, will include paths like the following, which, ignoring the Flag Diacritics, represent morphotactically well-formed words in Arabic.

the purposes of the present example, the representations of both the Arabic definite-article prefix and the case endings are simplified.

```

Multichar_Symbols @U.ART.PRESENT@ @U.ART.ABSENT@ uN aN iN

LEXICON Root
  Article ;

LEXICON Article
al@U.ART.PRESENT@ Stems ; ! optional article prefix
                   Stems ; ! empty string entry

LEXICON Stems
kitaab      Case ; ! one stem to represent tens of
              ! thousands

LEXICON Case
u           # ;
a           # ;
i           # ;
@U.ART.ABSENT@ IndefCase ;

LEXICON IndefCase
uN          # ;
aN          # ;
iN          # ;

```

Figure 8.1: An Arabic Grammar that Uses Flag Diacritics to Constrain Which Case Suffixes can Appear on Nouns Marked with the Explicit Definite-Article Prefix. Without the definite-article prefix, a noun stem can accept any one of the six case suffixes. If the definite-article prefix is present, indefinite-case suffixes are blocked at runtime.

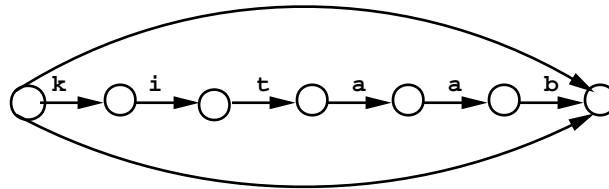


Figure 8.2: Abbreviated Representation of the Subnetwork Containing all Arabic Noun Stems

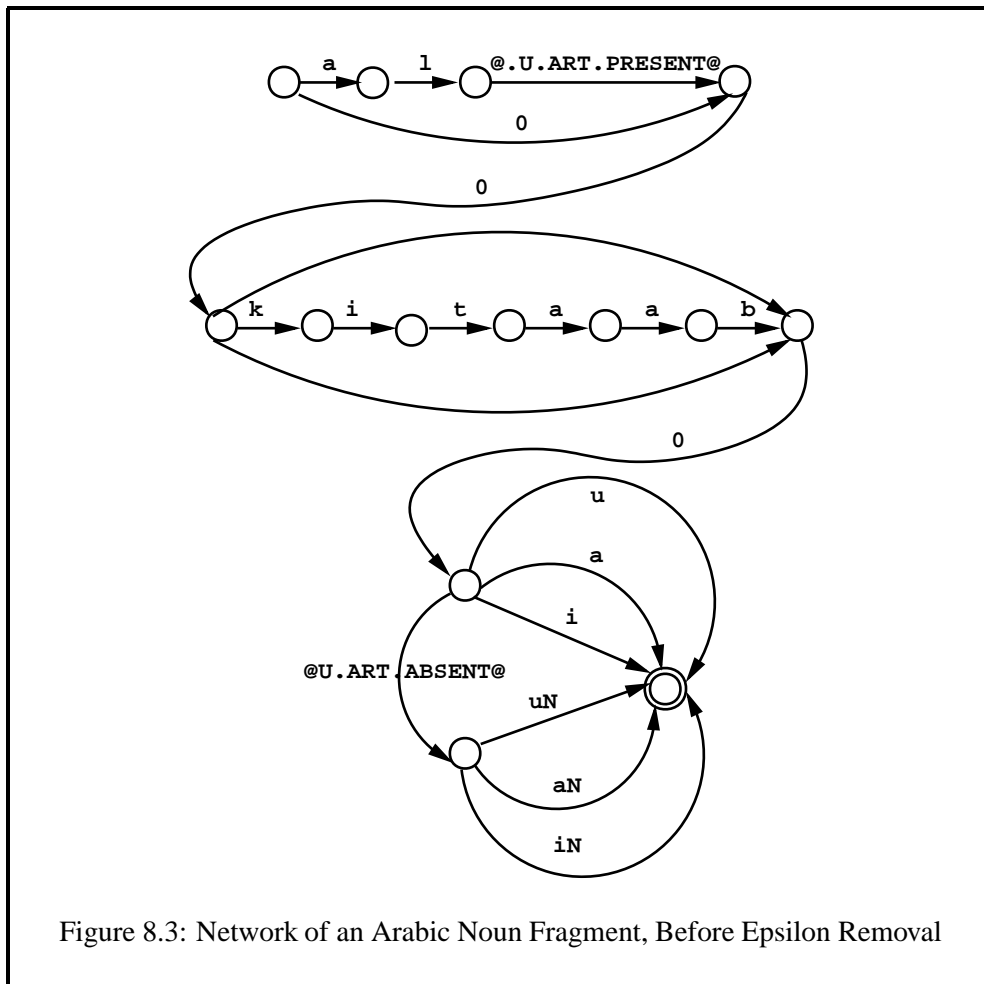


Figure 8.3: Network of an Arabic Noun Fragment, Before Epsilon Removal

```

kitaabu
kitaaba
kitaabi
kitaab@U.ART.ABSENT@uN
kitaab@U.ART.ABSENT@aN
kitaab@U.ART.ABSENT@iN
al@U.ART.PRESENT@kitaabu
al@U.ART.PRESENT@kitaaba
al@U.ART.PRESENT@kitaabi

```

As already mentioned, the Flag Diacritics in the network are indeed ignored at application time when it comes to matching input characters or producing output—they are treated like epsilons. Thus the **FST** will recognize input words such as *kitaabu*, *kitaabuN* and *alkitaabu*.

However, the **FST** also contains the following paths, which correspond to morphotactically ill-formed words of Arabic that contain both the definite-article prefix and an indefinite case suffix.

```

al@U.ART.PRESENT@kitaab@U.ART.ABSENT@uN
al@U.ART.PRESENT@kitaab@U.ART.ABSENT@aN
al@U.ART.PRESENT@kitaab@U.ART.ABSENT@iN

```

Notice, however, that each of these illegal paths contains two Flag Diacritics, @U.ART.PRESENT@ indicating that the article is present, and the semantically incompatible @U.ART.ABSENT@ indicating that the article is (or must be) absent. The values clash; they cannot be unified. The flag-aware **lexc lookup** and **look-down** routines, and the **apply up** and **apply down** utilities in **xfst**, recognize this incompatibility at runtime and block any analyses corresponding to these paths. If the same restrictions were imposed by composing in filters, the entire subnetwork of noun stems would be duplicated, almost doubling the size of the final network as shown in Sections 4.5.3 and 6.3.3.

The real situation is even more complicated. Arabic also allows the prefix *bi-*, with a prepositional meaning, to attach to the front of a noun; and if both the preposition *bi-* and the article *al-* are present, then the *bi-* must come before the *al-*, e.g. *bialkitaabi*. The complication is that *bi-* governs the genitive case, meaning that if *bi-* is present, then the nominative and accusative case suffixes are incompatible. This is another straightforward case of long-distance co-occurrence restrictions among morphemes that can be handled with simple U-type Flag Diacritics. If the same restrictions were imposed by composing in filters, the size of the final network would multiply yet again.

As *bi-* governs the genitive, we will mark it with a Flag Diacritic that sets **CASE** to **GEN**. (As always, the developer is free to spell the feature name and value name in any way that he or she finds mnemonically helpful.)

```

bi@U.CASE.GEN@

```



```

Multichar_Symbols @U.ART.PRESENT@ @U.ART.ABSENT@
                  @U.CASE.NOM@ @U.CASE.ACC@ @U.CASE.GEN@ uN aN iN

LEXICON Root
      Preposition ;

LEXICON Preposition
bi@U.CASE.GEN@   Article ; ! optional preposition prefix
                  Article ; ! empty string entry

LEXICON Article
al@U.ART.PRESENT@ Stems ;   ! optional def. arti-
cle prefix
                  Stems ;   ! empty string entry

LEXICON Stems
kitaab Case ; ! one stem representing tens of thousands

LEXICON Case
u@U.CASE.NOM@           # ;
a@U.CASE.ACC@           # ;
i@U.CASE.GEN@           # ;
@U.ART.ABSENT@         IndefCase ;

LEXICON IndefCase
uN@U.CASE.NOM@         # ;
aN@U.CASE.ACC@         # ;
iN@U.CASE.GEN@         # ;

```

Figure 8.4: An Arabic Grammar that Uses Flag Diacritics to Constrain both Indefinite Case Endings and Case Governing by *bi-*

In addition, we will also need to mark the case suffixes with @U.CASE.NOM@ (nominative), @U.CASE.ACC@ (accusative), or @U.CASE.GEN@ as appropriate. In fact, marking the genitive suffixes with @U.CASE.GEN@ is formally unnecessary in this case, but it will do no harm. The extended grammar is shown in Figure 8.4; note again that the Flag Diacritics are declared as normal multicharacter symbols.

The **FST** compiled from this grammar will include valid paths like the following, which, when the Flag Diacritics are ignored, correspond to well-formed words:

```
bi@U.CASE.GEN@kitaabi@U.CASE.GEN@
bi@U.CASE.GEN@kitaab@U.ART.ABSENT@iN@U.CASE.GEN@
bi@U.CASE.GEN@al@U.ART.PRESENT@kitaabi@U.CASE.GEN@
```

The resulting **FST** also includes invalid paths like the following

```
bi@U.CASE.GEN@kitaabu@U.CASE.NOM@
bi@U.CASE.GEN@al@U.ART.PRESENT@kitaaba@U.CASE.ACC@
bi@U.CASE.GEN@kitaab@U.ART.ABSENT@uN@U.CASE.NOM@
```

that contain two **CASE** flags with incompatible values. As with the **ART** features shown above, these **CASE** incompatibilities or clashes are noticed by the **apply up** and **apply down** utilities in **xfst** (or by the **lookup** and **lookdown** utilities in **lexc**), and the illegal paths are blocked at runtime.

Note that even if the **CASE** features are compatible, the **ART** values can still clash and block an invalid path like the following:

```
bi@U.CASE.GEN@al@U.ART.PRESENT@kitaab@U.ART.ABSENT@iN@U.CASE.GEN@
```

When both *bi-* and *al-* are present, the only valid case ending is *-i*, as in the path

```
bi@U.CASE.GEN@al@U.ART.PRESENT@kitaabi@U.CASE.GEN@
```

corresponding to the word *biakitaabi*.

There is no practical limit on the number of Flag Diacritic features that can be used; the apply routines keep a running check on all feature values and block any path wherein any feature clash is detected.

Using Flag Diacritics and Upper-Side Tags

When Flag Diacritics are used together with typical multicharacter TAGS marking part-of-speech, person, number, etc., the semantics of Flag Diacritics remains the same, but the addition of Flag Diacritics to a **lexc** or regular-expression grammar becomes more complicated syntactically. Recall that multicharacter tags such as +Noun, +Verb, +Masc, +Fem, +Sg and +Pl (or [Noun], [Verb], etc.) are meant to be seen by the user, and they typically appear only on the upper side of a

transducer. Flag Diacritics, in contrast, are not intended to be seen by the user, and they typically must be visible on *both* the upper and lower sides of a transducer if, as usual, it is desired to have parallel restrictions in both analysis and generation.

For **lexc**'s **lookup** and **xfst**'s **apply up** to see Flag Diacritics, they must be visible on the lower side of the network, where the input is matched. Conversely, for **lexc**'s **lookdown** and **xfst**'s **apply down** to see Flag Diacritics, the flags must be visible on the upper side of the network, where the input is matched. In practice, Flag Diacritics should typically be visible on both sides of the network to get parallel constraints on analysis and generation.

Expert developers might get special desired behavior by putting some Flag Diacritics only on the lower side or only on the upper side, but for most practical applications, Flag Diacritics should appear on both sides. Failure to make the Flag Diacritics visible on both sides of a transducer is a cause of much bafflement among beginners.

The marking of the Arabic prefix *al-* with the Flag Diacritic @U.ART.PRESENT@ was performed above with the entry:

```
al@U.ART.PRESENT@      Stems ;
```

This will result in a transducer that has the same symbols visible on each side. In the actual **Xerox** implementation of network, a simple FSM encoding a language is not distinguished from the FST encoding the identity relation on that language. Thus a single-symbol label on an arc is effectively visible both to the lookup and generation applications.

```
Upper :      a  l  @U.ART.PRESENT@
Lower :      a  l  @U.ART.PRESENT@
```

Suppose, however, that we want the tag +Art to appear on the upper side; yet we still want the other symbols, including the Flag Diacritics, to appear on both the upper and lower sides. The goal is therefore to produce a two-level transducer that looks something like this:

```
Upper :      a  l  +Art @U.ART.PRESENT@
Lower :      a  l  0   @U.ART.PRESENT@
```

One way to accomplish this is by using the **lexc** colon notation as in

```
al+Art@U.ART.PRESENT@:al0@U.ART.PRESENT@      Stems ;
```

Unfortunately, the syntactic limitations of **lexc** require the rather awkward repetition of the *al* and the @U.ART.PRESENT@ on both sides of the colon. Another possibility is to use regular-expression notation, which inside **lexc** must appear inside angle brackets.

```
< a l %+Art:0 %@U%.ART%.PRESENT%@ > Nouns ;
```

Here, as desired, the **a**, **l** and Flag Diacritic @U.ART.PRESENT@ will be visible on both sides while the +Art tag will be visible only on the upper side. The developer must remember that in regular expressions, including those inside angle brackets in **lexc**, at-signs and periods are special characters and must be literalized with a preceding percent sign (%). Plus signs and other punctuation marks, used by convention in the spelling of tags, are also special inside angle brackets. The more legible alternative is to surround entire Flag Diacritics and tags in double quotes, which literalize everything they enclose.

```
< a l "+Art":0 "@U.ART.PRESENT@" > Nouns ;
```

The syntactic result is admittedly a bit awkward, and the failure to literalize the at-signs and periods of a Flag Diacritic in regular expressions is a common source of errors and frustration. Some developers prefer to write their grammars using orthographically simpler multicharacter symbols like ^ART and ^NOART, and then use substitution commands (Section 3.7.1) or rules to replace them with Flag Diacritic symbols.

Inside regular expressions, a Flag Diacritic like @U.feature.value@ must be spelled %@U%.feature%.value%@ or alternatively "@U.feature.value@" because at-signs and periods are special characters.

Respecting special characters and the need to put Flag Diacritics on both sides of the resulting **FST**, the entries for the Arabic case endings might look like the following:

```
LEXICON Case
< u "+Def":0 "+Nom":0 "@U.CASE.NOM@" > # ;
< a "+Def":0 "+Acc":0 "@U.CASE.ACC@" > # ;
< i "+Def":0 "+Gen":0 "@U.CASE.GEN@" > # ;
@U.ART.ABSENT@ IndefCase ;

LEXICON IndefCase
< uN "+Indef":0 "+Nom":0 "@U.CASE.NOM@" > # ;
< aN "+Indef":0 "+Acc":0 "@U.CASE.ACC@" > # ;
< iN "+Indef":0 "+Gen":0 "@U.CASE.GEN@" > # ;
```

Another approach is to avoid the regular-expression notation, creating new LEXICONS to hold the Flag Diacritics.

```
LEXICON Case
+Def+Nom:u      MarkNOM ;
+Def+Acc:a      MarkACC ;
+Def+Gen:i      MarkGEN ;
@U.ART.ABSENT@  IndefCase ;
```

```
LEXICON IndefCase
+Indef+Nom:uN   MarkNOM ;
+Indef+Acc:aN   MarkACC ;
+Indef+Gen:iN   MarkGEN ;
```

```
LEXICON MarkNOM
@U.CASE.NOM@    # ;
```

```
LEXICON MarkACC
@U.CASE.ACC@    # ;
```

```
LEXICON MarkGEN
@U.CASE.GEN@    # ;
```

In practice, the syntactic awkwardness of adding Flag Diacritics to most grammars often leads to mistakes where developers add Flag Diacritics only to the upper side of the **FST**. Remember that if Flag Diacritics are to be noticed during analysis, they must be present (visible) on the lower side of the **FST**, where input symbols are matched. Conversely, if Flag Diacritics are to be noticed during generation, they must be present (visible) on the upper side of the **FST**, where the input is matched.

For the default case, where it is desirable to have parallel restrictions in analysis and generation, Flag Diacritics must be visible on *both* sides of a transducer.

How Analysis and Generation Routines Interpret Flags

While it is intuitively easy to grasp that a Flag Diacritic like @U.ART.PRESENT@ is incompatible with another like @U.ART.ABSENT@, a full understanding of Flag Diacritics requires knowing how the analysis and generation routines interpret them. In **lexc**, these routines are called **lookup** and **lookdown**; in **xfst**, they are called **apply up** and **apply down**.

When analysis or generation of an input string begins, the small bit of feature memory in the application routine is initialized. All potential features begin with an unset or neutral value. Arbitrarily, we will take an example from analysis; generation, as usual, is just the inverse operation. The input to the analysis will be the ill-formed Arabic word **alkitaabuN*, which includes a definite-article prefix and an incompatible indefinite case suffix. The path in the transducer from the grammar in Figure 8.4 that threatens to accept this string looks like

```

a1+Art@U . ART . PRESENT@kitaab@U . ART . ABSENT@uN@U . CASE . NOM@
a10   @U . ART . PRESENT@kitaab@U . ART . ABSENT@uN@U . CASE . NOM@

```

After initialization, the analysis routine will successfully match the input symbols **a** and **l** against the lower-side of the path in the transducer. The epsilon arc, labeled here as **0**, is then traversed without consuming any input. Then the analysis routine sees, still on the lower side, the multicharacter symbol `@U . ART . PRESENT@` and knows, just from the distinctive spelling, that it is a Flag Diacritic. The analysis routine therefore treats it as an epsilon, not matching it against any input symbol. But at the same time, the analysis routine tries to perform the operation indicated by the Flag Diacritic; in this case, it tries to unify the value of the feature **ART**, which is currently neutral, with the value **PRESENT**. This unification succeeds, because unification with the neutral value always succeeds, and the analysis routine records in its feature memory that **ART = PRESENT**. This feature-value information is then carried along as the analysis progresses.

Because the indicated operation, in this case unification, succeeded, the analysis routine traverses the arc labeled `@U . ART . PRESENT@` and continues to match and consume input symbols **k**, **i**, **t**, **a**, **a**, **b** before finding yet another multicharacter symbol with the distinctive flag-diacritic spelling, `@U . ART . ABSENT@`, which indicates another unification operation. But when analysis tries to unify **ART = ABSENT** with the currently stored value **ART = PRESENT**, the unification fails and the analysis path is blocked. Analysis then backtracks, as appropriate, to see if any other analysis paths are possible—there are none in this case—and stops. If the backtracking proceeds past arcs where feature values were changed, the previous feature values are automatically restored as appropriate.

Thus although the network itself includes the illegal path, the way that the analysis routine interprets the Flag Diacritics at runtime prevents the illegal path from succeeding. Generation is constrained in exactly the same way, but using symbols and Flag Diacritics visible on the upper side of the transducer. We shall see below that other flag-diacritic operations besides unification are possible, but the left-to-right processing of Flag Diacritics during analysis or generation operates in the same fashion.

Remember that feature names and feature values are chosen by the programmer and have no inherent meaning to the analysis and generation routines. The restriction imposed above by the @U.ART.PRESENT@ and @U.ART.ABSENT@ Flag Diacritics could just as well have been done by marking the *al-* morpheme with @U.FOO.XXRM@ and the indefinite case suffix morphemes with @U.FOO.ELEPHANT@; the effect would be exactly the same. After finding *al-* and recording that **FOO = XXRM**, a subsequent attempt to unify with **FOO = ELEPHANT** would fail, blocking the illegal path.

8.3.2 The Full Range of Flag-Diacritic Operators

Although the U-type (Unification) flags are the most used and will be sufficient for some applications, the overall **XRCE** flag-diacritic scheme allows other potentially valuable flag operations for setting, unsetting, resetting and testing feature values. The general flag-diacritic templates are

```
@operator.feature.value@
```

and

```
@operator.feature@
```

Each operation type, with its spelling and semantic effect, is explained separately below. It is important to realize that when analysis or generation starts on a new input string, all potential feature values are initialized to the unset or neutral value.

P or Positive (Re)Setting

```
@P.feature.value@
```

When a @P.feature.value@ Flag Diacritic is encountered, the value of the indicated *feature* is simply set or reset to the indicated *value*. This (re)setting never causes failure (i.e. never causes the analysis or generation routine to fail and backtrack for other solutions). The P command is intended as a mnemonic for POSITIVE (RE)SETTING of a feature value.

N or Negative (Re)Setting

```
@N.feature.value@
```

When an @N.feature.value@ Flag Diacritic is encountered, the value of *feature* is set or reset to the negation or complement of *value*. This (re)setting never

causes failure or backtracking. In any subsequent unification checks, the negation of *value* is compatible with any value except *value* itself. The N is intended to suggest NEGATIVE (RE)SETTING of a feature value.

If analysis or generation encounters @N.FOO.BLAH@ it will remember that **FOO** ≠ **BLAH**. A subsequent attempt to unify @U.FOO.BLAH@ will fail, while an attempt to unify @U.FOO.ON@ or @U.FOO.OFF@ or any other value except **BLAH** will succeed.

R or Require Test

@R.feature.value@

When an @R.feature.value@ Flag Diacritic is encountered, a test is performed; this test succeeds if and only if *feature* is currently set to *value*. If the test fails, the path is blocked and the application routine backtracks for other possible solutions. The R stands for REQUIRE TEST. Note that for @R.feature.value@ to succeed, *feature* must currently be set to the precise *value* indicated. The sequence @N.FEAT.X@@R.FEAT.Y@ causes failure.

@R.feature@

When an @R.feature@ Flag Diacritic is encountered, the test succeeds if and only if *feature* is currently set to some value other than neutral. This non-neutral value can include a negative setting: e.g. if @N.MYFEAT.MYVAL@ has set the feature **MYFEAT** to the negation of **MYVAL**, a subsequent test @R.MYFEAT@ will succeed.

The combination of P-type and R-type flags is often useful for tying together the two halves of a circumfix. Circumfixes are coordinated pairs consisting of a prefix and a suffix (see Sections 8.5.4 and 9.3.3).

D or Disallow Test

@D.feature.value@

When a @D.feature.value@ Flag Diacritic is encountered, the test succeeds if and only if *feature* is currently neutral or is set to a value that is incompatible with *value*. Otherwise failure and backtracking result. The D stands for DISALLOW TEST. Note that the sequence @N.FEAT.M@@D.FEAT.Q@ fails; the sequence @N.FEAT.M@@D.FEAT.M@ succeeds.

@D.feature@

When a @D.feature@ Flag Diacritic is encountered, the test succeeds if and only if *feature* is currently neutral (unset).

A combination of the P-, R-, and D-type Flag Diacritics is often useful for controlling the idiosyncratic inflectional and derivational possibilities of individual roots and for constraining circumfixation. See Sections 8.5.2, 8.5.4 and 9.3.3.

C or Clear Feature

@C.feature@

When a @C.feature@ Flag Diacritic is encountered, the value of *feature* is reset to neutral. This resetting itself cannot cause failure or backtracking. The C stands for CLEAR.

U or Unification Test

@U.feature.value@

For completeness, we recap the semantics of the U operator here. If *feature* is currently neutral, then encountering @U.feature.value@ simply causes *feature* to be set to *value*. Else if *feature* is currently set (non-neutral), then the test will succeed if and only if *value* is compatible with the current value of *feature*. Note that the sequence @N.FEAT.M@U.FEAT.Q@ succeeds, leaving FEAT=Q in the feature memory. The U stands for UNIFICATION. The overall Flag-Diacritic system is intentionally non-monotonic because other operations can change and clear feature values.

8.4 Flag Diacritics and Finite-State Algorithms

8.4.1 Analysis and Generation

As already mentioned, the Flag Diacritics inside grammars and networks have the same status as any other multicharacter symbols. In order for Flag Diacritics to work, they must be used in conjunction with analysis and generation routines that are written to notice and obey them. In other words, Flag Diacritics must be used together with application routines that are specially written to be FLAG-AWARE, also known as FLAG-SENSITIVE.

xfst apply up and apply down

The **apply up** and **apply down** routines in **xfst** are flag-aware and obey Flag Diacritics by default, treating them as a special kind of epsilon symbol. Like a real epsilon, a Flag Diacritic is not matched by any symbol in the input, and a Flag Diacritic transition on the output side produces no output. Traversing an arc labeled with a Flag Diacritic is like an epsilon transition, but the transition is conditional on the success or failure of the specified flag operation. The result depends on the state of a feature register that is initialized at the beginning of the analysis or generation and is continuously updated along each path that is being explored.

For example, if the first label on a path is @U.Harmony.Back@, the **Harmony** feature is neutral at that point and the arc can be traversed, setting **Harmony** = **Back** in the process. After that, an arc labeled @U.Harmony.Front@ cannot subsequently be traversed unless there is an intervening @C.Harmony@ arc that resets the **Harmony** feature back to the initial neutral value, or an arc with a label such as @P.Harmony.Front@ or @N.Harmony.Back@ that resets **Harmony** to a value compatible with **Front**.

The normal processing of Flag Diacritics in **xfst** occurs when the interface variable **obey-flags** is **ON**, which is the default. When **obey-flags** is reset by the user to **OFF**, Flag Diacritics are treated like any other multicharacter symbols. In **xfst**, use the command **show obey-flags** to see the current value of the variable **obey-flags**.

```
xfst[]: show obey-flags
```

Use **set obey-flags OFF** or **set obey-flags ON** to change the value. The default **ON** value is appropriate for most users.

```
xfst[]: set obey-flags OFF
xfst[]: set obey-flags ON
```

lexc lookup and lookdown

The user-settable variables that affect the operation of **lexc** are called SWITCHES.

When the **lexc** switch **obey-flags** is set to **ON** (the default value), Flag Diacritics are processed normally, as a special kind of epsilon symbol. The **lexc** command **status** prints out the current setting of **obey-flags** and all the other **lexc** switches.

```
lexc> status
```

Simply entering the **lexc** command **obey-flags** will toggle the switch to the opposite value.

```
lexc> obey-flags
```

8.4.2 Seeing Flag Diacritics in Networks

Checking the Alphabet

Flag Diacritics are normally intended to have a semantic status, and at application time they are, by default, not seen in either the input or the output. During development and debugging, however, it is often important to see what your Flag Diacritics are and where they appear in the strings of your network.

The first technique for checking Flag Diacritics is “checking the alphabet”, i.e. loading the network onto an **xfst** stack and entering **print sigma** or **print labels**. The output identifies Flag Diacritics and their semantic effect. See Section 7.3.

Using the **xfst print** Commands to see Flags

The second general technique for seeing where your Flag Diacritics are is to use the various **print** and **print random** commands of **xfst**:

```
print words
print upper-words
print lower-words
print random-words
print random-upper
print random-lower
```

What these commands display is influenced by the setting of three **xfst** interface variables, discussed below:

```
obey-flags    (ON/OFF, default ON)
show-flags   (ON/OFF, default OFF)
print-space  (ON/OFF, default OFF)
```

To see terse documentation about all the **xfst** interface variables, enter

```
xfst[0]: help variable
```

To see terse documentation about a particular variable, e.g. **obey-flags**, enter

```
xfst[0]: help obey-flags
```

To see the current setting of any variable, e.g. **obey-flags**, enter

```
xfst[0]: show obey-flags
```

The obey-flags Variable The **obey-flags** variable affects the application routines **apply up** and **apply down**, and they affect the output of the various **print** commands.

With the default setting **obey-flags=ON**, the application routines will “obey” Flag Diacritics and not analyze or generate any string that would involve a violation of flag-diacritic restrictions. In this mode, Flag Diacritics are not shown in the output strings.

If you manually set **obey-flags** to **OFF**, then the normal restrictions of the Flag Diacritics in the network will be ignored, and they will be output as part of the output strings. The input string, however, does not have to include the Flag Diacritics for application to be successful. See the example below in Section 8.5.5.

The **print** commands will obey Flag Diacritics, and not print out any flag-invalid examples, when **obey-flags** is **ON**.

The show-flags Variable If you want the various **print** commands to obey Flag Diacritics and yet to display them as well, set the variable **show-flags** to **ON**.

```
xfst[0]: set show-flags ON
```

The print-space Variable The various **print** commands will normally print out orthographical examples of strings that the network will accept, but they do not, by default, show you how those strings are tokenized into symbols. To cause the **print** commands to print spaces between symbols, and thus to show you exactly where the multicharacter symbols are, set the **print-space** variable to **ON**.

```
xfst[0]: set print-space ON
```

Flag Diacritics, Variables, and Debugging

For practical inspection and debugging of networks, especially when dealing with Flag Diacritics and the **compile-replace** algorithm presented in Chapter 9, developers are advised to set both **show-flags** and **print-space** to **ON**.

```
xfst[0]: set print-space ON
xfst[0]: set show-flags ON
```

Subsequent use of **print** commands such as **print random-lower** will then always print spaces between symbols, showing you where the multicharacter symbols are, and will always display any Flag Diacritics that are present, whether or not they are being obeyed.

These commands can be put into a trivial **xfst** script file, perhaps called *variable-settings.script*, which could be run when you invoke **xfst**:

```
unix> xfst -l variable-settings.script
```

If you want **xfst** always to start up with customized variable settings, then the following alias could be defined in your `.cshrc` (or equivalent) file.⁴

```
alias xfst 'xfst -l variable-settings.script'
```

Another way to get the same result is to define the following alias:

```
alias xfst 'xfst -e "set show-flags ON" -e "set print-space ON"'
```

Such an alias might also be given a different name, e.g. *xfstdb* for “xfst debug”.

8.4.3 Eliminating Flag Diacritics

From a formal point of view, any restriction that you can enforce using Flag Diacritics could also be done by writing suitable filters and composing them with the network at compile time. For example, the fact that the *al* prefix in Arabic is incompatible with the indefinite case endings *uN*, *aN*, and *iN* could easily be expressed on the lexical level as a co-occurrence restriction on the corresponding tags, which we will assume here are `+Art` and `+Indef`. Using **xfst**, the appropriate filter network can be defined as

```
xfst[]: define Filter ~$["+Art" ?* "+Indef"] ;
```

which denotes the language of all strings that do not contain a `+Art` tag followed at any distance by an `+Indef` tag. The composition of this filter on top of the original, unconstrained source lexicon,

```
xfst[]: define FilteredFST Filter .o. Lexicon ;
```

yields a constrained network result in which the *al* prefix, tagged as `+Art`, is never followed by any of the three case endings marked with the `+Indef` tag. In every other respect the result is identical to the original lexicon.

The disadvantage of “composing in” such restrictions is that the resulting transducer can often explode in size, as shown in Section 6.3.3; the advantage of Flag Diacritics is that they do not cause a size explosion and impose the same constraints at runtime using analysis and generation routines that have been specially written to recognize and “obey” the Flag Diacritics. The benefit in size, however, comes at a cost of slightly slower application speed because the runtime routine must spend time in exploring paths that ultimately fail because of a flag constraint.

It is often useful to experiment with the size/speed tradeoff between using and not using flags to enforce particular constraints. This is in fact very easy to do in **xfst**. The command **eliminate flag** removes from the network all flag diacritic symbols for a given feature and, at the same time, composes in the restrictions that they

⁴In a Unix-like system, the file named `.cshrc`, or something similar to that, is automatically executed whenever a new window is opened. See your system guru for local details.

encode. For example, if the Arabic lexicographer has used @U.ART.PRESENT@ and @U.ART.ABSENT@ to constrain the incompatibility of the definite article and the indefinite case endings, he or she can load the lexicon (let us call it `arabic.fst`) onto the top of the **xfst** stack and eliminate the Flag Diacritics for **ART** with the following commands:

```
xfst[n]: clear stack
xfst[0]: load arabic.fst
xfst[1]: eliminate flag ART
```

The effect of the **eliminate flag** command is that **xfst** first composes the lexicon with filters that interpret the flag diacritic operators U, P, C, etc. for the given feature, ART, and then removes from the network all diacritic symbols that contain **ART** as the feature. The resulting network left on the stack is equivalent to the original in that it does not contain any illegal sequences that were blocked by the removed diacritics. It is typically larger but faster to apply at runtime.

To take a concrete example, let us consider the mini lexicon for Arabic in Section 8.4 (page 449). Let us assume that the lexicon has been compiled with **lexc** and that we have saved the resulting network in file `arabic.fst`. The size of the lexicon at this point is 18 states, 25 arcs. It contains 24 paths, of which only 12 are valid, obeying flag constraints.

This lexicon contains two types of Flag Diacritics. The **ART** diacritics block the co-occurrence of the definite article *al* with indefinite case endings; the **CASE** flags constrain the *bi* prefix to co-occur only with genitive case endings. In the following, we eliminate the two types of flags one by one, watching the effect on the size of the network. The order of the elimination does not matter.

```
xfst[n]: clear stack
xfst[0]: load arabic.fst
xfst[1]: print size
716 bytes. 18 states, 25 arcs, 24 paths.
xfst[1]: eliminate flag ART
852 bytes. 22 states, 31 arcs, 18 paths.
xfst[1]: eliminate flag CASE
1.1 Kb. 32 states, 42 arcs, 12 paths.
```

As the above trace shows, the effect of eliminating the flags is twofold. The number of states and arcs goes up while the number of paths decreases. At the end we are left with a network that is considerably larger than the original, but it contains no illegal dead-end paths that could slow down the runtime routines somewhat.

Developers should not be too worried about the performance penalty of Flag Diacritics. First, Flag Diacritics were implemented very efficiently, compared to earlier schemes based on general feature unification. Second, you can always remove Flag Diacritics trivially by using the **eliminate flag** command.

8.4.4 Flag Diacritics and Composition

Flag Diacritics create complications for composition.

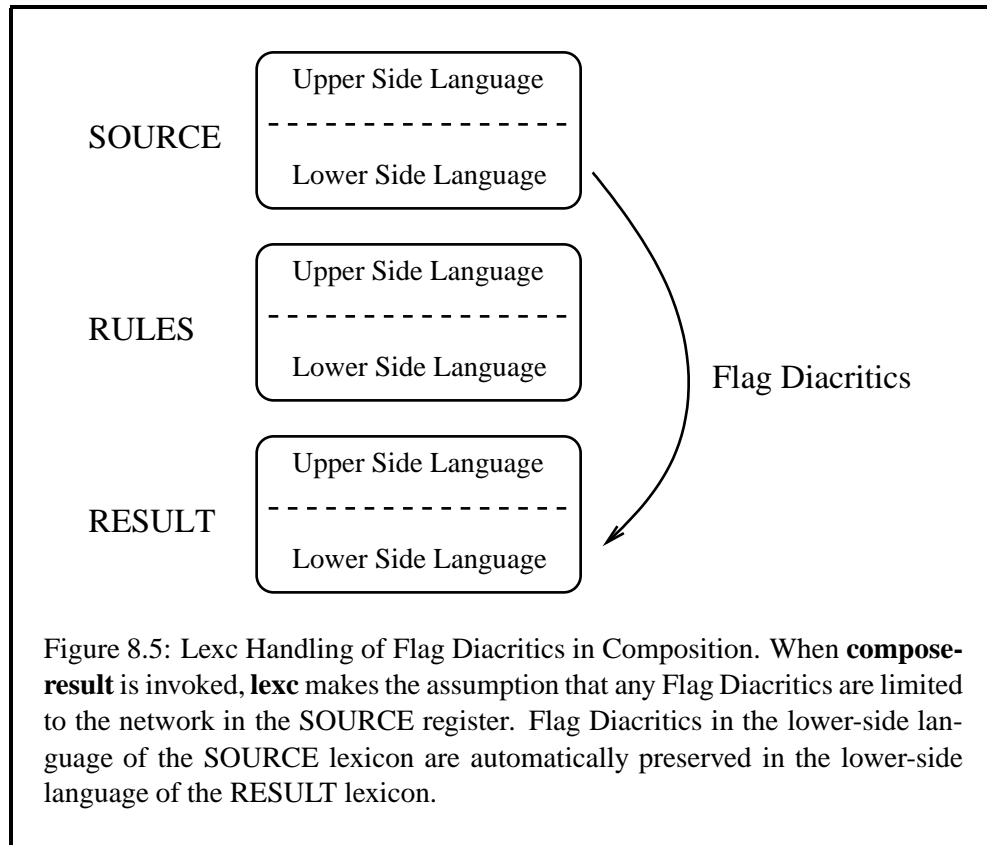
In most projects, Flag Diacritics are initially included in a network by writing them into the **lexc** or **xfst** grammar that defines the lexicon; and at this point, the system is typically only partially finished. Developers may subsequently compose rules and filters on both the top and bottom of the original lexicon network to create the final transducer. The challenge in such cases is to preserve the Flag Diacritics, which are typically intended to be visible on both the top and bottom of the final transducer. In examining how composition handles Flag Diacritics, there are two composition algorithms in the **Xerox** calculus to be considered: the **compose-result** algorithm in **lexc**, and **compose net** in **xfst**. For historical and practical reasons, they have different default behavior.

lexc Composition

In **lexc**, the composition algorithm invoked by the **compose-result** command assumes that the rules, loaded into the RULES register (see Section 4.4), are being composed on the lower side of the lexicon (stored in the SOURCE register), that any Flag Diacritics are in the lexicon, and that the rules do not contain or refer to Flag Diacritics in any way. Under these assumptions, it is safe to compose the rules on the lower side of the lexicon and preserve the lower-side Flag Diacritics from the lexicon in the result, and this is what **compose-result** does. After invoking **compose-result** in **lexc**, any Flag Diacritics on the lower side of the original SOURCE lexicon FST automatically re-appear on the bottom side of the RESULT FST as shown in Figure 8.5.

xfst Composition

In contrast, the composition algorithm invoked by the **xfst compose net** command, or by the `.o.` operator in regular expressions, treats Flag Diacritics by default as ordinary multicharacter symbols. This default setting is suitable for cases where the rules being composed actually change, delete or insert Flag Diacritics or where the presence of Flag Diacritics should cause a rule context to match (or not match).



To make the **xfst compose net** function work like composition in **lexc**, preserving Flag Diacritics in the result, the user-settable switch **flag-is-epsilon** must be reset to **ON**; it is **OFF** by default.

```
xfst[1]: show flag-is-epsilon
variable flag-is-epsilon = OFF
xfst[1]:
```

To reset **flag-is-epsilon**, enter **set flag-is-epsilon OFF** or **set flag-is-epsilon ON** as appropriate. **xfst** will echo the new setting.

```
xfst[]: set flag-is-epsilon ON
variable flag-is-epsilon = ON
```

In **xfst** composition, Flag Diacritics are treated by default as normal multicharacter symbols. To have Flag Diacritics treated specially during composition, as in **lexc**, just set the **xfst** interface variable **flag-is-epsilon** to **ON**.

There is a formal reason why **flag-is-epsilon** is **OFF** by default. The composition of two **FSTs**, both containing Flag Diacritics treated as epsilons, can result in mathematical confusion and unpredictability. **lexc** avoids the problem only by assuming that Flag Diacritics (if any) are present only in the **SOURCE** lexicon; given the traditional way that **lexc** was used, this is a reasonable assumption. But such an assumption is not generally safe in **xfst**. In **xfst** with **flag-is-epsilon** manually reset to **ON**, the composition algorithm first checks the label alphabets of both machines and generates a warning message if both contain Flag Diacritics.

8.4.5 Flag Diacritics and compile-replace

Like composition, the **compile-replace** algorithm presented in Chapter 9 modifies one side of a network, and there may of course be Flag Diacritics on the modified side. The **compile-replace** algorithm preserves the Flag Diacritics in the modified network as explained in Section 9.5.2.

8.4.6 Reflecting Flag Diacritics Side-to-Side

In some projects, it may be convenient to place Flag Diacritics on only one side of a network, e.g. on the upper side, then perform various options like composition and **compile-replace** on the other (e.g. lower) side, and then cause all the Flag Diacritics on either side to be reflected or echoed onto the other side. The result is a network wherein all the Flag Diacritics are visible on both sides. This somewhat controversial and potentially dangerous operation is available under the name **twosided flag-diacritics**; it takes as its argument the network on top of the **xfst** stack, makes all the Flag Diacritics visible on both sides, and then pushes the result back onto the stack.

```
xfst[0]: twosided flag-diacritics
```

Where the original network contains an arc labeled with a one-side flag-diacritic like **@U . X . Y@ : a**, this algorithm will create a sequence of two arcs labeled **@U . X . Y@** and **0:a**. Similarly, an arc labeled **b:@U.M.N@** will become the sequence **b:0 @U.M.N@**. Any arc labeled **0:@U.Q.R@** or **@U.Q.R@:0** will be simplified to **@U . Q . R@**.⁵

⁵As Flag Diacritics are treated like epsilons during application, and as two-sided epsilons present mathematical dilemmas for operations like composition, any arc with a Flag Diacritic on each side, like **@U.X.Y@:@R.M.N@**, is already suspect; the regular-expression compilers generate a warning message when such labels are produced. The **twosided flag-diacritics** algorithm cannot know whether to change **@U.X.Y@:@R.M.N@** into the sequence **@U . X . Y@ @R . M . N@** or into the oppositely ordered sequence **@R . M . N@ @U . X . Y@**. The ordering could be significant, and the suitability of the output cannot be guaranteed.

8.5 Examples

8.5.1 French Articles

The Arabic example in Figure 8.4 illustrates a classic application where Flag Diacritics are used to constrain long-distance dependencies within words. Another example along the same lines comes from French, where the definite article *le* (masculine) or *la* (feminine) is elided onto the front of a noun or adjective, but

1. The noun or adjective must begin with a vowel phoneme as in *l'arbre* (“the tree”). In the orthography, some words begin with a “silent h” and are subject to the same elision, e.g. *l'homme* (“the man”) and *l'heure* (“the hour”).
2. The noun or adjective must be singular, e.g. *l'arbre* is good but the plural **l'arbres* is definitely bad.

The most straightforward way to build a **lexc** lexicon for French involves defining the elided article *l'* initially as a kind of prefix that continues to attach to all nouns and adjectives, e.g.

```
LEXICON Root
l'      NounOrAdj ;
        NounOrAdj ;

LEXICON NounOrAdj
        Nouns ;
        Adjectives ;
```

The restricting of *l'* to nouns and adjectives beginning with a vowel phoneme can be done by the usual composition of a filter without much of a size penalty, but the restriction that prevents *l'* from co-occurring with the plural *+s* suffix causes a large section of the noun and adjective networks to be copied if done via the traditional composition of a filter. Flag Diacritics are ideal for such a task, e.g.

```
LEXICON Root
< l %' "@U.NUM.SG@" > NounOrAdj ;
                        NounOrAdj ;

LEXICON Num
+Sg:0                    # ;
< "+Pl":s "@U.NUM.PL@" > # ;
```

That is, the *l'* morpheme has associated with it the Flag Diacritic @U.NUM.SG@ and the incompatible plural suffix contains, as part of its spelling, the incompatible Flag Diacritic @U.NUM.PL@. With Flag Diacritics defined in this way, the analysis routine will reject **l'arbres* without an explosion in the size of the network.

8.5.2 Using the R-type Flag Diacritics

Although Flag Diacritics were originally conceived with the U-type functionality centrally in mind (to disallow words that contained incompatible morphemes, perhaps widely separated), the usefulness of some of the other flag-diacritic operations is easily demonstrated. The R-type Flag Diacritics, for example, are useful for limiting certain suffixes to occur only with specially-marked semantic subclasses of stems.

As will be recalled from the Esperanto exercise in Section 4.2.8, there are certain stems like *kat-* (“cat”), *hund-* (“dog”) and *elefant-* (“elephant”) that denote animals that can be masculine or feminine. Esperanto has an *-in* suffix that marks feminine (as opposed to the unmarked or masculine form) that can attach to such stems, but it cannot attach to other common-noun stems like *libr-* (“book”) and *dom-* (“house”). Such a restriction could be handled with continuation classes as in the following **lexc** fragment:

```
LEXICON Nouns
kat      Nmf ; ! Nmf for nouns that can take -in
hund     Nmf ;
elefant  Nmf ;
libr     N   ; ! N for nouns that cannot take -in
dom      N   ;
```

```
LEXICON Nmf
in       Nmf ;
eg       Nmf ;
et       Nmf ;
        NEnd ;
```

```
LEXICON N
eg       N ;
et       N ;
        NEnd ;
```

but this typically requires that the lexicographer maintain multiple copies of some suffixes, here *-eg* and *-et*, which can be problematic. (In full-sized systems, trying to handle idiosyncratic semantic and derivational restrictions with continuation classes alone can result in a proliferation of little LEXICONS.) The same effect can be achieved by marking all and only the subset of stems that allow the feminine *-in* suffix with a distinctive feature, e.g. @U.MF.ON@ or @P.MF.ON@, and then marking the *-in* suffix with @R.MF.ON@ meaning “succeed if and only if the MF feature has been set to ON”. The lexicons would look something like this:

```
LEXICON Nouns
kat@P.MF.ON@      N ; ! compatible with -in suffix
```

```

hund@P.MF.ON@      N ;
elefant@P.MF.ON@  N ;

libr                N ; ! not compatible with -in
dom                 N ;

LEXICON N
@R.MF.ON@in        N ; ! -in is allowed iff MF = ON
eg                 N ;
et                 N ;
                  NEnd ;

```

Once such Flag Diacritics are in place, the suffix *-in* will be allowed only on roots marked @P.MF.ON@; any attempt to attach *-in* to a noun stem like *libr-* will be blocked at runtime. The same effect could be achieved in different way by marking all the roots like *libr-* and *dom-* with @P.MF.OFF@, and marking *-in* with @U.MF.ON, but this would involve more work and be less intuitive.

8.5.3 Using P-type Flag Diacritics for Positive (Re)Set

The P-type Flag Diacritics simply set or reset a feature to a specified value, never causing a failure. This can be extremely useful in a language like Finnish or Mongolian, which have vowel harmony. Typically an overall word will have only front harmony or only back harmony, and incompatibilities between various morphemes can be handled using only U-type Flag Diacritics. But in compound words, the possibility exists that a front-harmony word will compound onto a back-harmony word (or vice versa) so that the harmony feature has to change in mid-word.

Take the Finnish example *itäänmuuttoko* (“eastward migration?”) where *itä* is a front-vowel root meaning “east”, *Vn* (realized here as *än*) a suffix meaning “towards”, *muutto* a back-vowel root meaning “migration”, and *kO* (realized here as *ko*) is the question marker. Assuming that *itä* and other front-vowel roots are marked @P.VOWEL.FRONT@, the analysis routine will remember **VOWEL = FRONT** and make sure that only compatibly marked front-vowel versions of suffixes, like *än*, attach to it. One grammar of Finnish (Blåberg, 1994) handles all vowel harmony by simply listing the allomorphs and marking each with suitable harmony features.

```

LEXICON Ward
än@U.VOWEL.FRONT@  Next ;    ! front-vowel variant
an@U.VOWEL.BACK@   Next ;    ! back-vowel variant

```

But at the compound boundary, the vowel harmony changes abruptly (and quite legally) to **BACK** with the *muutto* root. In such a case, the positive setting indicated by the flag @P.VOWEL.BACK@ is used to reset the value of **VOWEL** without

causing a failure. Then subsequent unifications will select the right allomorph of *kO*, which in this case is *ko*.

```
LEXICON Question
kö@U.VOWEL.FRONT@      # ;      ! front-vowel variant
ko@U.VOWEL.BACK@       # ;      ! back-vowel variant
```

Of course, it is not necessary to use Flag Diacritics for phenomena such as vowel harmony. The traditional rule-based solution, avoiding Flag Diacritics and the listing of all the allomorphs, is to define lexicons like

```
LEXICON Ward
^An      Next ;
```

```
LEXICON Question
k^O      # ;
```

where multicharacter symbol [^]A represents the open vowel and [^]O represents the mid vowel, both underspecified for harmony. The grammar will then generate lexical words like *it^äAnmuuttok[^]O* containing underspecified vowels. The following **xfst** Replace Rule can then be applied, via composition, to enforce the vowel harmony in the surface word.

```
xfst[]: define FrontVowel [i|e|ä] ;
xfst[]: define VowelHarmonyRule %^O -> ö,
%^A -> ä // FrontVowel Con+ _
.o.
%^A -> a
.o.
%^O -> o ;
```

If the VowelHarmonyRule is applied in a downward direction to *it^äAnmuuttok[^]O*, the result is the desired *it^äänmuuttoko*. Note that VowelHarmonyRule is defined with the rarely used // operator, which causes the left context to be matched on the lower side of the relation. This behavior is especially appropriate for vowel harmony in languages where the harmony can change within the word, and the realization of each underspecified vowel is controlled by the surface realization of the immediately preceding vowel.

8.5.4 Using Flag Diacritics to Constrain Circumfixes

Circumfixes are coordinated pairs consisting of a prefix and a suffix. In a typical case, the prefix part will be only one of many possible prefixes, and all will concatenate to the set of stems. The stems in turn continue on to sets of suffixes, including the suffix part of a circumfix. The challenge, then, is to recognize the

suffix part of a circumfix if and only if the prefix part of the same circumfix was recognized earlier in the word. This is a classic job for Flag Diacritics.

Consider an artificial example where *ik* and *lu* are simple prefixes, never co-occurring with any suffix, *ka* and *sin* are simple suffixes, never co-occurring with any prefix, and where *lan-sib* and *zan-fi* are circumfixes. The following grammar will constrain the prefixes and suffixes appropriately.

Multichar_Symbols

LEXICON Root

Prefixes ;

LEXICON Prefixes

```
ik@P.PREF.simple@  Stems ;
lu@P.PREF.simple@  Stems ;
lan@P.PREF.lan@    Stems ;
zan@P.PREF.zan@    Stems ;
                    Stems ; ! empty entry
```

LEXICON Stems

```
ranik  Suffixes ;
fin    Suffixes ;
zolar  Suffixes ;
```

LEXICON Suffixes

```
ka@D.PREF@      # ; ! disallow any prefixes
ik@D.PREF@      # ;

sib@R.PREF.lan@ # ; ! suffix sib requires
                  ! the lan prefix
fi@R.PREF.zan@  # ; ! suffix fi requires
                  ! the zan prefix
@R.PREF.simple@ # ; ! for simple prefixes with
                  ! no phonological suffix
@D.PREF@        # ; ! for bare stems
```

Note in this example how the D- and R-Type flags on the suffixes controlling the blocking or requiring of particular prefixes. A more complicated (and more real) example involving circumfixes in Malay is shown in Section 9.3.3.

8.5.5 Finnish Proper Names and Derivatives

The Problem

Flag Diacritics can be used to handle another challenge in Finnish, where proper names must appear with an initial uppercase letter, but their derivatives are normally in lowercase, although they will of course be capitalized at the beginning of a sentence like all ordinary words. For example, the Finnish word for “Paris” is *Pariisi*, with an obligatory capital P at the beginning; but the corresponding adjective “Parisian” translates normally into *pariisilainen*, or into *Pariisilainen* if it happens to be the first word in a sentence.

The derivatives of multiword names, e.g. *Palo Alto*, lose the internal capital and any internal spaces: *paloaltolainen* or *Paloaltolainen* but not, for example, **Palo altolainen* or **PaloAltolainen*. There are some additional complexities, but we will ignore them for the purposes of the following example.

A Solution

One solution is to prefix Finnish proper names, in the lexicon, with a pair of alternate Flag Diacritics, @U.Cap.Obl@ and @U.Cap.Opt@, to signify “initial capital letter is obligatory” and “initial capital letter is optional”, respectively. Derivative noun-to-adjective and noun-to-noun suffixes, such as *-lainen* above, are compatible with either flag. The proper noun tag +PN, on the other hand, is compatible only with @U.Cap.Obl@. Here is the lexicon, written in **lexc**.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! demo-lex.txt
!
! Simple Finnish Lexicon with Flag Diacritics
!
! Includes words like "Pariisi" (Paris) "pariisilainen"
! (Parisian), "Palo Alto", "paloaltolainen" (Palo Altan).
! The initial capital is obligatory in "Pariisi",
! optional in "pariisilainen". The internal space in
! "Palo Alto" is not present in "paloaltolainen".
```

Multichar_Symbols

```
@U.Cap.Obl@ @U.Cap.Opt@
+PN +Adj +Der+
```

LEXICON Root

```
@U.Cap.Obl@ PropNoun ;
@U.Cap.Opt@ PropNoun ;
```

LEXICON PropNoun

```
Pariisi PNSuff ;
Grenoble PNSuff ;
```

```

Palo_Alto          PNSuff ;

! N.B. that _ denotes the literal space
! character in this grammar

LEXICON PNSuff
  @U.Cap.Obl@      PN ;
  @U.Cap.Opt@      AdjSuff ;

LEXICON PN
  +PN:0            # ;

LEXICON AdjSuff
  +Der+:0          LAINEN ;

LEXICON LAINEN
  lainen           ADJ ;

LEXICON ADJ
  +Adj:0           # ;

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

```

If you compile this lexicon (using **compile-source**) and call **lookup** with the **obey-flags** switch set to **ON** (the default value), you will not see any flags in the output.

```

lexc> lookup Palo_Alto
NOTE: Using SOURCE.
Palo_Alto+PN

```

If you toggle the switch **OFF**, by simply entering **obey-flags**, you see that there are actually two matching paths in the transducer:

```

lexc> obey-flags
Obey-flags is OFF.

lexc> lookup Palo_Alto
NOTE: Using SOURCE.
@U.Cap.Obl@Palo_Alto@U.Cap.Obl@+PN
@U.Cap.Opt@Palo_Alto@U.Cap.Obl@+PN

```

With **obey-flags** set to **ON**, the second path is blocked at application time because the two values for the **Cap** feature are in conflict. Note also that when **obey-flags** is **OFF**, Flag Diacritics are still treated as epsilons for analysis purposes, but the Flag Diacritics are displayed in the output.

Let's assume that the **lexc** lexicon is compiled and that the resulting binary network is saved in file `lex.fst`. The downcasing of the initial P in the derived form

pariisilainen is then handled by an optional rule triggered by the @U.Cap.Opt@ flag. The two other rules force the deletion of any word-internal spaces and capitals following the @U.Cap.Opt@ flag, even if the initial capital remains. Here is the associated **xfst** script file containing the rules.

```

clear stack

define UC      A | B | C | D | E | F | G | H |
               I | J | K | L | M | N | O | P | Q | R | S |
               T | U | V | W | X | Y | Z ;

read regex [

# Allow optional initial downcasing after @U.Cap.Opt@

A (->) a, B (->) b, C (->) c, D (->) d, E (->) e,
F (->) f, G (->) g, H (->) h, I (->) i, J (->) j,
K (->) k, L (->) l, M (->) m, N (->) n, O (->) o,
P (->) p, Q (->) q, R (->) r, S (->) s, T (->) t,
U (->) u, V (->) v, W (->) w, X (->) x, Y (->) y,
Z (->) z

|| .#. %@U%.Cap%.Opt%@ _

.o.

# No uppercase in the middle of a downcasable word

A->a, B->b, C->c, D->d, E->e, F->f, G->g, H->h,
I->i, J->j, K->k, L->l, M->m, N->n, O->o, P->p,
Q->q, R->r, S->s, T->t, U->u, V->v, W->w, X->x,
Y->y, Z->z

|| %@U%.Cap%.Opt%@ ?+ _

.o.

# Eliminate internal spaces inside a downcasable word
# Spaces are indicated here with the literal
# underscore character

%_ -> [] || .#. %@U%.Cap%.Opt%@ ?+ _

] ;

echo >>>>This leaves the rule transducer on the stack
print stack

echo >>>>Loading lex.fst onto the stack
load stack lex.fst

```

```
echo >>>>There should now be two networks on the stack
print stack
```

```
echo >>>>Composing the rules under the lexicon
compose net
```

```
echo >>>>Composition complete
```

If we save the result to a file and look at it in **xfst**, we see that flags keep the size of the network smaller:

```
xfst[1]: size
34 states, 41 arcs, 18 paths.
```

The **eliminate flag** command gives us an equivalent network without any diacritic symbols for the specified feature:

```
xfst[1]: eliminate flag Cap
46 states, 51 arcs, 9 paths.
```

In this case, compiling the constraint (via **eliminate flag**) into the structure of the network increases the size of the net by about 35%. Note that the number of words drops from 18 to 9 as the paths that violate **Cap** constraints are eliminated.

8.5.6 Forward-Looking Feature Requirements

It is sometimes the case that the presence of one morpheme, let us call it *foo*, requires the presence of another morpheme, let us call it *fum*, somewhere later in the same word. The morpheme *foo* then has a forward-looking requirement of *fum*, and any word containing *foo* without a following *fum* is illegal and should be blocked. While such forward-looking requirements are not directly supported by the current implementation of Flag Diacritics, a simple idiom can achieve the desired restriction.

To capture such forward-looking feature requirements, think of the morpheme *foo* as setting or “raising” a feature requirement that must subsequently be satisfied or cleared by a following morpheme *fum*. In a **lexc** grammar, the entry for *foo* might be as follows:

```
LEXICON A
foo@P.RequireFum.ON@ CC1 ;
```

Feature names like **RequireFum** are of course meaningless to the system, but they help the developer to remember the intent. When the system matches *foo*, it will also remember the setting **RequireFum = ON**. Of course, only those morphemes requiring a subsequent *fum* should be marked in this way.

The *fum* morpheme can then be encoded thus:

```
LEXICON Q
fum@C.RequireFum@      CC2 ;
```

with a Clear Flag Diacritic that simply clears or satisfies the requirements raised by the presence of a preceding *foo*. Of course, only *fum* should clear this feature. The only remaining step is to ensure the failure of any analysis or generation with any unsatisfied requirements, and this requires a final feature check at the end of each word. In **lexc**, this is achieved by defining a final LEXICON like the following and making sure that it is the only LEXICON that directly refers to # for end-of-word.

```
LEXICON End
@D.RequireFum@        # ;
```

In such a grammar, all words must be funneled through LEXICON End as a final step, and the **D**-type Flag Diacritic will perform a final feature-check, disallowing (blocking) any word where **RequireFum** is currently set to **ON** or to any value other than neutral. In a grammar where multiple forward-looking requirements must be constrained, e.g. **RequireFum**, **RequireFie** and **RequireBar**, then the final LEXICON through which all words are funneled must perform a final feature-check for all of them, e.g.

```
LEXICON End
@D.RequireFum@@D.RequireFie@@D.RequireBar@    # ;
```


Chapter 9

Non-Concatenative Morphotactics

Contents

9.1	Introduction	478
9.1.1	The Challenge of Non-Concatenative Morphotactics	478
9.1.2	Morphotactic Processes	479
	Concatenation vs. Non-Concatenative Processes	479
	Fixed-Length Reduplication in Tagalog	479
	Full-Stem Reduplication in Malay	479
	Stem Interdigitation in Arabic	480
9.2	Formal Morphotactic Description	481
9.2.1	Concatenative Assumptions and Limitations	481
9.2.2	Overcoming the Concatenative Limitation	482
	A New Mindset	482
	The Mechanics of compile-replace	486
9.3	Practical Non-Concatenative Examples	487
9.3.1	Restricted Reduplication in Tagalog	487
9.3.2	Full-Stem Reduplication in Malay	489
9.3.3	More Malay Reduplication	491
9.3.4	Semitic Stem Interdigitation	501
	The Traditional Conception of Interdigitation	501
	Three Formalizations of Semitic Stems	502
	Stem Interdigitation as Intersection	503
	Stem Interdigitation using compile-replace	505
	Three-Way Solution using Merge	505
	Two-Way Solution using Merge	510
	Three-Way Solution using General-Purpose Intersection	513

	Returning Stems Instead of Roots	516
9.4	Usage Notes for <code>compile-replace</code>	518
9.4.1	The Non-Concatenative Challenge	518
9.4.2	Morphotactic Abstraction	519
9.4.3	Upper or Lower Application	519
9.4.4	Multiple Application	519
9.4.5	The No-Retokenize Option	519
	Two Modes of Operation	519
	An Example with <code>retokenize=OFF</code>	520
9.5	Debugging Tips for <code>compile-replace</code>	523
9.5.1	Properly Defining the Initial Network	523
	Default Retokenize Mode	524
	Alternative No-Retokenize Mode	524
9.5.2	Flag Diacritics	524
	Flag Diacritics and <code>compile-replace</code>	524
9.5.3	Size Problems	525
9.6	The Formal Power of Morphotactics	525
9.6.1	The Formal Power of Full-Stem Reduplication	525
9.6.2	Analyzing vs. Guessing Reduplications	526
9.7	Conclusion	526

9.1 Introduction

9.1.1 The Challenge of Non-Concatenative Morphotactics

Non-concatenative morphotactics is a difficult topic, deserving a book all to itself. This chapter introduces the **compile-replace** algorithm, included in **xfst**, which the authors have used successfully to handle some traditionally difficult non-concatenative phenomena—namely full-stem reduplication and Semitic stem interdigitation—without going beyond finite-state power (Beesley and Karttunen, 2000). There is much experimental and practical work that remains to be done.

It should be noted that non-concatenative morphotactics is a challenge no matter how you go about it, and that this is a subject best reserved for experts who are already fully comfortable with the rest of the **Xerox** finite-state calculus.

Non-concatenative morphotactics and the **compile-replace** algorithm are best approached by experienced finite-state developers.

9.1.2 Morphotactic Processes

Concatenation vs. Non-Concatenative Processes

As we have seen, most natural languages construct words by concatenating morphemes together, and concatenation is of course a finite-state operation. Such concatenative morphotactics can be impressively productive, especially in agglutinative and polysynthetic languages. In Inuktitut, as shown in Table 9.1, a single word may contain as many morphemes as an average-length English sentence (example from (Mallon, 1999)).

Upper	Paris+mut+nngau+juma+niraq+lauq+sima+nngit+junga
Lower	Parimunnngaujumaniralauqsimanngittunga
Gloss	“I never said I wanted to go to Paris.”

Table 9.1: An Inuktitut word, built by concatenation, can contain as many morphemes as an average-length English sentence.

However, some languages do not form their words exclusively via concatenation. The terms **NON-CONCATENATIVE MORPHOTACTICS** or **NON-CONCATENATIVE PROCESSES** are commonly used to cover a variety of phenomena including reduplication, infixation, interdigitation and metathesis. The languages that exhibit non-concatenative processes are sometimes called non-concatenative languages; however, most of them also employ concatenation or are even principally concatenative, so the description “not totally concatenative” (Lavie et al., 1988) is usually more appropriate. We will first look very briefly at some non-concatenative phenomena in Tagalog, Malay and Arabic, then introduce the **compile-replace** algorithm, and then illustrate its use to handle those phenomena.

Fixed-Length Reduplication in Tagalog

In Tagalog, as reported in Antworth (Antworth, 1990), one derivational process, which we will refer to as fixed-length reduplication, involves copying the first CV syllable of a verb root, whatever that syllable might be (see Table 9.2).

Full-Stem Reduplication in Malay

In Malay, Indonesian and many other languages, whole stems, of whatever length, may be copied. We will refer to this as full-stem reduplication or variable-length reduplication (see Table 9.3).

ROOT	CV+ROOT	GLOSS
pili	pipili	“choose”
tahi	tatahi	“sew”
kuha	kukuha	“take”

Table 9.2: One fixed-length form of Tagalog reduplication involves copying the first CV syllable of a verb root and can be handled even using classic Two-Level Morphology.

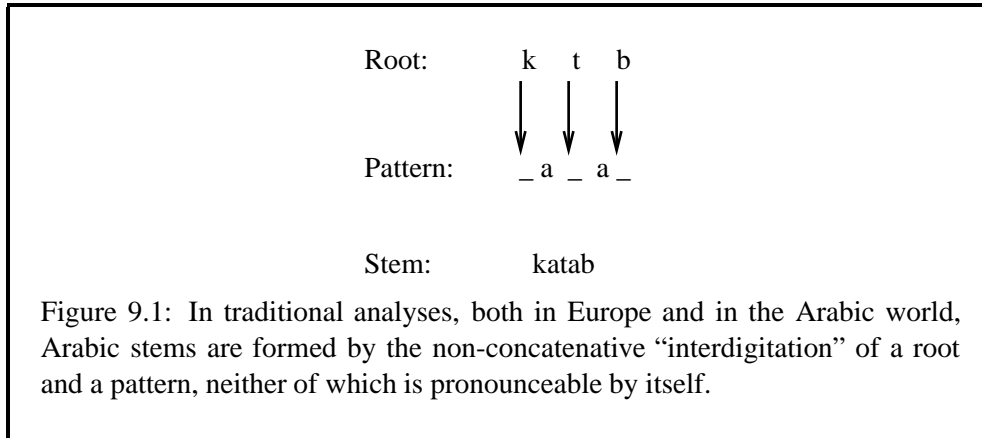
ROOT	GLOSS	REDUPLICATION	GLOSS
anak	“child”	anak-anak	“children”
lembu	“cow”	lembu-lembu	“cows”
buku	“book”	buku-buku	“books”
basikal	“bicycle”	basikal-basikal	“bicycles”

Table 9.3: Overt noun plurals in Malay and Indonesian are formed by full-stem reduplication. Such phenomena can be handled in finite-state morphology using the **compile-replace** algorithm.

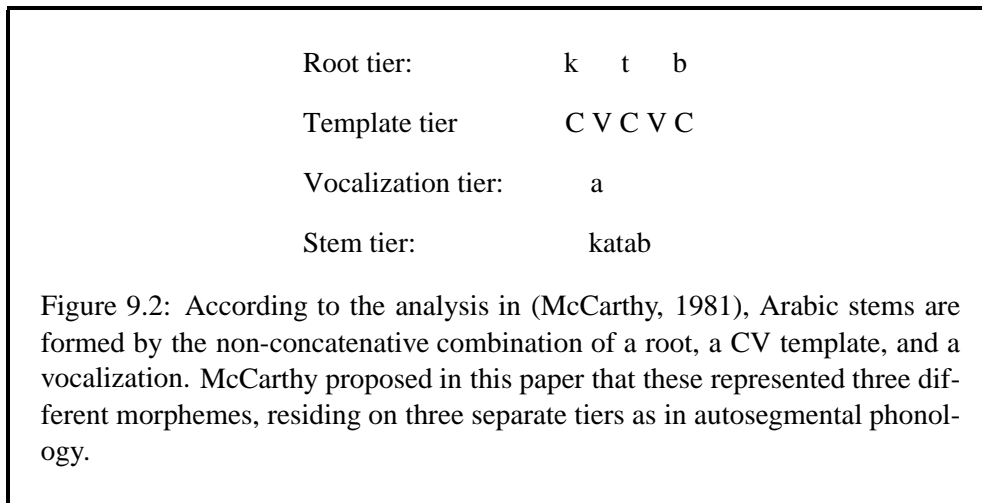
Stem Interdigitation in Arabic

In Arabic, and in other Semitic languages, prefixes and suffixes attach to stems via the usual process of concatenation, but the stems are generally held to be non-concatenatively composed of two or three morphemes, depending on one’s theory. The theory most commonly implemented in natural-language processing systems (see (Beesley, 1998c) for citations) postulates that stems are composed of a **ROOT**, usually consisting of three consonants like **ktb** or **drs**, and a **PATTERN** like **_a_a_** or **'i_ta_a_**, which consists of vowels, sometimes consonants and lengthening morphophonemes, and slots into which the consonants of the root are inserted. The root and pattern are said informally to be “interdigitated” together to form stems like *katab* and *'iktatab*. See Figure 9.1.

According to a classic and rather influential paper by McCarthy (McCarthy, 1981), Semitic stems are composed of not two but three morphemes, a root like **ktb** or **drs** (just as in the root-pattern theories), a template consisting of **C** (consonant) and **V** (vowel) slots, and a vocalization, e.g. **a** (denoting perfect aspect and active voice), **u i** (denoting perfect aspect and passive voice), etc. See Figure 9.2. McCarthy’s rules instantiate the **C** slots with consonants from the root, and the **V** slots with vowels from the vocalization, to form the stem. In essence,



the three-morpheme theories factor the pattern into two separate morphemes.



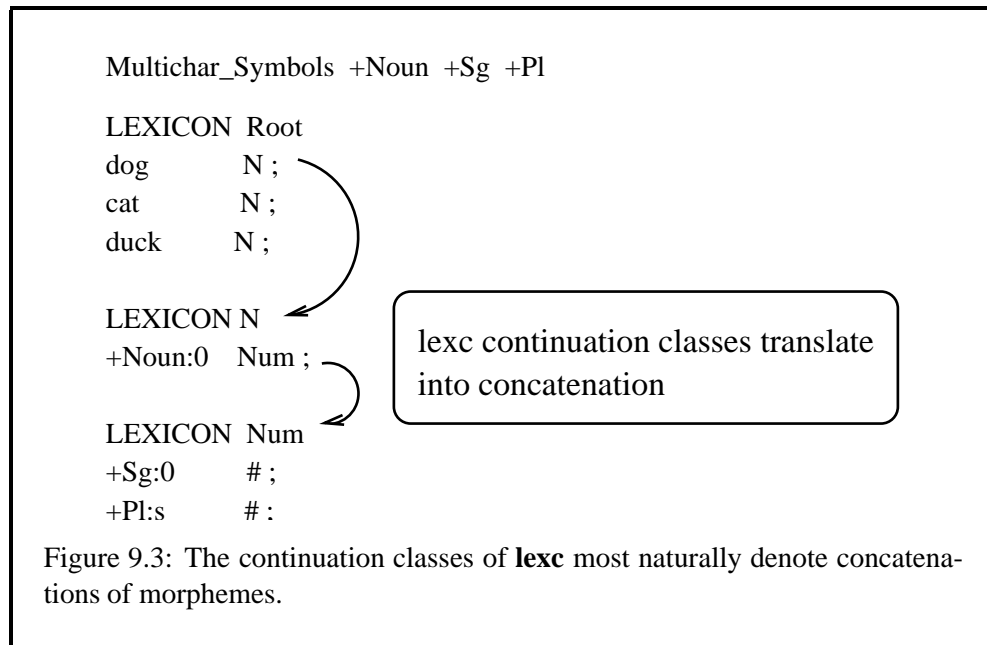
The difference between the two-morpheme and three-morpheme theories of Semitic stems is not significant here. Both approaches agree that the root is a separate morpheme and that the stem is not formed by concatenation; and both approaches can be modeled and computed using the **compile-replace** algorithm.

9.2 Formal Morphotactic Description

9.2.1 Concatenative Assumptions and Limitations

Before showing our solutions for Tagalog, Malay and Arabic, we must place the **compile-replace** algorithm in the context of general morphological description. Concatenative morphotactics is easily expressed in regular expressions via the simple juxtaposition of operands, and in formalisms like **lexc**, where LEXICON entries

themselves are typically just concatenations of symbols, and where continuation classes translate into concatenation between morphemes (see Figure 9.3).



Finite-state morphology in the tradition of the Two-Level (Koskenniemi, 1983) and **Xerox** implementations has been very successful in implementing large-scale, robust and efficient morphological analyzer-generators for concatenative languages, including the commercially important European languages and agglutinating non-Indo-European examples like Finnish, Turkish, Basque and Hungarian. However, Koskenniemi himself understood that his initial implementation had significant limitations in handling non-concatenative morphotactic processes:

Only restricted infixation and reduplication can be handled adequately with the present system. Some extensions or revisions will be necessary for an adequate description of languages possessing extensive infixation or reduplication. (Koskenniemi, 1983)

These traditional limitations have not escaped the notice of various reviewers, e.g. (Sproat, 1992), and non-concatenative morphotactics is rightly recognized as the cutting edge of computational morphology.

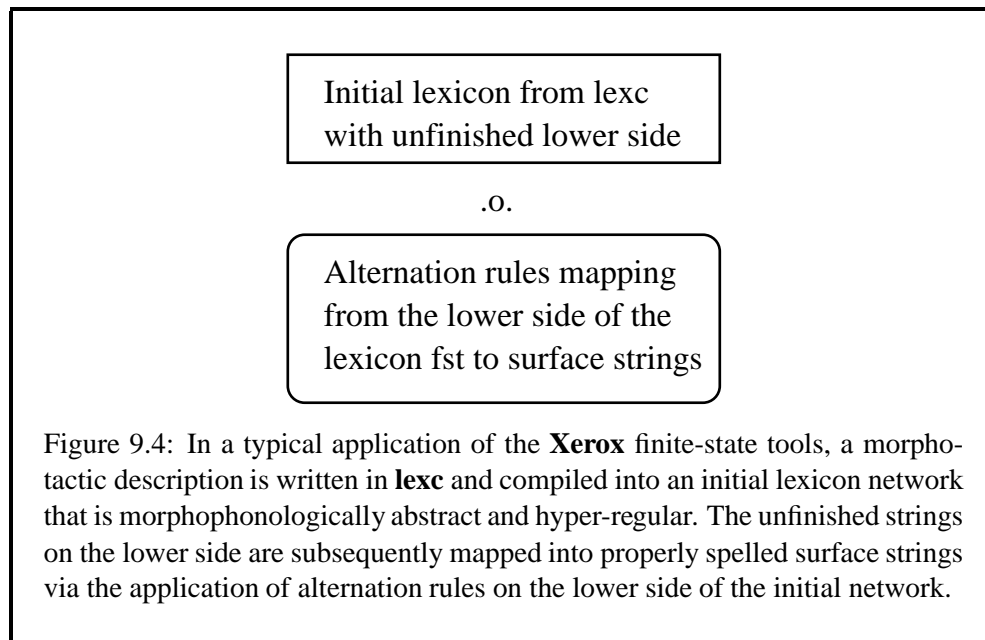
9.2.2 Overcoming the Concatenative Limitation

A New Mindset

Given that finite-state implementations have relied on **lexc** and similar formalisms, which most naturally denote concatenations of morphemes, it is not surprising that

they appear to be limited to modeling concatenative morphotactics. The way to overcome this limitation, in the broadest finite-state terms, is to open up morphotactic description to include finite-state operations other than concatenation, and this is what the **compile-replace** algorithm does.

When modeling a language with non-concatenative phenomena, the linguist starts in the usual way by writing an abstract morphotactic description using regular expressions or **lexc**, and compiling the description into a network. We are already comfortable with the notion that this initial lexicon network is phonologically and/or orthographically abstract, requiring the application of alternation rules to map the abstract strings into well-formed surface strings as shown in Figure 9.4. We are also comfortable with the idea that the initial lexicon network may over-generate, requiring restriction via composed filters or Flag Diacritics as shown in Figure 9.5.



We extend this notion of abstractness to include the possibility that the initial network is morphotactically abstract, including meta-morphotactic descriptions of non-concatenative phenomena. The subsequent application of **compile-replace** to the lower-side of a network as in Figure 9.6 makes the resulting lower-side strings more morphotactically surfacy, just as the application of alternation rules on the lower side of a network makes the resulting lower-side strings more phonologically or orthographically surfacy.

The notation we propose for meta-morphotactic description is already familiar to us; it is the language of regular expressions. The use of the **compile-replace** algorithm involves first using **lexc** or regular expressions as usual to define a network whose strings contain meta-morphotactic descriptions which are themselves in the

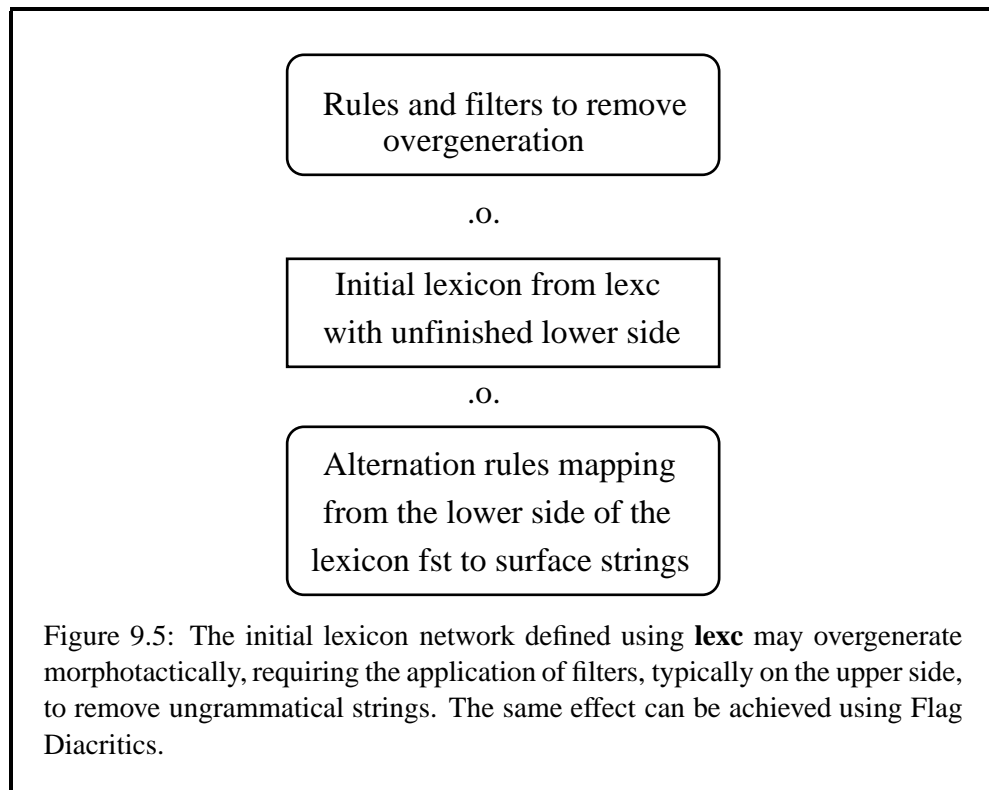
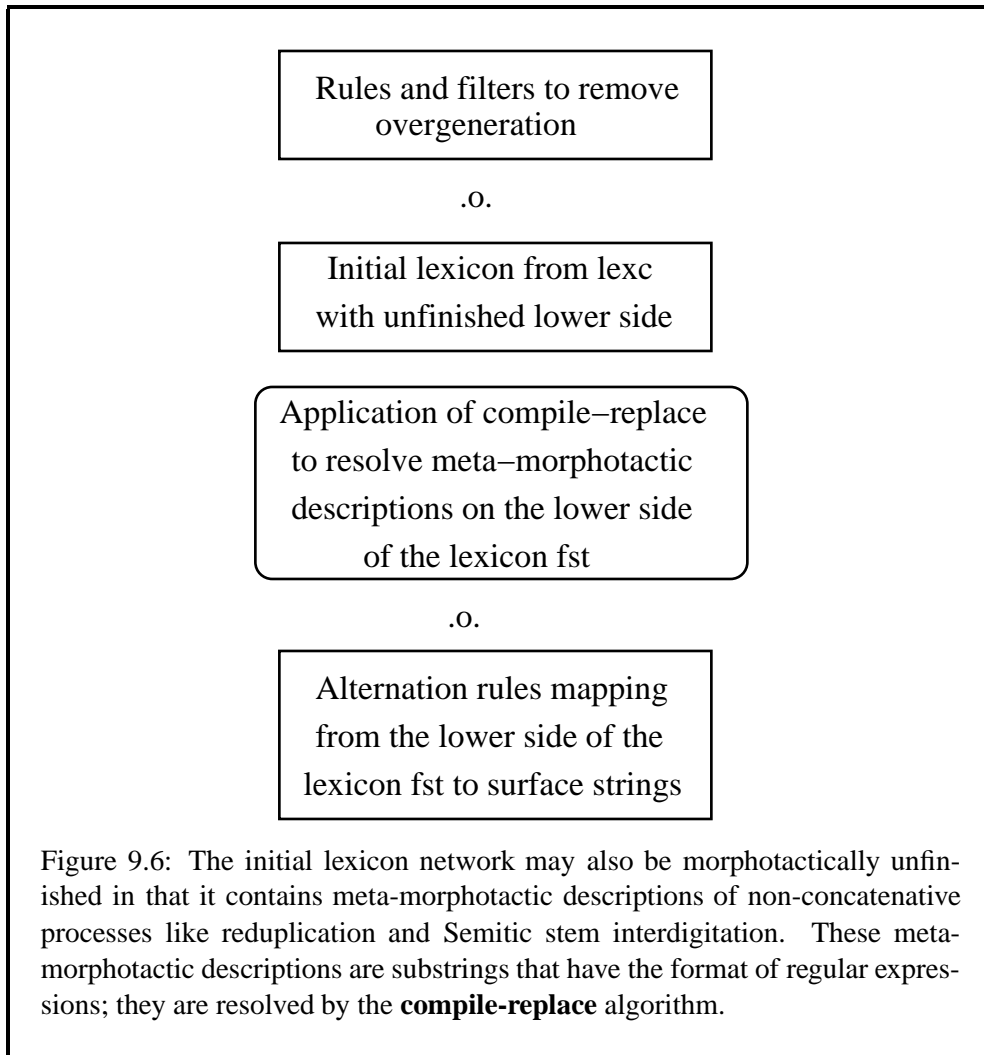


Figure 9.5: The initial lexicon network defined using **lexc** may overgenerate morphotactically, requiring the application of filters, typically on the upper side, to remove ungrammatical strings. The same effect can be achieved using Flag Diacritics.



format of regular expressions. Then **compile-replace** applies to the initial network, finding the meta-morphotactic descriptions, calling the regular-expression compiler to compile them as regular expressions, and replacing them in the network with the network resulting from the compilation.

Both **lexc** and the **read regex** utility of **xfst** produce finite-state networks and can be considered finite-state compilers. The **compile-replace** algorithm does a kind of bootstrapping, applying a regular-expression compiler to the output of a regular-expression compiler. The result is a modified but still finite-state network.

The Mechanics of compile-replace

The regular-expression substrings in the network to be modified by **compile-replace** must be delimited by the $\wedge[$ and $\wedge]$ multicharacter symbols. These delimiters were arbitrarily chosen and must be declared as `Multichar_Symbols` if they are introduced in a **lexc** description. Before **compile-replace** can work successfully, every substring of characters appearing on a subpath between $\wedge[$ and $\wedge]$ must be compilable as a regular expression; in other words, anything between $\wedge[$ and $\wedge]$ must look like a valid regular expression. It is the responsibility of the developer to make sure that the delimited substrings have the format of regular expressions, and this is a challenge best reserved for experienced developers. If any of the delimited substrings cannot be compiled as regular expressions, **compile-replace** will display appropriate error messages that include the offending substrings.

Before going into specific linguistic examples, we will start with a non-linguistic example that illustrates the operation of the algorithm. Figure 9.7 represents a network containing the lower-side substring “a*”, enclosed between the two delimiters, $\wedge[$ and $\wedge]$, that mark the path as a regular-expression substring. As far as the network itself is concerned, the delimited substring “a*” is simply a concatenation of the two symbols **a** and *****.



Figure 9.7: A network with a delimited substring that has the format of a regular expression. As far as the network itself is concerned, “a*” is just a concatenation of the two symbols **a** and *****.

The application of the **compile-replace** algorithm to the lower side of the network eliminates the delimiters and maps the string “a*” into the infinite language that the regular expression a^* denotes. The result is shown in Figure 9.8.

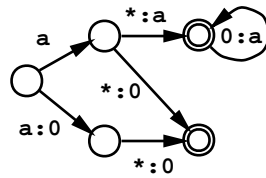


Figure 9.8: The network after the application of **compile-replace** to the lower side.

When applied in the upward direction, the transducer in Figure 9.8 maps any string of the infinite a^* language, i.e. “”, “a”, “aa”, “aaa”, ..., into the regular expression (i.e. a^*) from which the language was compiled.

The use of **compile-replace** is a challenge. We have to define an initial transducer (using **lexc** or regular expressions) that relates two regular languages as in Figure 9.7; and we have to define one or both of these languages so that they contain delimited substrings that themselves are valid **xfst** regular expressions. The biggest challenge is to visualize the non-concatenative phenomena in abstract finite-state terms and to define the delimited regular-expression substrings appropriately. This is best shown in practical examples.

9.3 Practical Non-Concatenative Examples

9.3.1 Restricted Reduplication in Tagalog

Antworth presented a kind of restricted reduplication that appears in Tagalog (Antworth, 1990)[pp. 151–162]. He handled it elegantly in classic KIMMO-style Two-Level Morphology, and his solution is easily reproduced using the **Xerox** formalisms, without any recourse to the **compile-replace** algorithm. We will briefly examine his Tagalog solution before moving on to the more difficult variable-length, full-stem reduplication seen in Malay.

ROOT	CV+ROOT	GLOSS
pili	pipili	“choose”
tahi	tatahi	“sew”
kuha	kukuha	“take”

Table 9.4: Limited Reduplication of the Initial CV Syllable in Tagalog Verbs

The Tagalog reduplication involves the copying of the initial CV syllable, what-

ever that syllable might be. The bare roots are also valid words. The sample data is repeated here in Table 9.4. In his lexicon, Antworth formalizes the reduplication at the lexical level as an abstract prefix of the form **RE+**, where **R** stands for a consonant and **E** for a vowel. A bare-bones recreation of his dictionary might look like this in **lexc**:

```
LEXICON Root
RE+      VRoots ;
         VRoots ;    ! empty entry
```

```
LEXICON VRoots
pili     # ;
tahi     # ;
kuha     # ;
```

Thus the initial lexicon network generates both *pili* and *RE+pili*. Using alternation rules, we then simply realize **R** as a copy of the first following consonant and **E** as a copy of the first following vowel. It happens that **twolc** rules are rather more convenient than **xfst** replace rules for this purpose. We assume that **Conson** is defined as an appropriate set of root-initial consonants and **Vowel** as an appropriate set of vowels; we also assume that the **twolc** alphabet contains the pair **%+:0** and no other pair with the literal plus sign on the top.

```
Alphabet %+:0 p t k a e i o u ;
```

```
Sets
```

```
Conson = p t k ;    ! expand as appropriate
Vowel = a e i o u ;
```

```
Rules
```

```
"R realization as Consonant"
```

```
R:CC <=> _ E: %+: CC ; where CC in Conson ;
```

```
"E realization as Vowel"
```

```
E:VV <=> _ %+: (Conson:) VV ; where VV in Vowel ;
```

As desired, these straightforward rules will map the abstract *RE+pili* into *pipili*, *RE+tahi* into *tatahi*, etc. Antworth's original rules also handle some infixation, which can act alone or in combination with reduplication; this is a classic example of the "restricted infixation and reduplication" that Koskenniemi knew his grammars could handle.

The key to Antworth’s Tagalog solution is his abstraction away from the varying surfacy realizations, finding a single consistent notation, i.e. $RE+stem$, of the reduplication at an underlying level. Such abstraction is also the key to using **compile-replace**.

9.3.2 Full-Stem Reduplication in Malay

The Tagalog examples just shown are relatively easy to handle because the substring to be reduplicated is limited and predictable in its length; examples of more difficult full-stem reduplication occur in Malay and Indonesian, which are closely related. The Malay stem *buku* means “book”; this bare form is in fact number-neutral and can translate in some contexts as the English plural. The overt plural is phonologically *bukubuku*,¹ formed by reduplicating the entire stem. Stems vary in length: the overt plural of *pelabuhan* (“port”), itself a derived form, is *pelabuhan-pelabuhan*.

To understand the general solution to full-stem reduplication using the **compile-replace** algorithm requires a bit of background. Recall that in the **Xerox** regular-expression syntax, expressions of the form A^n , where n is an integer, denote n concatenations of A . Recall also that $\{abc\}$ denotes the concatenation of the symbols **a**, **b**, and **c**.

The full reduplication of any string s can then be notated uniformly, in finite-state terms, as $\{s\}^2$. This is our abstract meta-description of full-stem reduplications, and we start by defining a network where the lower-side strings are built by straightforward concatenation of a prefix \wedge [, a root enclosed in braces, and an abstract overt-plural suffix $\wedge 2$ followed by the closing delimiter \wedge]. A simple **lexc**

¹In the standard orthography, such reduplicated words are now written with a hyphen, e.g. *buku-buku*, that we will ignore in this first example.

grammar to illustrate the technique is the following:

```
Multichar_Symbols ^[ ^] +Noun +Unmarked +Plural
```

```
LEXICON Root
0:^[{  NRoots ;
      NRoots ;
```

```
LEXICON NRoots
buku      NSuff ;
lembu     NSuff ;
pelabuhan NSuff ;
```

```
LEXICON NSuff
+Noun:0    Num ;
```

```
LEXICON Num
+Unmarked:0 # ;
+Plural:}^2^ # ;
```

Expressed in an **xfst** script using regular expressions, the equivalent grammar is

```
define Prefix 0 .x. "^[" "{" ;
define NRoots {buku} | {lembu} | {pelabuhan} ;
define NSuff  "+Noun":0 ;
define Num    "+Unmarked":0 |
              [ "+Plural" .x. "]" "^" 2 "^" ] ;
read regex (Prefix) NRoots NSuff Num ;
```

Compilation of either grammar will yield an initial transducer wherein the lower-side language contains ill-formed strings having unmatched delimiters. These undesirable strings can be removed easily by composing the following regular-expression filter, compiled in **xfst**, on the lower side:

```
~[ ?* "^[" ~$["^"] ] ] & ~[ ~$["^[" "]" ?* ] ]
```

The various grouping and literal brackets make the expression difficult to read, but in the end the application of the filter will simply eliminate all paths where the lower-side string contains an opening delimiter `^[` without a matching closing delimiter `^]`, or a closing delimiter `^]` without a matching opening delimiter `^[`. Lining up the symbols for easy comparison with the grammar, the valid overt-plural paths look like this

```
Upper:      b u k u +Noun +Plural
Lower:     ^[ { b u k u      } ^ 2 ^]
```

Upper: p e l a b u h a n +Noun +Plural
 Lower: $\hat{[\{ p e l a b u h a n \} ^ 2 \hat{]}$

and without the lineup, the lower-side strings are simply $\hat{[\{ buku \} ^ 2 \hat{]}$ and $\hat{[\{ pelabuhan \} ^ 2 \hat{]}$.

The **compile-replace** algorithm, applied to the lower-side of this network, recognizes each individual delimited regular-expression substring like $\hat{[\{ buku \} ^ 2 \hat{]}$, compiles it, and replaces it with the result of the compilation, here *bukubuku*. The same process applies to the entire lower-side language, resulting in a network that directly relates pairs of strings like the following.

Upper: buku+Noun+Plural
 Lower: bukubuku

Upper: pelabuhan+Noun+Plural
 Lower: pelabuhanpelabuhan

This provides the desired solution, still completely finite-state, for analyzing and generating full-stem reduplication in Malay.

The key to the solution for full-stem reduplication is to abstract away from the surface phenomena and to find an underlying meta-morphotactic description, here $\{s\}^2$, that uniformly describes full-stem reduplication for any stem *s*.

9.3.3 More Malay Reduplication

There are in fact at least a dozen derivational processes in Malay that involve reduplication, and one of the authors has written a prototype morphological analyzer that so far has managed to handle all the processes with minor variations of the technique just shown.

The following Malay/Indonesian grammar fragment illustrates how one might handle some more complicated kinds of full-stem reduplication. Malay has no real inflectional morphology, but stems can undergo dozens of derivational processes involving prefixation, suffixation, circumfixation (a coordinated combination of prefixation and suffixation), infixation and reduplication. The reduplication can occur by itself or together with other derivational processes, and in this example we also handle the orthographical hyphen that appears between reduplicated elements. The analysis has been simplified slightly to make a manageable example, and we mix Malay and Indonesian stems.

Any individual stem takes only an idiosyncratic subset of the possible derivational processes, and these need to be marked explicitly in the dictionary. On the

Suffix Tag	Human Reading
+an	Suffix -an attaches after the stem.
+meN-	Prefix meN- attaches before the stem.
+per-an	Circumfixation of the stem by prefix per- and suffix -an.
+meN-kan	Circumfixation of the stem by prefix meN- and suffix -kan.
+redup[-]	Reduplication of the stem.
+meN[redup[-]]	Reduplication of the stem, with a meN- prefix on the result.
+meN[redup[-]]kan	Reduplication of the stem, then that result circumfixed by prefix meN- and suffix -kan.

Table 9.5: Multicharacter suffix tags are designed to indicate, to human readers, the prefixation, suffixation, circumfixation, reduplications, etc. that are involved in a particular derivation of a stem.

upper side of the lexicon transducer we will represent each of these derivational processes as a single multicharacter tag suffix, spelling the multicharacter tags in a human-readable way that reflects the formal notations in some Malay grammars and dictionaries (Echols and Shadily, 1989). For example, Table 9.5 shows a few tags and their intended readings. It must be emphasized that tags like `+per-an` and `+meN[redup[-]]kan` mean nothing to the system; they are just multicharacter symbols that were spelled for human convenience.

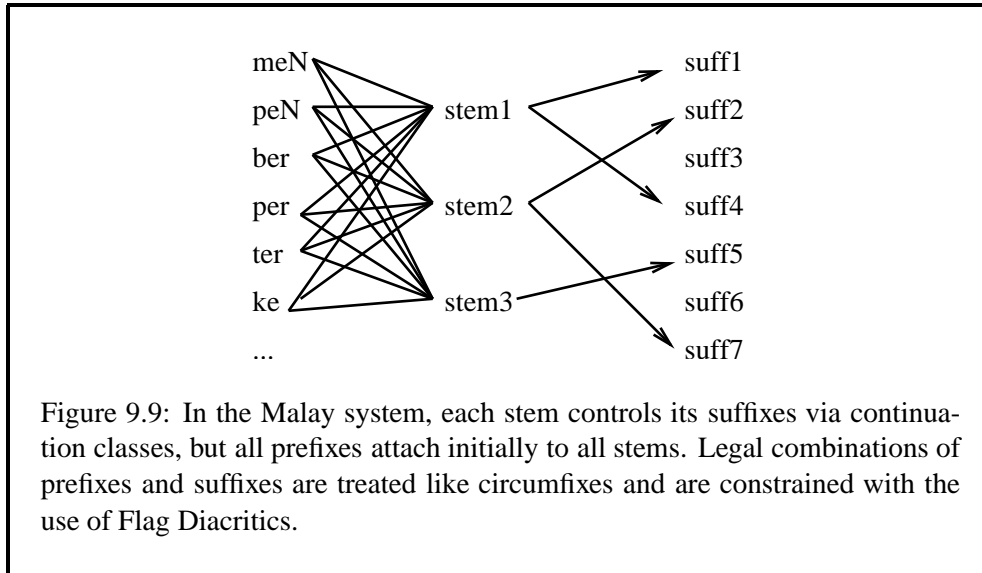
In overview, we formalize Malay morphotactics as involving the set of all prefixes concatenating to all stems (see Figure 9.9), and each stem has a continuation class or classes that lead to the idiosyncratic set of “derivation suffixes” allowed for that stem. The scheme threatens to overgenerate wildly, for while the suffixes of a given stem can be strictly controlled by its continuation class, the prefixes attach promiscuously to all stems. We will use Flag Diacritics, and in particular the P, R and D types, to constrain the dependencies; stems constrain their suffixes, and Flag Diacritics in the suffixes will in turn constrain the allowable prefixes. This is a classic example of flag-constrained circumfixation, of which a simpler example is shown in Section 8.5.4.

The prefixes and suffixes can be rather abstract, and one of the prefixes is empty. All prefixes in this grammar have the empty string on the upper side. Let us assume that the `lexc` grammar below is in the file `malay-lexc.txt`.

```
! malay-lexc.txt

Multichar_Symbols

! tag suffixes for the upper side
```



```
+Bare +-an +ke-an +per-an +peN-an
+meN- +meN-i +redup[-]
+meN-kan +meN[redup[-]] +meN[redup[-]]kan

! flag diacritics

@P.PREF.peN@ @R.PREF.peN@
@P.PREF.ter@ @R.PREF.ter@
@P.PREF.ber@ @R.PREF.ber@
@P.PREF.per@ @R.PREF.per@
@P.PREF.ke@ @R.PREF.ke@
@P.PREF.redup[-]@ @R.PREF.redup[-]@
@P.PREF.meN@ @R.PREF.meN@
@P.PREF.meN[redup[-]]@ @R.PREF.meN[redup[-]]@
@D.PREF@

! morphophoneme
^N          ! underspecified nasal
^REHYPH     ! reduplication hyphen

!***** end of Multichar_Symbols*****

LEXICON Root
      Prefixes ;

LEXICON Prefixes
```

```

                                                    Stems ;
! the empty prefix
< [ 0 .x. p e "^N" ] "@P.PREF.peN@" >      Stems ;
< [ 0 .x. t e r ]      "@P.PREF.ter@" >      Stems ;
< [ 0 .x. b e r ]      "@P.PREF.ber@" >      Stems ;
< [ 0 .x. p e r ]      "@P.PREF.per@" >      Stems ;
< [ 0 .x. k e ]        "@P.PREF.ke@" >      Stems ;
< [ 0 .x. "^[" "[" "{" ]
                        "@P.PREF.redup[-]@" > Stems ;
< [ 0 .x. m e "^N" ] "@P.PREF.meN@" >      Stems ;
< [ 0 .x. m e "^N" "^[" "[" "{" ]
                        "@P.PREF.meN[redup[-]]@" > Stems ;

```

LEXICON Stems

```

ajar    bare ;
ajar    meN- ;
ajar    meN-kan ;
capai   meN- ;
curah   meN[redup[-]] ;
dapat   meN- ;
baik    redup[-] ;
bagi    -an ;
bagi    meN[redup[-]] ;
baka    meN- ;
pakai   meN- ;
paksa   meN[redup[-]] ;
payah   meN[redup[-]]kan ;
taman   meN- ;
tampar  meN[redup[-]] ;
galah   meN- ;
kaji    meN- ;
kuyuh   meN[redup[-]] ;
salak   meN- ;
sama    meN[redup[-]] ;
maki    meN- ;

```

LEXICON bare

```

< "+Bare":0           "@D.PREF@" >      # ;

```

LEXICON -an

```

< [ "+-an" .x. a n ] "@D.PREF@" >      # ;
! simple suffix blocks all prefixes

```

LEXICON ke-an

```

< [ "+ke-an" .x. a n ] "@R.PREF.ke@" > # ;
! circumfix requires ke-

LEXICON per-an
< [ "+per-an" .x. a n ] "@R.PREF.per@" > # ;

LEXICON peN-an
< [ "+peN-an" .x. a n ] "@R.PREF.peN@" > # ;

LEXICON meN-
< "+meN-":0 "@R.PREF.meN@" > # ;
! require simple prefix, no phonological suffix

LEXICON meN-i
< "+meN-i":i "@R.PREF.meN@" > # ;

LEXICON redup[-]
< "+redup[-]" .x. "}" "%^REHYPH" "]" "^" 2 "]"
"@R.PREF.redup[-]@" > # ;

LEXICON meN-kan
< [ "+meN-kan" .x. k a n ] "@R.PREF.meN@" > # ;

LEXICON meN[redup[-]]
< [ "+meN[redup[-]]" .x.
  "}" "%^REHYPH" "]" "^" 2 "]"
  ] "@R.PREF.meN[redup[-]]@" > # ;

LEXICON meN[redup[-]]kan
< [ "+meN[redup[-]]kan" .x.
  "}" "%^REHYPH" "]" "^" 2 "]" k a n
  ] "@R.PREF.meN[redup[-]]@" > # ;

```

We can compile this grammar inside the **lexc** interface, of course, or from the **xfst** interface using the **read lexc** command. We ignore the compiler warnings, not shown here, which simply tell us that some of the defined suffixes are not yet being used.

```

xfst[0]: read lexc < malay-lexc.txt
4.5 Kb. 100 states, 136 arcs, 189 paths.
xfst[1]: eliminate flag PREF
4.1 Kb. 95 states, 114 arcs, 21 paths.
xfst[1]: print lower-words
^[[{baik}%^REHYPH]^2^]

```

```

me^N^ [[ {tampar}%^REHYPH]^2^ ]
me^N^ [[ {sama}%^REHYPH]^2^ ]
me^N^ [[ {payah}%^REHYPH]^2^ ] kan
me^N^ [[ {paksa}%^REHYPH]^2^ ]
me^N^ [[ {kuyuh}%^REHYPH]^2^ ]
me^N^ [[ {curah}%^REHYPH]^2^ ]
me^N^ [[ {bagi}%^REHYPH]^2^ ]
me^Ntaman
me^Nsalak
me^Npakai
me^Nmaki
me^Nkaji
me^Ngalah
me^Ndapat
me^Ncapai
me^Nbaka
me^Najar
me^Najarkan
bagian
ajar

```

Notice that eight of the lower-side strings contain delimited regular expressions that will ultimately be handled by the **compile-replace** algorithm. But before we invoke **compile-replace lower**, we need to consider assimilation and re-syllabification.² Here we will treat a fairly straightforward assimilation to point-of-articulation: The underlying underspecified-nasal \hat{N} disappears completely before nasals, liquids, and semi-vowels; It assimilates to /m/ before labial and labiodental stops and fricatives (/p/, /b/ and /f/), to /nj/ (orthographical **ny**) before /s/, to **n** before other alveolars, and to /ŋ/ (orthographical **ng**) before vowels, velars and /h/. Thus *me^Nbaka* should be realized as *membaka*, *me^Ndapat* as *mendapat*, *me^Ngalah* as *menggalah*, *me^Nmaki* as *memaki*, etc. The first-draft rules to handle these assimilations, using orthographical notation, are shown in Figure 9.10.

There are, however, some interesting complications, the first involving assimilation of \hat{N} with voiceless stops and /s/. Underlying *me^Npakai* is realized not as **mempakai* but as *memakai*; the **p** is effectively deleted. Similarly, *me^Ntaman* is realized as *menaman*, *me^Nkaji* as *mengaji*, and *me^Nsalak* as *menyalak*. Thus the \hat{N} assimilates to the place of articulation of the following underlying voiceless consonant, but the consonant itself doesn't reach the surface. The second-draft rules in Figure 9.11 handle these deletions as well; note that we use the double comma to separate rules that are to be compiled in parallel.

²I wish to thank Prof. George Poulos of the University of South Africa for clarifying to me the effect of re-syllabification in some of the reduplication phenomena. Similar cases can be seen in Bantu languages.


```

%N -> m | | _ [ p | b | f ]
.ο.
%N -> n | | _ [ t | d | c | j | s y | z ]
.ο.
%N -> n y | | _ s
.ο.
%N -> n g | | _ [ a | e | i | o | u | k | g | h ]
.ο.
%N -> 0 | | _ [ m | n | l | r | w | y ]

```

Figure 9.10: First-Draft Rules to Handle Nasal Assimilation in Malay

Another complication concerns reduplication. Looking first at a straightforward example, our grammar includes the abstract $^{\wedge}[[\{\text{baik}\}^{\wedge}\text{REHYPH}]^{\wedge}2^{\wedge}]$, which after **compile-replace** will be $\text{baik}^{\wedge}\text{REHYPH}\text{baik}^{\wedge}\text{REHYPH}$. The application of the following rules

```

xfst[1]: define HyphenRules %REHYPH -> 0 | | %REHYPH ?* _
.ο.
%REHYPH -> "-" ;

```

will then map $\text{baik}^{\wedge}\text{REHYPH}\text{baik}^{\wedge}\text{REHYPH}$ to *baik-baik*, which is the final surface form.

The most interesting cases of reduplication occur when assimilation and deletion affect the syllabification of the stem. When *me^Ntaman* is realized as *menaman*, the resulting syllable structure changes from abstract $/\text{me}^{\wedge}\text{N.ta.man}/$ to $/\text{me.na.man}/$. In examples such as $\text{me}^{\wedge}\text{N}^{\wedge}[[\{\text{tampar}\}^{\wedge}\text{REHYPH}]^{\wedge}2^{\wedge}]$, the result is not **menampar-tampar* but *menampar-nampar*. Similarly, the underlying string

$\text{me}^{\wedge}\text{N}^{\wedge}[[\{\text{paksa}\}^{\wedge}\text{REHYPH}]^{\wedge}2^{\wedge}]$

is realized as *memaksa-maksa*,

$\text{me}^{\wedge}\text{N}^{\wedge}[[\{\text{payah}\}^{\wedge}\text{REHYPH}]^{\wedge}2^{\wedge}]\text{kan}$

as *memayah-mahahkan*,

$\text{me}^{\wedge}\text{N}^{\wedge}[[\{\text{kuyuh}\}^{\wedge}\text{REHYPH}]^{\wedge}2^{\wedge}]$

as *menguyuh-gnuyuh*,

$\text{me}^{\wedge}\text{N}^{\wedge}[[\{\text{sama}\}^{\wedge}\text{REHYPH}]^{\wedge}2^{\wedge}]$

```

[ %^N -> m || _ [ p | b | f ] ,, p -> 0 || %^N _ ]
.o.
[ %^N -> n || _ [ t | d | c | j | s y | z ] ,,
t -> 0 || %^N _ ]
.o.
[ %^N -> n y || _ s ,, s -> 0 || %^N _ ]
.o.
[ %^N -> n g || _ [ a | e | i | o | u | k | g | h ] ,,
k -> 0 || %^N _ ]
.o.
%^N -> 0 || _ [ m | n | l | r | w | y ]

```

Figure 9.11: Second-Draft Rules for Malay Nasal Assimilation

as *menyama-nyama*, etc. The assimilation and deletion conceptually occur first, modifying the stem, and then the modified stem is reduplicated. We can visualize the *maksa* example in the following way:

```

Lexicon upper level:      paksa+meN[redup[-]]
Lexicon lower level: me^N^[[{paksa}%^REHYPH]^2^]

```

Eliminate the PREF flag

Apply the assimilation/deletion rules

```

Intermediate level:  me^[[{maksa}%^REHYPH]^2^]

```

Apply compile-replace

```

Intermediate level:  memaksa^REHYPHmaksa^REHYPH

```

Apply the hyphenation rules

```

Surface level:      memaksa-maksa

```

The final system should be able to analyze *memaksa-maksa* and return `paksa+meN[redup[-]]`, which clearly identifies the stem *paksa*, which is the headword in any standard paper dictionary.

The final draft rules for assimilation and deletion are then as follows:

```

xfst[1]: define lbraces "^[" | "[" | "{" ;

xfst[1]: define AssimDelRules [
p  -> m || %^N lbraces* _ ,,
%^N -> 0 || _ lbraces* p ]
.o.
%^N -> m || _ lbraces* [ b | f ]
.o.
[ t -> n || %^N lbraces* _ ,,
%^N -> 0 || _ lbraces* t ]
.o.
%^N -> n || _ lbraces* [ d | c | j | s y | z ]
.o.
[ s -> n y || %^N lbraces* _ ,,
%^N -> 0 || _ lbraces* s ]
.o.
[ k -> n g || %^N lbraces* _ ,,
%^N -> 0 || _ lbraces* k ]
.o.
%^N -> n g || _ lbraces* [ a | e | i | o | u | g | h ]
.o.
%^N -> 0 || _ lbraces* [ m | n | l | r | w | y ] ;

```

We can now build and test the entire system by performing the following steps in order:

1. Compile the **lexc** file malay-lexc.txt
2. Eliminate the PREF flag
3. Apply the AssimDelRules
4. Apply **compile-replace lower**
5. Apply the HyphenRules

```

xfst[0]: read lexc < malay-lexc.txt
4.5 Kb. 100 states, 136 arcs, 189 paths.
xfst[1]: eliminate flag PREF
xfst[1]: define initialnet
defined initialnet: 4.1 Kb. 95 states, 114 arcs, 21 paths.
xfst[0]: read regex initialnet .o. AssimDelRules ;
4.4 Kb. 107 states, 126 arcs, 21 paths.
xfst[1]: compile-replace lower

```

```

8 regular expressions compiled successfully. No errors.
4.8 Kb. 120 states, 139 arcs, 21 paths.
xfst[1]: define intermediatenet
defined intermediatenet: 4.8 Kb. 120 states, 139 arcs, 21 paths.
xfst[0]: read regex intermediatenet .o. HyphenRules ;
4.7 Kb. 118 states, 137 arcs, 21 paths.
xfst[1]: print lower-words
menguyuh-nguyuh
mengaji
menyama-nyama
menyalak
menampar-nampar
menaman
menggalah
mengajar
mengajarkan
mendapat
mencurah-curah
mencapai
memayah-mayahkan
memakai
memaksa-maksa
memaki
membaka
membagi-bagi
baik-baik
bagian

```

We can now play with **apply up** to see how these surface strings are analyzed:

```

xfst[1]: apply up
apply up> ajar
ajar+Bare
apply up> bagian
bagi+-an
apply up> membaka
baka+meN-
apply up> membagi-bagi
bagi+meN[redup[-]]
apply up> menampar-nampar
tampar+meN[redup[-]]
apply up> memaksa-maksa
paksa+meN[redup[-]]
apply up> memayah-mayahkan

```

```

payah+meN[redup[-]]kan
apply up> menguyuh-nguyuh
kuyuh+meN[redup[-]]
apply up> menyama-nyama
sama+meN[redup[-]]
apply up> END;
xfst[1]:

```

Note that each solution starts with a stem, and that the derivational process is represented as single multicharacter-symbol tag following the stem.

9.3.4 Semitic Stem Interdigitation

The Traditional Conception of Interdigitation

The non-concatenative nature of Semitic stem formation has been appreciated for centuries. Traditional Arabic grammarians identified abstract roots underlying whole families of derivationally related words; the root **ktb**, for example, is found in words relating to writing, documents, libraries, bookstores, offices, schools, etc. The number of roots used in Modern Standard Arabic is estimated somewhere between 5000 and 7000.

The root itself consists of consonants only and is unpronounceable; the root **ktb** should not be confused with the pronounceable citation form *kataba* conventionally used to refer to the root. The word *kataba* is in fact the perfect aspect, active voice form of **ktb**, with an *a* suffix for third-person masculine singular; this citation form translates roughly as “he wrote”. The vowels that appear between the consonants or RADICALS of the root are, according to traditional theory, part of a separate morpheme called the pattern. In this case the pattern could be formalized as **CaCaC** or **_a_a_**, with the **Cs** or underscores indicating slots where the radicals are to be inserted or interdigitated to form a stem.³ Another root like **drs** could combine with the same pattern to form the stem *daras*. Prefixes and suffixes can then attach to the resulting stems in the usual concatenative way.

In the traditional linguistic analysis of Semitic stems (see (Harris, 1941)), patterns can contain radical slots, vowels, non-radical consonants and morphophonemes that indicate lengthening or doubling of the previous vowel or consonant. The **Xerox** Arabic morphological analyzer⁴ is based on this traditional division of roots and patterns, and the dictionaries contain over 4900 roots and approximately 400 phonologically distinct patterns, many of which are ambiguous.

In traditional Arabic lexicography, including the best dictionaries produced in the European Arabist tradition, words must be looked up under root headings. This means that one must somehow know or surmise the root in order to look up a word.

³In the native tradition, the real Arabic root **ʔl** is used conventionally as a filler for citing pattern paradigms, e.g. *faʔal* rather than *CaCaC*

⁴<http://www.arabic-morphology.com/>

As root radicals are sometimes separated, assimilated, or even missing completely in surface words, root identification and dictionary lookup are difficult subjects for all students and many natives. Arabic is therefore a language where morphological analysis, returning the root, is required even for simple dictionary lookup.

Three Formalizations of Semitic Stems

When writing a morphological analyzer for a Semitic language, there are at least three possible approaches to handling stems: First, one could simply ignore all tradition and treat whole stems as indivisible units; this reduces Semitic morphotactics down to concatenation. Second, one could model the traditional two-way division of stems into roots and patterns, writing a morphological analyzer that somehow manages to separate and identify them as separate morphemes in analyses. Third, one could model a three-way division of stems into ROOT, TEMPLATE and VOCALIZATION as proposed in the paper by McCarthy.⁵

McCarthy's 1981 paper, though not computational, was well written and provided enough data for others to test their theories and programs. In his formalization, a Semitic stem is a composite of three separate morphemes: a root like **ktb**, a CV template like CVCVC or CVVCVC, and a vocalization like **a** or **u i**. In a rough summary, this three-way theory posits the same roots as the two-way theories, but the patterns are factored into two separate morphemes.

McCarthy's notation was based on autosegmental phonology, proposing that each of these morphemes occupies a separate TIER, and that the root, template and vocalization tiers combine to produce a stem tier. A passive example, using the vocalization **u i** is shown in Figure 9.12.

Root tier	k t b
Template tier	C V C V C
Vocalization tier	u i
Stem tier	kutib

Figure 9.12: In McCarthy's 1981 analysis, a perfect passive stem like *kutib* was produced by instantiating C slots in a template with root radicals and V slots with vowels from a perfect passive vocalization, which he notated as **u i**.

The rules that inserted the radicals into the C slots and the vowels into the V slots in McCarthy's paper 1981 paper were criticized from several quarters, see

⁵(McCarthy, 1981)

for example (Hudson, 1986), and McCarthy himself eventually moved on to other formalizations of Arabic. Martin Kay (Kay, 1987) reformalized the autosegmental tiers of McCarthy as projections of a multi-level transducer and wrote a small Prolog-based prototype that handled the interdigitation of roots, CV-templates and vocalizations into abstract Arabic stems; this general approach using multi-tape transducers has been explored and extended by George Kiraz in several papers (Kiraz, 1994a; Kiraz, 1996; Kiraz, 1994b; Kiraz, 2000) with application to Syriac. The implementation is described in (Kiraz and Grimley-Evans, 1999).

In what follows, we will generally assume that it is important for a Semitic morphological analyzer to return analyses that identify at least the root, and probably the pattern as well, though we will also show how such a system could be modified to return whole undivided stems. Our solutions are based on the formalization of Semitic stem interdigitation as finite-state intersection.

Stem Interdigitation as Intersection

In finite-state terms, the morphotactic formation of Semitic stems can be formalized very cleanly and intuitively as intersection. That is, the roots and patterns of the two-way analysis, or the roots, templates and vocalizations of the three-way analysis, can be formalized as regular languages, and stems can be formalized as the intersections of those regular languages. We will review the history of this formalization and the various ways that this intersection has been implemented in practical systems. Our own best solution will involve the **merge** operation, which is an efficient subtype of general intersection, and the **compile-replace** algorithm.

To our knowledge, the first researchers who overtly formalized Semitic stem interdigitation as finite-state intersection were Kataja and Koskenniemi (Kataja and Koskenniemi, 1988), who were working on Ancient Akkadian, which is a Semitic language, using a classic Two-Level Morphology. This was the first hint that morphotactic description could be opened up to include finite-state operations other than just concatenation.

In this and other systems, the formalization must be distinguished from the actual computational implementation; classic Two-Level Morphology did not, in fact, have any kind of intersection algorithm to work with. Roots like **ktb** and patterns like **CaCaC** existed in separate source lexicons, but in a pre-compilation step they were combined by awk scripts into the stem *katab* and written into a new source file for the TwoL compiler, which is similar to **lexc**. This TwoL file was then compiled into a standard lexicon.

This approach was not completely satisfactory. The lexicons in a classic Two-Level system are one-level,⁶ and once **ktb** and **CaCaC** had been pre-intersected into *katab* before compilation, the resulting morphological analyzer had no way to separate the root and pattern as part of the analysis. One would, for example,

⁶The two levels in Two-Level Morphology refer to the lexicon level and the surface level, which are related by a single set of rules compiled in parallel, as in **twolc**.

analyze the input string *kataba* and get back the stem *katab* and the suffix *a*, just as in a purely concatenative analysis based on stems.

In 1989 at the ALPNET company, Beesley (Beesley, 1989; Beesley et al., 1989; Beesley, 1990; Beesley, 1991) took this intersection formalization from Kataja and Koskenniemi and implemented it differently. Using an enhanced Two-Level implementation, he performed the combination of the root and pattern at runtime, in a way that kept the root and pattern separate on the lexical side but effectively created an intermediate level that included the intersected but still abstract stem. Again, there was no actual finite-state algorithm to perform the intersection, which was simulated at runtime in C code in a technique he called “detouring”.

```

Analysis level:      prefixes  ktb CaCaC  suffixes

Intermediate level
(at runtime):      prefixes   katab   suffixes

```

Then two-level rules mapped between the intermediate level and the real surface strings. Analysis of *kataba* yielded an analysis string like

```

      ktb  CaCaC  a

```

that showed the root, pattern and suffix clearly as separate morphemes. This was an improvement over the Kataja and Koskenniemi implementation, but it ran rather slowly, at about two words per second on a small IBM mainframe.

When Beesley joined **Xerox**, the old dictionaries were licensed from ALPNET and the system was completely rewritten using **Xerox** finite-state morphology (Beesley, 1996; Beesley, 1998a). The dictionaries were compiled as two-level transducers with the patterns initially concatenated after the roots, both on the upper and lower sides.

```

Upper:      prefixes  ktb  CaCaC  suffixes
Lower:      prefixes  ktb  CaCaC  suffixes

```

An elaborate and rather inefficient compile-time algorithm was then applied to this initial lexicon transducer to find the roots and patterns on the lower side and transform them into intersected stems. This algorithm employed the composition operation, applying rule-like transductions to the lower side so that the upper side would not be affected.

```

Upper:      prefixes  krb  CaCaC  suffixes
Lower:      prefixes      katab  suffixes

```

Alternation rules were then applied in the usual way, at compile time, to the lower side of this lexicon transducer to yield the final lexical transducer, which maps

directly from surface strings to analysis strings that separate and identify the morphemes, including the root and pattern. The runtime performance, i.e. the processing of real input, then reached hundreds of words per second.

The lower-side stem-intersection algorithm worked perfectly well, but it was rather slow.⁷ This stem intersection is now computed equivalently, using **compile-replace** and a new finite-state operation called **merge**, which is much faster than general intersection and is sufficient to handle the special-case intersection required to interdigitate Semitic stems.

Stem Interdigitation using compile-replace

Three-Way Solution using Merge Intersection and the special-case **merge** are n-ary operations and so can take any number of arguments. For Semitic stems, this means that you can postulate either a two-way or a three-way division of the stem, according to your taste. Either way, the stem is formed by intersection/merge. We will start with a three-way example.

The **merge** algorithm is implemented in two regular-expression operators: **.m>** for merge-to-the-right and **.<m** for merge-to-the-left. Having both operators is perhaps formally unnecessary, but it allows a bit more freedom in the way that you set up the delimited substrings before invoking **compile-replace**. The **merge** operation is intended especially for cases, as in Semitic languages, where consonant radicals and vowels are merged into a template.

For this illustration, we start by defining a network that, when compiled, contains on the lower side delimited substrings of the form

$$\hat{[\{ktb\} .m> . \{CVCVC\} .<m . [a+] \hat{]}$$

where *ktb* represents the root, *CVCVC* represents the template, and *[a+]* represents the vocalization. $\hat{[}$ and $\hat{]}$ must be multicharacter symbols, and as far as the network is concerned, everything between these two special boundary symbols is just a string of symbols. What is vital for subsequent application of **compile-replace** is that these delimited substrings have the format of a valid regular expression.

The rightward merge operator **.m>** indicates a merging of the root radicals, being **k**, **t**, and **b** in this example, into the template **CVCVC**, where they fill the **C** slots to form **kVtVb**. Then the vowel expression is merged leftward into the template to get *katab*. The **merge** operation allows the definition and use of super-symbols, e.g. defining **C** to cover the set of all possible consonant radicals and **V** to cover the set of all vowels. If the possible values of **C** and **V** are properly specified, then $\{ktb\} .m> . \{CVCVC\} .<m . [a+]$ is a valid regular expression, and it can

⁷It should be understood that this slowness was faced only once, at compile time, not at runtime. However, the slow compile time was something of a hindrance during development.

be computed by **compile-replace** or by the normal **read regex** command of **xfst**.

```
xfst[0]: list C  b t y k l m n f w r z d s
xfst[0]: list V  a i u
xfst[0]: read regex {ktb} .m>. {CVCVC} .<m. [a+] ;
xfst[1]: print words
katab
xfst[1]:
```

Super-symbols like **C** and **V** are meaningful only to the **merge** operation and are defined using the **list** command. The command **list** is followed by an enumeration of symbols separated by spaces, and all the symbols must appear on the same line. Note that the symbols following the **list** command are not a regular expression, and so they are not terminated with a semicolon as in a call to **read regex** or **define**. After the following **list** specifications, the regular expression is compiled, and the resulting network accepts only one string: *katab*.

As in the Malay example, the initial network is defined using either **lexc** or regular expressions, and the challenge is to use concatenation to build delimited substrings that look like

$$\wedge [\{ktb\} .m> . \{CVCVC\} .<m. [a+] \wedge]$$

$$\wedge [\{drs\} .m> . \{CVVCVC\} .<m. [u*i] \wedge]$$

etc. on the lower side. Figure 9.13 shows one way, of many, to build a small example in **lexc**. Figure 9.14 shows an equivalent grammar written as an **xfst** script.

If either of these grammars is compiled and stored as *lex.fst*, we can then

```

Multichar_Symbols ^[ ^] +3P +Masc +Fem
+Sg +Act +Pass +FormI +FormIII

LEXICON Root
      LBound ;

LEXICON LBound
[:^[{  Roots ;

LEXICON Roots
ktb      MergeRight ;
drs      MergeRight ;

LEXICON MergeRight
0:}.m%>.{  Template ;    ! % literalizes the >

LEXICON Template
+FormI:CVCVC  MergeLeft ;
+FormIII:CVVCVC MergeLeft ;

LEXICON MergeLeft
0:}.%<m.  Vocalization ; ! % literalizes the <

LEXICON Vocalization
+Act:[a+]  RBound ;
+Pass:[u*i] RBound ;

LEXICON RBound
]:^] PerfEnd ;

LEXICON PerfEnd
+3P+Masc+Sg:a # ;
+3P+Fem+Sg:at # ;

```

Figure 9.13: A *lexc* grammar of a fragment of Arabic verbs showing a three-way division of stems into root, template and vocalization.

```

clear stack
define rootlanguage {ktb} | {drs} ;
define tpltlanguage "+FormI":{CVCVC}
                    | "+FormIII":{CVVCVC} ;
define vocalization "+Pass":"[u*i]"
                    | "+Act":"[a+]" ;
define perfsuffixes "+3P":0 [ [ "+Masc" "+Sg" .x. a ]
                             | [ "+Fem" "+Sg" .x. a t ]
                             ] ;

read regex "[":"^["
           0:"{" rootlanguage 0:"}"
           0:".m>."
           0:"{" tpltlanguage 0:"}"
           0:".<m."
           vocalization
           "]" : "^]"
           perfsuffixes ;

```

Figure 9.14: An **xfst** script grammar of a fragment of Arabic verbs showing a three-way division of stems into root, template and vocalization.

load it onto an **xfst** stack and examine the lower-side strings.

```
xfst[0]: load lex.fst
xfst[1]: print lower-words
^[{ktb}.m>.{CVCVC}.<m.[a+]^]a
^[{ktb}.m>.{CVCVC}.<m.[a+]^]at
^[{ktb}.m>.{CVCVC}.<m.[u*i]^]a
^[{ktb}.m>.{CVCVC}.<m.[u*i]^]at
^[{ktb}.m>.{CVVCVC}.<m.[a+]^]a
^[{ktb}.m>.{CVVCVC}.<m.[a+]^]at
^[{ktb}.m>.{CVVCVC}.<m.[u*i]^]a
^[{ktb}.m>.{CVVCVC}.<m.[u*i]^]at
^[{drs}.m>.{CVCVC}.<m.[a+]^]a
^[{drs}.m>.{CVCVC}.<m.[a+]^]at
^[{drs}.m>.{CVCVC}.<m.[u*i]^]a
^[{drs}.m>.{CVCVC}.<m.[u*i]^]at
^[{drs}.m>.{CVVCVC}.<m.[a+]^]a
^[{drs}.m>.{CVVCVC}.<m.[a+]^]at
^[{drs}.m>.{CVVCVC}.<m.[u*i]^]a
^[{drs}.m>.{CVVCVC}.<m.[u*i]^]at
```

For a larger grammar, **print lower-words** will be impractical, and **print random-lower** would be more appropriate. Note that these lower-side strings do indeed contain substrings of symbols between `^[` and `^]` that look like regular expressions. However, we admit that we didn't get it right the first time, and you too will typically have to go through a few rounds of checking and editing before the delimited substrings are right.

In this case, the grammar has been written to show roots and tags on the upper side, with brackets around the parts representing the stem.

```
xfst[1]: print upper-words
[ktb+FormI+Act]+3P+Masc+Sg
[ktb+FormI+Act]+3P+Fem+Sg
[ktb+FormI+Pass]+3P+Masc+Sg
[ktb+FormI+Pass]+3P+Fem+Sg
[ktb+FormIII+Act]+3P+Masc+Sg
[ktb+FormIII+Act]+3P+Fem+Sg
[ktb+FormIII+Pass]+3P+Masc+Sg
[ktb+FormIII+Pass]+3P+Fem+Sg
[drs+FormI+Act]+3P+Masc+Sg
[drs+FormI+Act]+3P+Fem+Sg
[drs+FormI+Pass]+3P+Masc+Sg
[drs+FormI+Pass]+3P+Fem+Sg
[drs+FormIII+Act]+3P+Masc+Sg
[drs+FormIII+Act]+3P+Fem+Sg
[drs+FormIII+Pass]+3P+Masc+Sg
[drs+FormIII+Pass]+3P+Fem+Sg
```

Of course, the upper-side strings could be designed in any number of ways, according to the tastes and needs of the users.

If we then apply **compile-replace lower** to this network, remembering to use **list** to specify the symbols covered by **C** and **V**, the result is the following:

```

xfst[1]: list C  b t y k l m n f w r z d s
xfst[1]: list V  a i u
xfst[1]: compile-replace lower
8 regular expressions compiled successfully. No errors.
1.2 Kb. 31 states, 38 arcs, 16 paths.
xfst[1]: print lower-words
kataba
katabat
kaataba
kaatabat
kutiba
kutibat
kuutiba
kuutibat
darasa
darasat
daarasa
daarasat
durisa
durisat
duurisa
duurisat
xfst[1]: apply up kataba
[ktb+FormI+Act]+3P+Masc+Sg
xfst[1]:

```

As can be seen, the resulting system analyzes surface words, returning analyses that separate and identify the root, template and vocalization.

Two-Way Solution using Merge A two-way root-and-pattern analysis can be modeled and computed in much the same way. Let's assume here that the user wants to see the actual root and pattern symbols on the upper side. The grammar in Figure 9.15 arbitrarily uses a literal hyphen symbol on the upper side as a visual separator of the root and pattern symbols.

Note in this example that the patterns effectively combine the template and the vocalization of the three-way theory. Compiled in **lexc**, saved, and then loaded onto an **xfst** stack, we first verify the lower-side words before trying to call **compile-**

```

Multichar_Symbols ^[ ^] +3P +Masc +Fem
+Sg +Act +Pass +FormI +FormIII

LEXICON Root
      LBound ;

LEXICON LBound
[:^]{  Roots ;

LEXICON Roots
ktb      MergeRight ;
drs      MergeRight ;

LEXICON MergeRight
-:}.m%>.{  Pattern ;

LEXICON Pattern
CaCaC    RBound ;
CaaCaC   RBound ;
CuCiC    RBound ;
CuuCiC   RBound ;

LEXICON RBound
]:}^] PerfEnd ;

LEXICON PerfEnd
+3P+Masc+Sg:a # ;
+3P+Fem+Sg:at # ;

```

Figure 9.15: A **lexc** grammar encoding a fragment of Arabic verbs using a two-way root and pattern analysis of stems.

replace.

```

xfst[0]: load lex2.fst
xfst[1]: print lower-words
^[\{ktb\}.m>.{CaCaC}^]a
^[\{ktb\}.m>.{CaCaC}^]at
^[\{ktb\}.m>.{CaaCaC}^]a
^[\{ktb\}.m>.{CaaCaC}^]at
^[\{ktb\}.m>.{CuCiC}^]a
^[\{ktb\}.m>.{CuCiC}^]at
^[\{ktb\}.m>.{CuuCiC}^]a
^[\{ktb\}.m>.{CuuCiC}^]at
^[\{drs\}.m>.{CaCaC}^]a
^[\{drs\}.m>.{CaCaC}^]at
^[\{drs\}.m>.{CaaCaC}^]a
^[\{drs\}.m>.{CaaCaC}^]at
^[\{drs\}.m>.{CuCiC}^]a
^[\{drs\}.m>.{CuCiC}^]at
^[\{drs\}.m>.{CuuCiC}^]a
^[\{drs\}.m>.{CuuCiC}^]at
xfst[1]:

```

In this example, the upper-side strings show a root and pattern, separated by a hyphen.

```

xfst[1]: print upper-words
[ktb-CaCaC]+3P+Masc+Sg
[ktb-CaCaC]+3P+Fem+Sg
[ktb-CaaCaC]+3P+Masc+Sg
[ktb-CaaCaC]+3P+Fem+Sg
[ktb-CuCiC]+3P+Masc+Sg
[ktb-CuCiC]+3P+Fem+Sg
[ktb-CuuCiC]+3P+Masc+Sg
[ktb-CuuCiC]+3P+Fem+Sg
[drs-CaCaC]+3P+Masc+Sg
[drs-CaCaC]+3P+Fem+Sg
[drs-CaaCaC]+3P+Masc+Sg
[drs-CaaCaC]+3P+Fem+Sg
[drs-CuCiC]+3P+Masc+Sg
[drs-CuCiC]+3P+Fem+Sg
[drs-CuuCiC]+3P+Masc+Sg
[drs-CuuCiC]+3P+Fem+Sg
xfst[1]:

```

After specifying the coverage of **C** and **V** using **list** commands, and calling **compile-replace**, the lower side is exactly the same as in the three-way example

above.

```

xfst[1]: list C   b t y k l m n f w r z d s
xfst[1]: list V   a i u
xfst[1]: compile-replace lower
8 regular expressions compiled successfully. No errors.
1.4 Kb. 40 states, 47 arcs, 16 paths.
xfst[1]: print lower-words
kataba
katabat
kaataba
kaatabat
kutiba
kutibat
kuutiba
kuutibat
darasa
darasat
daarasa
daarasat
durisa
durisat
duurisa
duurisat
xfst[1]: apply up kataba
[ktb-CaCaC]+3P+Masc+Sg
xfst[1]: apply up kuutiba
[ktb-CuuCiC]+3P+Masc+Sg
xfst[1]:

```

This example illustrates in simplified microcosm the way that the full-sized **Xerox** Arabic morphological analyzer is implemented. The call to **compile-replace** that intersects over 700,000 root-pattern combinations on the lower side takes ten to twenty minutes on a SUN Ultra workstation.⁸

Three-Way Solution using General-Purpose Intersection The **merge** operation efficiently implements the special case of intersection that involves plugging consonants and vowels into template slots. But, as illustrated by Beesley⁹, stems can be constructed using general-purpose intersection. This approach is not recommended, and has never been used by us in practice, because general-purpose intersection is relatively slow; but we offer the following examples just to show that it is possible.

As always, we first define an initial lexicon transducer that contains delimited regular-expression substrings that indicate the literal intersection of a root lan-

⁸The speed depends mainly on the amount of RAM available.

⁹(Beesley, 1998b)

guage, a template language, and a vocalization language. For variety, we build the initial transducer in this case using an **xfst** script (see Figure 9.16) rather than **lexc**.

```

clear stack
define rootlanguage {ktb} | {drs} ;
define templatelanguage "C V C V C" | "C V V C V C" ;
define vocalization "[u*i]" | "[a+]" ;
define perfsuffixes "+3P":0
                [ [ "+Masc" "+Sg" .x. a ]
                  | [ "+Fem"  "+Sg" .x. a t ]
                  ] ;
read regex "[":"^["
           0:"{" rootlanguage 0:"}" 0:"/" 0:"?"
           "-":"&"
           templatelanguage
           "-":"&"
           vocalization 0:"/" 0:"\\" 0:"V"
           "]" ":" ^ "]"
           perfsuffixes ;
define C k | t | b | d | r | s | z | m | n | q ;
define V a | i | u ;

```

Figure 9.16: Arabic stem interdigitation using **compile-replace** and the general-purpose intersection algorithm. This approach is possible but not recommended.

Once this script is run, the initial network on the top of the stack contains the

following lower-side strings:

```
xfst[1]: print lower-words
^[{drs}/?&C V V C V C&[a+]/\V^]at
^[{drs}/?&C V V C V C&[a+]/\V^]a
^[{drs}/?&C V V C V C&[u*i]/\V^]at
^[{drs}/?&C V V C V C&[u*i]/\V^]a
^[{drs}/?&C V C V C&[a+]/\V^]at
^[{drs}/?&C V C V C&[a+]/\V^]a
^[{drs}/?&C V C V C&[u*i]/\V^]at
^[{drs}/?&C V C V C&[u*i]/\V^]a
^[{ktb}/?&C V V C V C&[a+]/\V^]at
^[{ktb}/?&C V V C V C&[a+]/\V^]a
^[{ktb}/?&C V V C V C&[u*i]/\V^]at
^[{ktb}/?&C V V C V C&[u*i]/\V^]a
^[{ktb}/?&C V C V C&[a+]/\V^]at
^[{ktb}/?&C V C V C&[a+]/\V^]a
^[{ktb}/?&C V C V C&[u*i]/\V^]at
^[{ktb}/?&C V C V C&[u*i]/\V^]a
```

This script, which is admittedly tricky to write, defines stems as the intersection of three separate languages. Taking the example of

```
^[{drs}/?&C V V C V C&[u*i]/\V^]at
```

the root we normally think of as *drs* is encoded as $\{drs\}/?$, a regular expression that denotes the language of all strings that contain a **d**, an **r** and an **s**, in that left-to-right order, ignoring any characters that might appear around or between them; this language is the finite-state approximation of all possible strings that contain the root **drs**. The second language is encoded as $C V V C V C$, which denotes a concatenation of **C**s and **V**s, where **C** and **V** are defined subsequently in the script as variables ranging over consonants and vowels, respectively. Finally, the vocalization is modeled as $[u*i]/\V$ (Beesley, 1998c), which denotes the language of all strings that contain zero or more **us**, followed by a single **i**, ignoring any other non-vowel symbols that might appear around or between them. (The notation \V denotes the language of all single-symbol strings except for the strings denoted by **V**, which are vowels.) Thus each of the three parts of the delimited substring

1. $\{drs\}/?$
2. $C V V C V C$
3. $[u*i]/\V$

denotes a language and therefore a constraint. The intersection of these three languages is the language containing the single string *duuris*; this stem is the only string that satisfies all the constraints.

The delimited strings are all resolved, at compile time, by calling **compile-replace**.

```
xfst[1]: compile-replace lower
```

The algorithm finds each substring delimited by `^[` and `^]`, compiles it into a network, and then replaces the original substring path on the lower side of the transducer with the result of the compilation. The resulting lower-side strings are exactly the same as in the previous examples.

```
xfst[1]: print lower-words
daarasat
daarasa
darasat
darasa
duuriset
duurisa
duriset
durisa
kaatabat
kaataba
katabat
kataba
kuutibat
kuutiba
kutibat
kutiba
```

There are many variations of the stem-intersection script that will work, but the coding is tricky, and even the slightest coding error can prevent **compile-replace** from working at all.¹⁰ The conceptual difficulty is that the user must write regular expressions and scripts such that the resulting network will contain delimited substrings that themselves look like valid regular expressions. The script in Figure 9.17 is a variation that applies some rules to the lower side of the network before invoking **compile-replace lower**.

Construction of Arabic stems via general-purpose intersection is therefore possible but not recommended; we at **Xerox** have never done it this way because the general-purpose intersection algorithm is relatively slow. The **merge** operation implements a special case of intersection which is far more efficient and is perfectly adequate for handling the Arabic data.

Returning Stems Instead of Roots We have assumed up to now that the ultimate user wishes to see Semitic roots and patterns, or perhaps roots, templates and vocalizations, returned and identified somehow in the analysis strings. But in some

¹⁰As biologist Richard Dawkins states in *The Blind Watchmaker*, “However many ways there may be of being alive it is certain that there are more ways of being dead.”

```

clear stack
define rootlanguage {ktb} | {drs} ;
define templatelanguage {CVCVC} | {CVVCVC} ;
define vocalization "[u*i]" | "[a+]" ;
define perfsuffixes "+3P":0
    [ "+Masc" "+Sg" .x. a ]
    | [ "+Fem" "+Sg" .x. a t ] ;
read regex [ "[:]"^["
    0:"{" rootlanguage 0:"}" 0:"/?"
    "-":"&"
    templatelanguage
    "-":"&"
    vocalization 0:"/" 0:"\\" 0:"V"
    "]"^"]"
    perfsuffixes
    ]
    .o.
C -> Cons " "
    .o.
V -> Vow " " ;
define Cons k | t | b | d | r | s | z | m | n | q ;
define Vow a | i | u ;
compile-replace lower

```

Figure 9.17: A variation of Arabic stem interdigitation using **compile-replace** and the general-purpose intersection algorithm. This script illustrates that the network can be modified, e.g. by the composition of rules, before **compile-replace** is invoked.

applications, it might be valuable to build a variant of the analyzer that returns interdigitated stems, collapsing the underlying roots and patterns. All that would be necessary to perform this trick is to apply the **compile-replace** algorithm to the upper side as well as to the lower side of the initial network. This requires that the network have properly-delimited regular-expression substrings on both sides, as produced by this **xfst** script:

```
clear stack
define rootlanguage {ktb} | {drs} ;
define tpltlanguage {CVCVC} | {CVVCVC} ;
define vowel a | i | u ;
define vocalization "[u*i]" | "[a+]" ;
define perfsuffixes "+3P":0 [ "+Masc" "+Sing" .x. a ]
                        | [ "+Fem" "+Sg" .x. a t ] ;
read regex "^[" "{ " rootlanguage "}" ".m>."
           "{ " tpltlanguage "}" ".<m."
           vocalization
           "]" perfsuffixes ;
list C k t b d r s
list V a i u
```

Upper	<code>^[{drs}.m>.{CVCVC}.<m.[u*i]^]+3P+Masc+Sing</code>
Lower	<code>^[{drs}.m>.{CVCVC}.<m.[u*i]^]a</code>

Table 9.6: In order to apply **compile-replace** to both sides of a network, both sides must have properly defined and delimited regular-expression substrings.

Before application of **compile-replace**, the resulting network has two-level paths that look like those in Table 9.6. After applying **compile-replace** to both sides,

```
xfst[1]: compile-replace lower
xfst[1]: compile-replace upper
```

the same path will appear as in Table 9.7, with intersected stems on both sides.

9.4 Usage Notes for **compile-replace**

9.4.1 The Non-Concatenative Challenge

Non-concatenative morphotactics is inherently a challenge, no matter how it is handled. This is the cutting edge of morphological theory, and it is not surprising that **compile-replace** is a challenge to understand and master. It should be

Upper	duuris+3P+Masc+Sing
Lower	duurisa

Table 9.7: After applying **compile-replace** on both sides, the resulting transducer will return analysis strings that identify only the collapsed stem, not the component root, pattern and vocalization.

attempted after the developer is already comfortable with the rest of the finite-state techniques.

9.4.2 Morphotactic Abstraction

The ability to abstract away from surfacy phenomena is often the key to successful finite-state development. In the case of non-concatenative morphotactics, the developer must avoid trying to describe surfacy strings directly in favor of meta-descriptions that employ finite-state operations like iteration and intersection (merge). Other finite-state operations may prove useful in the future.

9.4.3 Upper or Lower Application

The **compile-replace** algorithm comes in two varieties, **compile-replace lower** and **compile-replace upper**. Thus **compile-replace** can be applied to either the upper side or the lower side of a transducer, or both, as shown in Table 9.6 and 9.7 on page 518.

9.4.4 Multiple Application

In the examples shown, the regular-expression substrings, including the delimiters $\hat{[}$ and $\hat{]}$ were built directly by the original **lexc** descriptions or regular expressions that describe the lexicon. However, there is nothing to prevent the appropriate delimiters from being introduced or reintroduced into a network in other ways, e.g. by substitution or transduction, and **compile-replace** could be invoked multiple times on the same network.

9.4.5 The No-Retokenize Option

Two Modes of Operation

There were two points of view, represented separately by the two authors, about how **compile-replace** should work, and after sometimes warm debate, two modes of operation are now possible. The default mode, which has been assumed thus far, is called the RETOKENIZATION MODE. In this mode, each substring between

[and ^] is extracted as a stand-alone string and then parsed as a regular expression, including retokenization according to the usual rules of regular-expression compilation. Thus each extracted substring is handled as if it were written in a text file and parsed by a call to **read regex**.

The alternative is the `PRE-TOKENIZED MODE`, wherein the delimited substrings are extracted and parsed, but the regular-expression parser respects and depends on the symbol tokenization established in the original network. In this mode, the **compile-replace** parser does not retokenize the regular-expression substring.

The selection between the two modes is controlled by an `xfst` variable appropriately named **retokenize**. By default, **retokenize** is **ON**. To turn it off, enter

```
xfst[0]: set retokenize OFF
```

An Example with `retokenize=OFF`

Experience has shown that it is a challenge to create a network that can be successfully processed by **compile-replace**; writing regular expressions (or **lexc** descriptions) that themselves produce regular expressions involves a level of abstraction beyond what is normally required. When the user sets the **retokenize** variable to **OFF**, the tasks of compiling and replacing become easier for the algorithm but arguably harder for the user. Not only must the delimited substrings *look like* regular expressions, but they must also be properly pre-tokenized.

Consider the following `xfst` script named `noretok.script` that correctly defines a network for **compile-replace** without retokenization. Keep in mind that the typical developer must go through several rounds of scrutiny and correction before the delimited substrings are well-formed.

```
clear stack
define rootlanguage {ktb} | {drs} ;
define tpltlanguage "+FormI":{CVCVC}
                    | "+FormIII":{CVVCVC} ;
define vocalization [ "+Pass" .x. "[" u "*" i "]" ]
                    | [ "+Act" .x. [" a "+" "]" ] ;
define perfsuffixes "+3P":0 [ [ "+Masc" "+Sg" .x. a ]
                             | [ "+Fem" "+Sg" .x. a t ]
                             ] ;
read regex [ ":" ^ [ " rootlanguage 0: ".m>."
                    tpltlanguage 0: ".<m."
                    vocalization
                    "]" : ^ ] " perfsuffixes ;
list C k t b d r s
list V a i u
```


If you run the script and then invoke **print lower-words**, you get the following output:

```
xfst[0]: source noretok.script
Opening file noretok.script...
defined rootlanguage: 248 bytes. 6 states, 6 arcs, 2 paths.
defined tpltlanguage: 300 bytes. 7 states, 7 arcs, 2 paths.
defined vocalization: 400 bytes. 8 states, 8 arcs, 2 paths.
defined perfsuffixes: 316 bytes. 5 states, 5 arcs, 2 paths.
1.0 Kb. 27 states, 30 arcs, 16 paths.
Closing file noretok.script...
xfst[1]: print lower-words
^[drs.m>.CVVCVC.<m.[u*i]^]at
^[drs.m>.CVVCVC.<m.[u*i]^]a
^[drs.m>.CVVCVC.<m.[a+]^]at
^[drs.m>.CVVCVC.<m.[a+]^]a
^[drs.m>.CVCVC.<m.[u*i]^]at
^[drs.m>.CVCVC.<m.[u*i]^]a
^[drs.m>.CVCVC.<m.[a+]^]at
^[drs.m>.CVCVC.<m.[a+]^]a
^[ktb.m>.CVVCVC.<m.[u*i]^]at
^[ktb.m>.CVVCVC.<m.[u*i]^]a
^[ktb.m>.CVVCVC.<m.[a+]^]at
^[ktb.m>.CVVCVC.<m.[a+]^]a
^[ktb.m>.CVCVC.<m.[u*i]^]at
^[ktb.m>.CVCVC.<m.[u*i]^]a
^[ktb.m>.CVCVC.<m.[a+]^]at
^[ktb.m>.CVCVC.<m.[a+]^]a
```

The delimited substrings, as displayed by **print lower-words** or **print random-lower**, do not in fact look like valid regular expressions. If, for example, you cut the string `ktb.m>.CVCVC.<m.[a+]` and try paste it to **read regex** for compilation, it will fail.

```
xfst[1]: read regex ktb.m>.CVCVC.<m.[a+] ;
304 bytes. 1 state, 0 arcs, 0 paths.
```

However, this network is in fact correctly set up for **compile-replace** in the **no-retokenize** mode because the string between `^[` and `^]` is pre-tokenized. As far as the network itself is concerned, the string displayed as `ktb.m>.CVCVC.<m.[a+]` is the concatenation of the following symbols:

```
k t b .m>. C V C V C .<m. [ a + ]
```

and that spaced string, if fed to **read regex**, is successfully parsed.

```

xfst[1]: read regex k t b .m>. C V C V C .<m. [ a + ] ;
324 bytes. 6 states, 5 arcs, 1 path.
xfst[2]: words
katab

```

To see the tokenization when you invoke the **print** commands, you need to set the **xfst** interface variable **print-space** to **ON**. This causes a space to be printed between each symbol, showing which are considered as multicharacter symbols.

```

fst[1]: set print-space ON
variable print-space = ON
fst[1]: print lower-words
^[ d r s .m>. C V V C V C .<m. [ u * i ] ^] a t
^[ d r s .m>. C V V C V C .<m. [ u * i ] ^] a
^[ d r s .m>. C V V C V C .<m. [ a + ] ^] a t
^[ d r s .m>. C V V C V C .<m. [ a + ] ^] a
^[ d r s .m>. C V C V C .<m. [ u * i ] ^] a t
^[ d r s .m>. C V C V C .<m. [ u * i ] ^] a
^[ d r s .m>. C V C V C .<m. [ a + ] ^] a t
^[ d r s .m>. C V C V C .<m. [ a + ] ^] a
^[ k t b .m>. C V V C V C .<m. [ u * i ] ^] a t
^[ k t b .m>. C V V C V C .<m. [ u * i ] ^] a
^[ k t b .m>. C V V C V C .<m. [ a + ] ^] a t
^[ k t b .m>. C V V C V C .<m. [ a + ] ^] a
^[ k t b .m>. C V C V C .<m. [ u * i ] ^] a t
^[ k t b .m>. C V C V C .<m. [ u * i ] ^] a
^[ k t b .m>. C V C V C .<m. [ a + ] ^] a t
^[ k t b .m>. C V C V C .<m. [ a + ] ^] a

```

Note that when working in this no-retokenize mode, you must define complex regular-expression operators like **.m>.** and **.<m.** to be multicharacter symbols.

With the network defined as it is, with the symbol tokenization as shown, the user can then invoke **set retokenize OFF** and **compile-replace lower** to get the desired result.

```

xfst[1]: set retokenize OFF
variable retokenize = OFF
xfst[1]: compile-replace lower
8 regular expressions compiled successfully. No errors.
1.2 Kb. 32 states, 39 arcs, 16 paths.
fst[1]: print lower-words
d u u r i s a t
d u u r i s a
d u r i s a t

```

```

d u r i s a
d a a r a s a t
d a a r a s a
d a r a s a t
d a r a s a
k u u t i b a t
k u u t i b a
k u t i b a t
k u t i b a
k a a t a b a t
k a a t a b a
k a t a b a t
k a t a b a

```

This example shows that it is certainly possible to define networks for successful use of **compile-replace** in no-retokenize mode, but in the opinion of one author (Beesley), the job of visualizing the delimited regular-expression substrings and getting them right is rather harder. In particular, if the initial tokenization is wrong, as it often will be during development, then the default displays of **print random-lower** tend to hide the tokenization problems from you.

The main problem is that **print random-lower** by default does not print spaces between the symbols in the strings that it outputs, rather it prints connected strings to which the transducer could be successfully applied. The practical need of the developer and debugger, on the other hand, is more often to see what the paths of the network look like, including where the symbols are divided. To get more useful displays for debugging, be sure to set the variables **show-flags** and **print-space** to **ON** before invoking the **print** utilities, e.g.

```

xfst[1]: set show-flags ON
xfst[1]: set print-space ON
xfst[1]: print random-lower

```

These commands can be invoked automatically via aliases as shown in Section 8.4.2.

9.5 Debugging Tips for compile-replace

9.5.1 Properly Defining the Initial Network

Setting up a network prior to the application of **compile-replace** is admittedly tricky, often requiring a bit of **lexc** or regular-expression virtuosity. **compile-replace** will itself report any errors in trying to compile delimited substrings, which are supposed to be valid regular expressions. But during initial development you can almost be assured that you have made systematic errors, and it's usually best to examine the network carefully and fix as many as possible before trying to apply **compile-replace**.

Default Retokenize Mode

If you are using **compile-replace** in the default retokenize mode, the strings between `^[` and `^]` should look like valid regular expressions when displayed by the default **print lower-words** and **print random-lower** commands. Assuming that the defined variable **initialnet** holds the initial transducer and that **compile-replace** is to be applied to the lower side, the developer is advised to invoke

```
xfst[0]: read regex initialnet .o. $["^[ " ] ;
```

to limit the network to the subset having delimited regular-expression substrings on the lower side, then

```
xfst[1]: print random-lower
```

multiple times and examine closely the substrings that appear between the delimiters `^[` and `^]`. They should *look like* regular expressions, and if you copy any delimited string from the display and paste it as the input to **read regex**, then it should compile successfully into the desired result. Fix your source grammars as necessary and recompile the **initialnet** until the lower-side delimited substrings look right—and then invoke **compile-replace**.

Alternative No-Retokenize Mode

If you have explicitly set the **xfst retokenize** variable to **OFF**,

```
xfst[1]: set retokenize OFF
```

then it is your responsibility to ensure that the delimited substrings are both in the format of regular expressions and are properly pre-tokenized. To see how the strings are being tokenized, set the **print-spaces** variable to **ON** before invoking the various **print** commands.

```
xfst[1]: set print-space ON
xfst[1]: print random-lower
```

In this mode, the spaced out strings displayed by **print lower-words** and **print random-lower** should look like and be compilable as regular expressions.

9.5.2 Flag Diacritics

Flag Diacritics and compile-replace

As shown above, the **compile-replace** algorithm takes strings that look like

```
^[ [ {buku}%^REHYPH]^2^ ]
```

or

```
^{\{ktb\}.m>.{CVVCVC}.<m.[u*i]^}at
```

and compiles the expressions that appear between the `^` and `^` delimiters. If Flag Diacritics occur inside the delimiters, then **compile-replace** first extracts them, retaining the original order, and replaces them at the end of the replaced string. In most cases, therefore, developers do not need to worry about the presence of Flag Diacritics when invoking **compile-replace**.

If you do feel a need to see exactly where the Flag Diacritics are, set the **xfst** interface variables **show-flags** and **print-space** to **ON**.

```
xfst[1]: set show-flags ON
xfst[1]: set print-space ON
```

With these settings, the various **print** commands will always show you the Flag Diacritics and will always print a space between symbols, showing you clearly which sequences are being treated as multicharacter symbols.

9.5.3 Size Problems

The application of **compile-replace** to one side of a network will usually introduce significant discrepancies between the upper and lower strings. These discrepancies translate directly into more states and arcs because the network is not able to share structure as efficiently as before. The application of **compile-replace** therefore tends to make a network significantly larger.

During the application of **compile-replace**, the algorithm must handle many intermediate results before minimization can be invoked, and this can require a lot of memory. A great deal of effort has been put into **compile-replace** to make its performance and memory usage as efficient as possible, but, as with other intensive operations like intersection and composition, it may sometimes necessary to have a great deal of RAM to make a particular computation practical.

9.6 The Formal Power of Morphotactics

9.6.1 The Formal Power of Full-Stem Reduplication

It is well known that the formal language containing all reduplicated strings $\alpha\alpha$, where the first half and second half of each word are identical, cannot be described by finite-state or even context-free formalisms. This $\alpha\alpha$ language is in fact context-sensitive in power, and the corollary is that this language cannot be encoded in a finite-state network.

It might be assumed from this mathematical result that it is impossible to handle Malay reduplication in a finite-state system. However, the proper task in writing a morphological analyzer for Malay is not to recognize the $\alpha\alpha$ language, but rather

to recognize all and only valid Malay reduplications. We don't want to accept just any string from the infinite $\alpha\alpha$ language, but only the finite subset of $\alpha\alpha$ strings wherein α is a valid Malay stem and where α is known to be the kind of Malay stem that reduplicates. To put things even more in perspective, we don't even want to accept the plain unreduplicated stem α for any language unless it is a valid stem in that language. As always, the set of valid stems for a natural language is finite and must be specified in a dictionary.

So although we cannot, mathematically, build a finite-state transducer that recognizes and analyzes the $\alpha\alpha$ language, we can, using **compile-replace**, build a finite-state transducer that handles Malay reduplication over some finite set of valid Malay stems.

9.6.2 Analyzing vs. Guessing Reduplications

Section 10.5.4 shows how to write finite-state morphological guessers that can analyze words based on guessed roots that are not enumerated in a lexicon. The trick, in such morphological guessers, is to define a language of phonologically possible roots using regular expressions, and then to build that language into a network in place of the usual language of enumerated roots. When such a guesser returns an analysis, based on the guessed root R, it is basically saying "this is a solution, if and only if R is a valid root with these affixes". The guesser can be a valuable back-up to the normal morphological analyzer, filling the gaps until the wide-coverage dictionary of roots is built, and actively suggesting roots that might need to be added to that dictionary.

It is interesting to note that building such a guesser for a language with non-concatenative phenomena is usually impractical or impossible. If the language in question uses full-stem reduplication for noun plurals, and the language of guessable roots is not constrained in its length, then the task of recognizing any possible plural becomes once again the recognition of the $\alpha\alpha$ language, which is known to be beyond finite-state power.

9.7 Conclusion

Non-concatenative morphotactics is very much a hot issue in descriptive phonology and morphology, and it is probably the main challenge in current computational morphology. We are well aware that some languages illustrate more complex kinds of reduplication than we have tried to handle so far, and so there remains much interesting work to be done. In addition to our own experiments with Arabic and Malay, we are aware of current work being done with **compile-replace** on Hebrew and Amharic, which are Semitic languages with the same challenges as Arabic, and on Bantu languages, which exhibit reduplication. We expect some growth and modification in our finite-state algorithms, and we look forward to reports and feedback from these and other developers working on the cutting edge.

Chapter 10

Finite-State Linguistic Applications

Contents

10.1 Introduction	528
10.2 The <code>tokenize</code> Utility	529
10.2.1 Tokenization is Non-Trivial	529
10.2.2 Using the <code>tokenize</code> Utility	530
Input to <code>tokenize</code>	530
Printing a Help Message	530
Displaying the Version Number	531
Invoking <code>tokenize</code>	531
Invoking <code>tokenize</code> in a Pipe	531
Examples	532
10.2.3 Transducers for Finite-State Tokenization	532
Example 1: Separate on Whitespace and Punctuation	533
Example 2: Naive Handling of Multi-Word Tokens	535
Example 3: Better Handling of Multi-Word Tokens	537
10.3 The <code>lookup</code> Utility	539
10.3.1 Introduction to <code>lookup</code>	539
10.3.2 Input to <code>lookup</code>	539
10.3.3 Documentation	539
10.3.4 <code>lookup</code> with a Single FST	539
10.3.5 Lookup Strategy Scripts	540
Specifying a <code>lookup</code> Strategy Script	540
Lookup Scripts and Virtual Composition	540
Lookup Strategy	541
10.3.6 Output of <code>lookup</code>	542

10.3.7	Multicharacter Symbols and lookup	545
10.4	Using xfst in Batch Mode	546
10.5	Transducers for Morphological Analysis	547
10.5.1	Monolithic Lexical Transducers	547
10.5.2	Normalizers	547
	Initial-Capitalization Normalization	547
	All-Capitalization Normalization	549
	Encoding Normalization	550
10.5.3	Relaxation	551
	Relaxation of Accentuation Rules	551
	General Relaxation of Spelling Rules	551
10.5.4	Guessers	552
	Guessers vs. Normal Morphological Analyzers	552
	Constructing a Simple Morphological Guesser	552
	A Better Definition of Possible Stems	555
10.6	Spelling	557
10.6.1	Spelling Checkers	557
10.6.2	Spelling Correctors	557
10.7	Beyond Tokenization and Morphological Analysis	559
10.7.1	Other Uses for tokenize	559
10.7.2	Advanced Tokenization and Normalization	559
10.7.3	Part-of-Speech Disambiguation	559
10.7.4	Parsing and Higher Linguistic Processing	561
10.8	Application Summary	561

10.1 Introduction

This book has concentrated on morphological analysis, but in this chapter we will show some related linguistic applications that can be built using the finite-state tools and the command-line utilities **tokenize** and **lookup**, introduced briefly in Chapter 7. The applications include tokenization, normalization, morphological guessing, spelling checking and correction, and eventually part-of-speech disambiguation and some kinds of shallow syntactic parsing.

The **xfst**, **lexc** and **twolc** languages described in this book are DEVELOPMENT TOOLS intended for constructing finite-state networks. In order to integrate these finite-state networks, practical software applications must include RUNTIME CODE that knows how to access and apply them.

Xerox and its commercial partners have developed a number of runtime packages, and two simple runtime utilities, called **tokenize** and **lookup**, have been

included in the software supplied with this book to facilitate testing and the integration of transducers in non-commercial systems.¹ On-line documentation for **tokenize** and **lookup** is available via <http://www.fsmbook.com/>, and this page will be updated as necessary to include up-to-date information on available runtime code and licensing procedures.

tokenize and **lookup** are command-line utilities that access and apply finite-state transducers built using **lexc**, **xfst** and **twolc**. Do not confuse the command-line **lookup** utility with the **lookup** command available inside the **lexc** interface.

10.2 The **tokenize** Utility

10.2.1 Tokenization is Non-Trivial

Tokenization is the process of dividing a running text into individual word-like chunks known as **TOKENS**. In Section 7.2.2 we illustrated some fairly simple tokenizers built using Perl and off-the-shelf Unix utilities; they did little more than map whitespace and punctuation into newline characters, creating tokenized output files, i.e. files that contain exactly one token per line. The **tokenize** utility is tool that maps a string of running text into a tokenized string using a finite-state transducer.

But before getting into **tokenize**, it's important to realize that tokenization is a non-trivial task. In real life, tokenization is seldom as simple as dividing a text at the spaces and punctuation marks. For one thing, **MULTI-WORD TOKENS** like the classics *to and fro* and *far and away* contain spaces and yet should be kept together as single tokens; the meaning of these tokens is not a syntactic function of the individual orthographical words, which sometimes, like the archaic word *fro*, cannot even appear outside the fixed construction. Other possible multi-word tokens are shown in Table 10.1. In practical systems, a token is any string that should be kept together for purposes of morphological analysis. In some systems, multi-word tokens may include dates, times, and the proper names of people, countries, companies, etc.

Automatic separation of punctuation from letters is not always appropriate either, as evidenced in abbreviations such as *etc.*, *et al.*, *Mr.*, *Mrs.*, *R.S.V.P.* and in filenames and URLs such as *myfile.txt* and *http://www.arabic-morphology.com/*.

Finally, in a number of languages like Japanese, Chinese and Thai, the standard orthography does not insert spaces between words, making tokenization a very difficult challenge indeed. Here we will limit the discussion to European or-

¹Another much larger and highly sophisticated runtime package, called XeLDA, is licensed commercially by **Xerox**. XeLDA is not included with this book.

Borrowed expressions	
Latin	a priori, ad hoc, in situ, persona non grata
French	coup d'état
Prepositions	
English	in front of, on behalf of, instead of
French	le long de, à cause de, en face de
Spanish	dentro de, frente a
German	in Bezug auf
Multi-word Conjunctions	
French	parce que, afin de
Spanish	bien que, luego que
Adverbial Expressions	
English	inside out, to and fro
French	au fur et à mesure, par conséquent
German	ein bisschen, gang und gäbe
Spanish	en particular, por el contrario

Table 10.1: Some Multi-Word Tokens

thographies, but it should still be fairly obvious that intelligent tokenization is a difficult task requiring a great deal of language-specific information.

In finite-state tokenization using the **tokenize** utility, the language-specific knowledge is incorporated into a finite-state transducer that accepts as input a string of running text and outputs a string containing token-boundaries. If these token boundaries are just newline symbols, as shown in Figure 10.1, then the output can be written out as a tokenized file, i.e. as a file with one token on each line.

10.2.2 Using the **tokenize** Utility

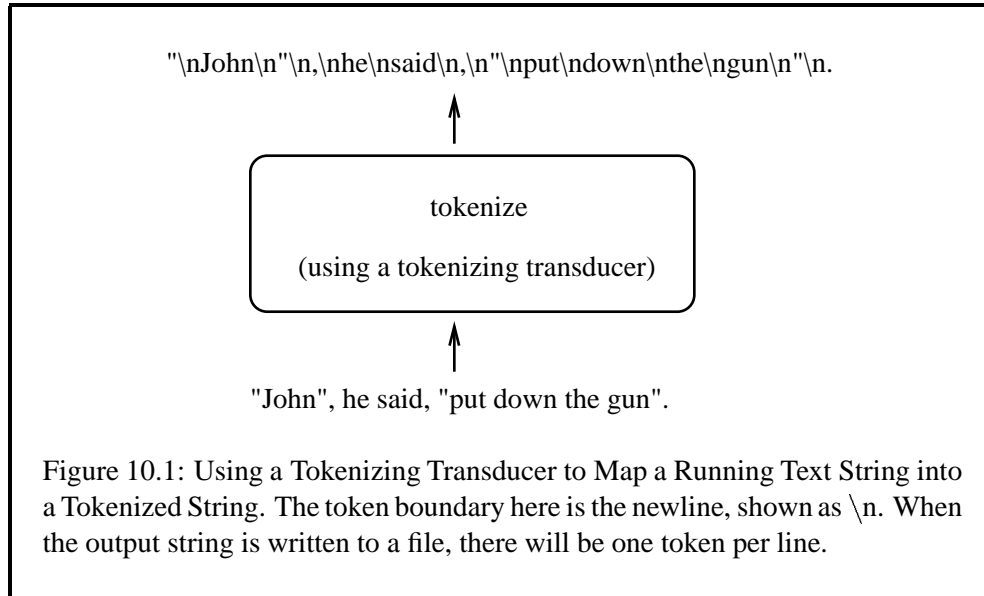
Input to **tokenize**

tokenize is a command-line utility that takes a string or file of running text as input and, using a language-specific TOKENIZING TRANSDUCER, produces as output a tokenized text, with one token per line. **tokenize** may appear in pipes, reading from the standard input and writing to the standard output. Online documentation is available,² and the most important options are presented here.

Printing a Help Message

To display a terse summary of usage and options, enter

²See the link to **tokenize** from <http://www.fsmbook.com/>.



```
unix> tokenize -h
```

Displaying the Version Number

To display the **tokenize** version number, enter

```
unix> tokenize -v
```

Invoking tokenize

In use, **tokenize** takes as its argument the pathname of a binary file containing a single transducer. File-based input can be directed to the application using the left angle bracket (<); and output can be directed to a file using the right angle bracket (>). If no output file is specified, then the output is displayed on the screen.

```
unix> tokenize binary_fst_file < input_file > output_file
```

Invoking tokenize in a Pipe

tokenize usually operates in a command-line pipe, following the following pattern,

```
... | tokenize tokenizing_fst | ...
```

where *tokenizing_fst* is the name of a file containing a pre-compiled tokenizing transducer. Technically speaking, **tokenize** always reads from the standard input and writes to the standard output.

Examples

Assuming that a suitable pre-compiled tokenizing transducer is stored in the binary file `MyLanguageTok.fst`, an **echo**-ed string of words can be piped to it, with the output printed by default to the screen.

```
unix> echo "This is a test." | tokenize MyLanguageTok.fst
This
is
a
test
.
```

The input to **tokenize** may also come from a **cat**-ed file containing a running text.

```
unix> cat corpus_file | tokenize MyLanguageTok.fst
```

tokenize output may be directed into a file, which will have one token on each line, or piped to a subsequent process like **lookup**, to be explained below. The **lookup** utility is a perfect match, expecting its input to have one token per line.³

```
unix> cat corpus_file | tokenize tokenizing_fst > \
tokenized_output_file
```

```
unix> cat corpus_file | tokenize tokenizing_fst | \
lookup analyzer_fst | ...
```

At runtime, **tokenize** loads the indicated tokenizing transducer from file and applies it in an upward (analysis) direction to all the input tokens.

10.2.3 Transducers for Finite-State Tokenization

Tokenizing transducers are defined using **xfst** regular expressions and scripts, just like any other transducer, and they allow you to incorporate language-specific information about multi-word expressions, dates, abbreviations, etc. If you aren't quite ready to get into writing your own tokenizing transducers, some of the simpler tokenization solutions shown in Section 7.2.2 may be sufficient for your earlier stages of testing.

Although **tokenize** does have some experimental provisions for non-deterministic, ambiguous tokenization, we assume here that all tokenization is deterministic and unambiguous. The transducer used by **tokenize** must therefore satisfy the following two requirements:

³The backslashes in these examples indicate to the Unix-like operating system that the command continues on the next line. These commands could also be typed on a single line, in which case no backslash should be used.

1. The lower side (here the input side) must match the universal language `?*`. In other words, the lower side must accept any input string.
2. The output must be unambiguous. For each input string, there must be a single tokenized output string.

The first requirement is easily tested by loading the transducer onto the **xfst** stack and invoking **test lower-universal**, which will return 1 for true and 0 for false.

```
xfst[]: load MyLanguageTok.fst
xfst[]: test lower-universal
```

The second requirement, that the output be unambiguous, is not easily testable. If one follows the lines of the examples given below, an unambiguous transducer will be produced.

Example 1: Separate on Whitespace and Punctuation

The following simple **xfst** script by Anne Schiller builds a tokenizing transducer for English that separates on whitespace and punctuation, and considers number expressions, initials, and abbreviations, but does not handle multi-word tokens like *to and fro*. Note the use of comments and the helpful `echo` statements that print messages to the screen, informing the user of the progress of the computation.

```
# =====
# CONTENT: Sample Finite-State Tokenizer
#           (no multi-words)
# AUTHOR: Anne Schiller
# CREATED: 12-Jun-1997
# UPDATED: 05-Sep-2001
# =====
# Usage: xfst -f [ThisFile]
# =====

clear stack

echo >>> define white space
define SP " ";
define TAB "\t";
define NL "\n";

define WS [SP|NL|TAB];

# =====
echo >>> define single character symbols
define SINGLE [ %" | %. | %, | %; | %: | %! | %?
| %( | %) | %[ | %] | %{ | %}
];
```

```

define PUNCT [ %. % . (%.) | %' %' | %' %' | %, %, ] ;

define Char \[ WS | SINGLE ] ;

# =====
echo >>> define SYMBOL
define SYMBOL [ SINGLE | PUNCT ] ;

echo >>> define WORD
define WORD [ Char ]+ ;

# =====
echo >>> list of abbreviations
define ABBR [
{Mr.} | {Mrs.} | {Ms.}
| {etc.} | {e.g.} | {i.e.}
| {ltd.} | {Ltd.} | {inc.} | {Inc.}
];

# =====
echo >>> regular abbreviations
define Letter [A|B|C|D|E|F|G|H|I|J|K|L|M|
N|O|P|Q|R|S|T|U|V|W|X|Y|Z
|a|b|c|d|e|f|g|h|i|j|k|l|m|
n|o|p|q|r|s|t|u|v|w|x|y|z
];

define INIT [ Letter %. ]+ ;

# =====
echo >>> numeric expressions
define Digit [ %0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9];
define NumOp [ %- | %+ | %* | %/ | %= | %: ];
define NumSep [ %. | %, ];

define NUM [ Digit | NumOp | NumSep]+ & ${Digit} ;

# =====
echo >>> define tokens
define Token [ WORD | SYMBOL | ABBR | INIT | NUM ];

# =====
echo >>> longest match--insert a newline after each token
define TOK1 [ Token @-> ... NL ] ;

echo >>> map spaces to a newline

```

```
define TOK2      [ [WS]+ @-> NL ];

# =====
echo >>> compose
read regex [TOK1 .o. TOK2 ];
invert net
save stack tok1.fst
```

The final tokenizing transducer must work in an upward-oriented fashion, mapping from running text on the lower side to tokenized text as shown in Figure 10.1. Note in this script that the tokenizing transducer is first built upside-down and then inverted; this is because the longest-match operator, @->, does not yet have an upward-oriented counterpart. The longest-match operator is crucial to ensure that the output is unambiguous.

Assuming that the script is in file `tok1.xfst`, the following command will run the script, which outputs the resulting transducer to the binary file `tok1.fst`, and then **xfst** will automatically exit back to the operating system.

```
unix> xfst -f tok1.xfst
unix>
```

With the tokenizing transducer compiled and stored in binary file `tok1.fst`, it can be used by the **tokenize** utility as follows,

```
unix> echo "A New Yorker lives in New York." | \
tokenize tok1.fst
```

which will produce the output

```
A
New
Yorker
lives
in
New
York
.
```

Example 2: Naive Handling of Multi-Word Tokens

Handling multi-word tokens correctly is a challenge. The following script begins in the same way as the previous one (page 533), up to the definition of `NUM`, and then includes a few multi-word tokens and handles them in a naive and ultimately inadequate way.

```
% same as Example 1 up to and including the following definition

define NUM [ Digit | NumOp | NumSep]+ & ${Digit} ;

# =====
echo >>> some multi-words
define MWE [
{a priori} | {A priori} | {ad hoc} | {Ad hoc}
| {New York} | {Hong Kong} | {Tel Aviv} | {to and fro}
];

# =====
echo >>> define tokens
define Token [ WORD | SYMBOL | ABBR | INIT | NUM | MWE ];

# =====
echo >>> longest match

define TOK1      [ Token @-> ... NL ] ;

echo >>> remove spaces
define WS1 [WS]+ & ${NL} ;
define WS2 [TAB|SP]+ ;
define TOK2      [ WS1 @-> NL ] .o. [ WS2 @-> SP ] ;

# =====
echo >>> compose
read regex [TOK1 .o. TOK2 ];
invert net
save stack tok2.fst
```

Assuming that the script is in file `tok2.xfst`, it can be run with the command

```
unix> xfst -f tok2.xfst
```

and the resulting transducer is stored in the binary file `tok2.fst`. It can then be used by the **tokenize** utility to produce the following output.

```
unix> echo "A New Yorker comes from New York." | \
tokenize tok2.fst
A
New York
er
comes
from
```


New York

.

The obvious problem with this script is that multi-word tokens like *New York* are recognized even when they are just part of a longer alphabetic string, such as *New Yorker*. This leads to the undesirable tokenization shown.

Example 3: Better Handling of Multi-Word Tokens

The following **xfst** script illustrates a better handling of multi-word tokens. It begins by inserted special brackets around maximally long multi-word expressions, but only if the expression boundaries correspond to normal word boundaries.⁴ This script, `tok3.xfst`, is the same as the first script (page 533) up to the definition of **NUM**.

```
% same as Example 1 up to and including the following definition

define NUM [ Digit | NumOp | NumSep]+ & ${Digit} ;

# =====
echo >>> some multi-words
define MWE [
  {a priori} | {A priori} | {ad hoc} | {Ad hoc}
  | {New York} | {Hong Kong} | {Tel Aviv} | {to and fro}
];

# marker for multi-words:
define M1 "<<" ;
define M2 ">>" ;

define MWE1 [M1 MWE M2];

# =====
echo >>> define tokens
define Token [ WORD | SYMBOL | ABBR | INIT | NUM | MWE1 ];

# =====
echo >>> longest match

define Bound [ SINGLE | WS | .#. ] ;

define TOK1 [
```

⁴The auxiliary special brackets around multiword expressions are deleted when they are no longer needed. Just deleting them, however, is not enough. If the auxiliary brackets remain in the sigma alphabet, the input side of the resulting transducer is not the universal language. To restore this important property, we need to explicitly “absorb” the auxiliary symbols into the unknown alphabet. This is the effect of the two **substitute** commands near the end of the script.

```

MWE @-> M1 ... M2 || Bound _ Bound
.o. Token @-> ... NL
.o. [M1|M2] -> 0
];

# =====
echo >>> normalize space
define TOK0 [ WS+ @-> SP ];

echo >>> remove spaces
define WS1 [WS]+ & ${NL} ;
define TOK2 [ WS1 @-> NL ] ;

# =====
echo >>> compose
read regex [TOK0 .o. TOK1 .o. TOK2 ];

# absorb the special brackets into the unknown alphabet
substitute symbol ? for "<<"
substitute symbol ? for ">>"

invert net
save stack tok3.fst

```

With the tokenizing transducer compiled and saved in `tok3.fst`, the output is now

```

unix> echo "A New Yorker comes from New York." | \
tokenize tok3.fst
A
New
Yorker
comes
from
New York
.

```

avoiding the breakup of *Yorker*. Of course, *New Yorker* and many other examples could now be added to the list of multi-word tokens.

What counts as adequate tokenization can vary widely, from the simple **tr**-based solutions shown in Section 7.2.2 to transducers much more complicated than we have shown above.⁵ Developers should be aware that tokenizing transducers, especially those that incorporate lots of multi-word tokens, can “blow up” in size.

⁵See (Karttunen et al., 1996) for some more sophisticated examples.

10.3 The lookup Utility

10.3.1 Introduction to lookup

The **lookup** utility is a runtime program, invoked from the command line, that applies a pre-compiled transducer or transducers to look up words. **lookup** can be used in conjunction with other programs to build prototype applications. The input to **lookup** is typically piped to it from a tokenizer, and the output is, in turn, typically piped to a subsequent program such as a disambiguator or syntactic parser.

10.3.2 Input to lookup

The **lookup** utility takes as input a tokenized file, i.e. a file that has one token per line. As shown in Section 7.2.2, such a tokenized file can be produced by standard command-line utilities such as **tr** and by languages such as Perl. The **tokenize** utility, used in the standard way, outputs a tokenized file; so the output of **tokenize** can be piped directly to the input of **lookup**.

```
unix> cat input | tokenize options | lookup options | ...
```

10.3.3 Documentation

Documentation for **lookup** can be found online,⁶ and the most important options are presented here. A summary of usage and command-line options is displayed when you enter

```
unix> lookup -h
```

To display the version number, enter

```
unix> lookup -v
```

10.3.4 lookup with a Single FST

To use **lookup** with a single transducer, pre-compiled and stored in a binary file, simply indicate the binary filename as the argument to **lookup**.

```
... | lookup binary_fst_file | ...
```

At runtime, **lookup** loads the transducer from the indicated file and applies it in an upward (analysis) direction to the input tokens. Examples of the use of **lookup** with a single transducer have already been shown in Section 7.2.2.

⁶<http://www.fsmbook.com/>

10.3.5 Lookup Strategy Scripts

Specifying a lookup Strategy Script

Instead of using a single transducer, the **lookup** utility can be controlled by a LOOKUP STRATEGY SCRIPT, stored as a plain text file, that refers to multiple transducers and specifies a lookup strategy. The name of the lookup script is written after the `-f` flag.

```
... | lookup -f lookup_script_file | ...
```

Lookup Scripts and Virtual Composition

The format of a lookup script is simple and somewhat idiosyncratic. Each script starts with a variable-declaration section wherein each line associates a user-chosen variable name with the pathname of a binary file. The user-chosen name and the name of the file are separated by whitespace. Each binary file should contain a single pre-compiled network.

```
user_chosen_name      binary_filename
```

Any number of variables can be defined, e.g.

```
deaccent              allow-deaccentuation.fst
normalize             allow-allcaps.fst
analyzer              /opt/lexicons/mylanguage.fst
```

There should be no blank lines in this declaration section, and the order of definition is not significant.

After the declaration section, separated by a blank line, comes the lookup-strategy section, which may contain one or more lookup strategies, which are either single transducers or virtual compositions of transducers to be simulated at runtime. The virtual compositions are notated in terms of the defined variables.

The complete lookup script in Figure 10.2 defines three variables and indicates that the input tokens are to be looked up using a virtual composition of the three networks. In more detail, `encoding` in the script is intended to denote a transducer that maps, in an upward direction, from strings represented in encoding X to equivalent strings in encoding Y; it might, for example, be used to map from a proprietary Microsoft encoding to a standard ISO8859 encoding. `deaccent` is intended to denote a transducer that allows deaccented words to be mapped upward to properly accented words; and `analyzer` is intended to denote a normal morphological analyzer. The lookup strategy indicates that input words should be looked up in a simulated composition equivalent to

```
[analyzer .o. deaccent .o. encoding]
```

```

encoding      encodingX2encodingY.fst
deaccent     allow-deaccent.fst
analyzer     mylanguage.fst

encoding deaccent analyzer

```

Figure 10.2: A Simple Lookup-Strategy Script. The first part of the file associates user-selected variable names with names of binary files. `encoding` here is intended to denote a transducer that maps, in an upward direction, from strings represented in encoding X to equivalent strings in encoding Y. `deaccent` is intended to denote a transducer that allows deaccented words to be mapped to properly accented words; and `analyzer` is intended to denote a normal morphological analyzer. The lookup strategy indicates that input words should be looked up in a simulated composition equivalent to `[analyzer .o. deaccent .o. encoding]`.

Note the way that virtual composition is notated in the script,

```
encoding deaccent analyzer
```

which is intended to indicate a kind of left-to-right piped processing of an input token, first through the `encoding` transducer, then through `deaccent`, and finally through `analyzer`. In each case, the transducer is applied in an upward (analysis) direction on its input.

The composition of the three transducers is simulated in the runtime code; the real computation might be very time-consuming or result in a network that is huge or even too large to compute on your machine. The **lookup** utility, using lookup-strategy scripts, can therefore help to keep your transducers from exploding in size.

Lookup Strategy

Lookup-strategy scripts can indicate multiple strategies, which are tried one-at-a-time in the specified top-to-bottom order. For example, the lookup-strategy script in Figure 10.3 sets three variables and defines three lookup strategies. The script indicates that input words are first to be looked up in `analyzer` alone. If and only if that fails, then lookup is performed using a simulation of `[analyzer .o. allcap]`. If and only if that fails, then lookup is performed using a simulation of `[analyzer .o. allcap .o. deaccent]`.

```

deaccent      allow-deaccent.fst
allcap       allcap.fst
analyzer     mylanguage.fst

analyzer
allcap analyzer
deaccent allcap analyzer

```

Figure 10.3: A Typical Lookup Strategy Script. Here input words are first looked up using `analyzer` alone, second, if that fails, using a simulation of `[analyzer .o. allcap]` and third, if that fails, using a simulation of `[analyzer .o. allcap .o. deaccent]`.

10.3.6 Output of lookup

The output of **lookup** is rather idiosyncratic and makes certain default assumptions that may not be suitable for your application. Flags are available to override the defaults and customize the output for your needs.

The transducers and simulated compositions of transducers used in **lookup** often produce multiple outputs for input tokens, reflecting the ambiguity of the tokens. The output of **lookup** contains one line for each solution, with a blank line terminating each set of solutions for one input token.

```

input_token1 TAB baseform1_1 TAB tags1_1
input_token1 TAB baseform1_2 TAB tags1_2
...
input_token1 TAB baseform1_n TAB tags1_n
<blank line>
input_token2 TAB baseform2_1 TAB tags2_1
input_token2 TAB baseform2_2 TAB tags2_2
...
input_token2 TAB baseform2_n TAB tags2_n
<blank line>
...

```

Each default output line, as shown here, contains three columns: input-token, baseform and tags, with tabs separating the columns. The defaults and flags for changing the outputs are best explained with concrete examples.

Let us assume that the tokenized input file is called `tokenized-input` and contains the following three words.

```
table
```

```
the
bills
```

Let us also assume that a morphological analyzer for English is contained in a file called `English`, and that the output is simply displayed on the screen. The basic call is then this:

```
unix> cat tokenized-input | lookup English
```

At runtime, **lookup** will load the indicated transducer from file, use it to analyze the input words, and then display the results.

By default, **lookup** echoes the input word in the first column, and separates the solution string into a baseform, printed in the second column, and the tags, printed in the third column. The columns are separated, by default, with tabs.

```
table    table    +Noun+Sg
table    table    +Verb+Pres+Non3sg

the      the      +Det+Def+SP

bills    bill     +Noun+Pl
bills    bill     +Verb+Pres+3sg
```

To split up the analysis strings into baseform (column 2) and tags (column 3), **lookup** makes the assumption, perhaps inappropriate for your morphological analyzer, that each analysis string begins with the baseform and that all the tags begin with a plus sign (+) or circumflex (^). The separation is made at the first plus sign or circumflex.

If an input token like *ciper* is not found, the default output is

```
ciper    ciper    +?
```

with `+` in the third column.

To suppress the echoing of the input word in the first column, specify `-flags x` after `lookup`. E.g. the command

```
unix> cat tokenized-input | lookup -flags x English
```

produces the output

```
table    +Noun+Sg
table    +Verb+Pres+Non3sg

the      +Det+Def+SP

bill     +Noun+Pl
bill     +Verb+Pres+3sg
```

The three columns are separated, by default, with tabs. The separator between the first and second columns can be explicitly reset to *whatever* by specifying the flag `LwhateverL`. E.g. to reset the first separator to a colon, the command

```
unix> cat tokenized-input | lookup -flags L:L English
```

produces the following output.

```
table:table      +Noun+Sg
table:table      +Verb+Pres+Non3sg

the:the          +Det+Def+SP

bills:bill       +Noun+Pl
bills:bill       +Verb+Pres+3sg
```

The second separator can be reset to *whatever* by placing it between Ts as in `TwhateverT`. E.g. to set the second separator to the empty string,

```
unix> cat tokenized-input | lookup -flags TT English
```

produces the following output, with the analysis strings kept intact.

```
table  table+Noun+Sg
table  table+Verb+Pres+Non3sg

the    the+Det+Def+SP

bills  bill+Noun+Pl
bills  bill+Verb+Pres+3sg
```

Flags can be combined. To set the first separator to a colon and the second separator to the empty string,

```
unix> cat tokenized-input | lookup -flags L:LTT English
```

produces the following output.

```
table:table+Noun+Sg
table:table+Verb+Pres+Non3sg

the:the+Det+Def+SP

bills:bill+Noun+Pl
bills:bill+Verb+Pres+3sg
```


To output just the solution strings, with no column separators at all,

```
unix> cat tokenized-input | lookup -flags xLLTT English
```

produces the following output.

```
table+Noun+Sg
table+Verb+Pres+Non3sg

the+Det+Def+SP

bill+Noun+Pl
bill+Verb+Pres+3sg
```

The **lookup** command-line interface was never intended as a polished or commercially viable tool; its assumptions and idiosyncrasies matched the needs of the original programmers. Use the **TT** flag if your analysis strings do not follow the default conventions expected by **lookup**. In particular, use **TT** if

1. Your tags do not always begin with a plus sign or circumflex
2. Your analysis strings include prefixes and/or tags that appear before the base-form, or
3. You have any other reason to divide up analysis strings differently from the default.

If you need to divide up analysis strings differently, use the **TT** flag and pipe the output of **lookup** to *yourscript*, written in Perl or your favorite language, to do whatever you need to do before piping the output to the subsequent processing step.

```
unix> cat tokenized-input | lookup -flags TT English | yourscrip | ...
```

10.3.7 Multicharacter Symbols and **lookup**

The **lookup** utility was built for speed, and the handling of multicharacter symbols is relatively expensive. By default, therefore, **lookup** assumes that the input strings are normal alphabetic words that do not contain multicharacter symbols, and it further assumes that it is safe to output the results as strings of individual alphabetic characters, without worrying about whether any of the strings of characters being output are really multicharacter symbols. These assumptions are not appropriate in some cases.

If your input strings do contain sequences that should be treated as multicharacter symbols, then you should specify the flag **m1**. The flag **m1** specifies that a parse table, capable of processing multicharacter symbols present on the *lower*

side of the transducer, should be created and used to tokenize each input word into symbols.⁷

```
unix> cat input | tokenize tok.fst | \
lookup -flags ml binary_fst_file | ...
```

If it is important for multicharacter symbols to be handled properly on the output or *upper* side, then you should specify the flag `mu`. This tag might be appropriate if the input words themselves contain no multicharacter symbols, but the user has specified a lookup-strategy script wherein virtually composed networks expect to “pass” strings containing multicharacter symbols among themselves.

```
unix> cat input | tokenize tok.fst | \
lookup -flags mu -f lookup_strategy_script | ...
```

For example, if the user has specified the lookup strategy

```
lowfst midfst highfst
```

and the `lowfst` has multicharacter symbols on its upper side, and those multicharacter symbols are also present on the lower side of `midfst`, then the flag `mu` is needed.

Finally, the `mb` flag tells **lookup** always to handle multicharacter symbols correctly on *both* sides.

```
unix> cat input | tokenize tok.fst | \
lookup -flags mb -f lookup_strategy_script | ...
```

Thus the flag `mb` is always the safest option, but it carries an unnecessary performance penalty if just `ml`, `mu`, or no flag at all is sufficient.

The various `m` flags can be combined with other flags, e.g.

```
unix> cat input | tokenize tok.fst | \
lookup -flags mbL:LTT -f lookup_strategy_script | ...
```

10.4 Using `xfst` in Batch Mode

Bulk data can also be processed by running **xfst** in batch mode. If the file `myInput` contains tokenized words, and the file `myLex` contains a transducer, then the following command will look up all the words and output the results to the file `myOutput`.

```
unix> xfst -q -e "load myLex" -e "apply up < myInput" -stop > myOutput
```

⁷The `mi` flag specifies equivalently that a parse table be created for the *input* side, which is the same as the lower side when using **lookup**.

10.5 Transducers for Morphological Analysis

10.5.1 Monolithic Lexical Transducers

Until now we have generally assumed that morphological analysis should be performed by a single monolithic transducer, a `LEXICAL TRANSDUCER`, that incorporates the lexicon, morphotactics, filters and morphophonological alternations. Such a lexical transducer is constructed by compile-time composition of various component transducers, defined using `lexc`, `xfst` and perhaps `twolc`. While monolithic lexical transducers are maximally efficient, in some cases they may become too large to be practical or even too large to compute at all on your machine. The time necessary to compile a lexical transducer may also be unjustifiable, especially if the result is temporary or used to process only a small amount of input. The `lookup` utility, with its compositions simulated at runtime, is a solution to these problems.

The other problem with a lexical transducer is that it necessarily encodes a single relation; in morphology applications this is typically a mapping between a language of properly spelled words (according to a standard orthography) and a language of analysis strings. In many applications, it may be necessary or useful to analyze input tokens via a sequence of relations, trying first to analyze the word strictly, and then if that fails, resorting to increasingly modified and relaxed relations that handle accidents or mistakes in capitalization, accentuation, spelling, etc. The `lookup` utility, with its ability to perform a user-specified sequence of analysis strategies, was designed for such applications. We will proceed with a presentation of various auxiliary transducers that can be used in morphological analysis and related applications.

10.5.2 Normalizers

Initial-Capitalization Normalization

Normalization of words is the general process of mapping accidental spelling variations to yield normalized forms for analysis. The most commonly needed normalizations in natural-language processing are those that handle initial capitalization (upper-casing) and whole-word capitalization. For example, the initial capitalization of “It” in the sentence “It is a dog.” is accidental, reflecting the English orthographical convention of capitalizing the first letter of the first word in a sentence. One way or another, “It” needs to be normalized to “it” for lookup. The all-caps spelling “HUGE” in “It was a HUGE cockroach!” needs to be reduced similarly.

Handling accidental initial capitalization in a finite-state system is easy. Consider the following rule:

```
xfst[]: define initcap a (->) A, b (->) B, c (->) C,
d (->) D, e (->) E, f (->) F, g (->) G, h (->) H, i (->) I,
```

j (->) J, k (->) K, l (->) L, m (->) M, n (->) N, o (->) O,
 p (->) P, q (->) Q, r (->) R, s (->) S, t (->) T, u (->) U,
 v (->) V, w (->) W, x (->) X, y (->) Y, z (->) Z || .# _ ;

This rule states that upper-side **a** can optionally be written as lower-side **A**, upper-side **b** can optionally be written as lower-side **B**, etc. when it appears as the first symbol in a word. If we have a network that recognizes the words “dog”, “cat”, and “mouse”, and we compose the `initcap` transducer underneath it, the resulting transducer will recognize the original words and, in addition, will analyze “Mouse” as “mouse”, “Dog” as “dog” and “Cat” as “cat”. This can be demonstrated easily in `xfst`.

```
xfst[n]: clear stack
xfst[0]: define mylang {dog} | {cat} | {mouse} ;
10 states, 11 arcs, 3 paths.
xfst[0]: print words mylang
dog
cat
mouse
xfst[0]: push initcap
xfst[1]: apply down dog
dog
Dog
xfst[1]: clear stack
xfst[0]: define mylanginitcap mylang .o. initcap ;
10 states, 14 arcs, 6 paths.
xfst[0]: print words mylanginitcap
mouse
<m:M>ouse
dog
<d:D>og
cat
<c:C>at
xfst[0]: push mylanginitcap
xfst[1]: apply up Cat
cat
xfst[1]: apply up cat
cat
```

Accidental initial capitalization is thus easily handled, and the compile-time composition of the normalizing transducer on the bottom of a morphological analyzer adds only a small number (26 or so, depending on the number of letters in the alphabet) additional arcs at the start of the network. This is only a tiny overhead in a typical morphological-analyzer network, which may contains hundreds of thou-

sands of arcs. In this trivial case, only three arcs are added. Initial capitalization is almost always best handled via compile-time composition.

All-Capitalization Normalization

The more difficult problem is all-caps normalization, recognizing “CAT”, for example, as a form of “cat”. In all-caps examples, the usual convention is that a particular letter can be capitalized only if all the letters in the word are capitalized at the same time. The transducer for such all-caps normalization is easily defined, e.g.

```
xfst[]: define upper [A|B|C|D|E|F|G|H|I|J|K|L|
M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z] ;
xfst[]: define allcaps a (->) A, b (->) B, c (->) C,
d (->) D, e (->) E, f (->) F, g (->) G, h (->) H, i (->) I,
j (->) J, k (->) K, l (->) L, m (->) M, n (->) N, o (->) O,
p (->) P, q (->) Q, r (->) R, s (->) S, t (->) T, u (->) U,
v (->) V, w (->) W, x (->) X, y (->) Y, z (->) Z
\ / .#. upper* _ upper* .#. ;
```

This rule states that any letter in the word can be accidentally capitalized, as long as all the other letters on the surface are upper-case as well; the `\ /` rule operator, introduced in Section 3.5.5, matches both the left and right contexts on the output side of the relation, which in a right-arrow rule is the lower side. Where the original network accepts “dog”, composition of allcaps on the lower side will result in a transducer that also accepts “DOG” as a form of “dog”, but not “Dog”, “dOg”, “doG” or any other string that is not all-caps. This is easily demonstrated in **xfst**:

```
xfst[n]: clear stack
xfst[0]: define mylang {dog} | {cat} | {mouse} ;
10 states, 11 arcs, 3 paths.
xfst[0]: push allcaps
xfst[1]: apply down dog
DOG
dog
xfst[1]: clear stack
xfst[0]: define mylangallcaps mylang .o. allcaps ;
18 states, 22 arcs, 6 paths.
xfst[0]: push mylangallcaps
xfst[1]: up dog
dog
xfst[1]: up DOG
dog
xfst[1]: up dOg
```

Unfortunately, the composition of allcaps on the lower side of the lexicon typically results in a very significant increase in the size of the resulting network. Even in this trivial example, the number of states jumps from 10 to 18, and the number of arcs from 11 to 22. In full-sized systems, the “explosion” may grow too large to be practical. To avoid this explosion, caused by composing allcaps at compile time, the composition can be simulated at runtime using the **lookup** utility. Assuming that the morphological analyzer, with normalization of initial capitalization already composed in, has been stored in the file `analyzer.fst`, and that the all-caps normalizer is stored in `allcaps.fst`, then the **lookup** strategy file would look as follows:

```
analyzer      analyzer.fst
allcaps      allcaps.fst

allcaps analyzer
```

Using this lookup strategy, **lookup** will look up each input token using the simulated composition [`analyzer .o. allcaps`].

Encoding Normalization

For some orthographies there exist a number of file encodings in current use; Arabic encodings, for example, including Unicode, ISO8859-6, MS 1256, and even some strict Roman-alphabet transliterations that can be converted unambiguously into Unicode. Assuming that one creates a standard Arabic morphological analyzer `arabic.fst` that has Unicode strings on the lower side, the accommodation of the other compatible encodings can be performed by creating fairly trivial transducers that map upward from the target encoding to Unicode, e.g. `iso2unicode.fst`, `ms2unicode.fst`, etc.

As with initial capitalization, an encoding transducer can usually be composed on the bottom of the standard Arabic morphological analyzer at compile time without any significant increase in size, but if there are a half dozen alternate encodings to deal with, the storage of a half dozen full-sized pre-compiled transducers might be inconvenient. By keeping the standard Arabic transducer and the encoding-mapping transducers separate, and by using a set of appropriate lookup-strategy scripts, the same core transducer can be used to handle incoming text in a variety of different but compatible encodings. A representative lookup-strategy script, for handling the MS 1256 encoding, might look like this:

```
analyzer      Arabic.fst
ms            ms2unicode.fst

ms analyzer
```

10.5.3 Relaxation

Relaxation of Accentuation Rules

Some orthographies, like those for Spanish and Portuguese, use accented letters like á, é, í, ó, ú, ü and ñ, but in some contexts and kinds of text, the accents are often missing. Even in correctly spelled Spanish, upper-case accented vowels may respectably lose their accents (although Ñ̃ is never properly reduced to N). In informal text, such as email, even lower-case vowels and the ñ are often reduced to their unaccented counterparts.

The first (orthographically respectable) deaccentuation of uppercase vowels can be controlled by the following rule:

```
xfst[n]: clear stack
xfst[0]: define upperdeaccent Á (->) A, É (->) E, Í (->) I,
Ó (->) O, Ú (->) U, Û (->) U ;
xfst[0]: push upperdeaccent
xfst[1]: save stack upperdeaccent.fst
```

The second (sloppy) deaccentuation can be controlled by a similar rule.

```
xfst[n]: clear stack
xfst[0]: define sloppydeaccent Á (->) A, É (->) E, Í (->) I,
Ó (->) O, Ú (->) U, Û (->) U, Ñ̃ (->) N, á (->) a, é (->) e,
í (->) i, ó (->) o, ú (->) u, ü (->) u, ñ (->) n ;
xfst[0]: push sloppydeaccent
xfst[1]: save stack sloppydeaccent.fst
```

Assume that the transducer that handles fully accented Spanish words is stored in `Spanish.fst`. A lookup strategy that first tries to look up a word in `Spanish.fst`, and then resorts, if necessary, to allowing upper-case deaccentuation, and then sloppy deaccentuation, is the following:

```
analyzer          Spanish.fst
upperdeaccent     upperdeaccent.fst
sloppydeaccent    sloppydeaccent.fst
```

```
analyzer
upperdeaccent analyzer
sloppydeaccent analyzer
```

General Relaxation of Spelling Rules

In some languages, even educated users may produce certain kinds of spelling errors so often that a robust natural-language processing system must be written to handle them. In Arabic orthography, for example, the official rules for spelling the glottal-stop sound are so complicated that few people fully master them.

This does not mean, however, that misspellings should be built into your basic dictionaries and rules. It is almost always best to write your core morphological analyzer to accept only formally correct spellings, so that it can be used as a basis for strict spelling checking and spelling correction (see Section 10.6). But using the **lookup** utility and appropriate spelling-relaxation transducers, similar to the accent-relaxation examples above, the core morphological analyzer can also be used to analyze characteristic misspellings.

10.5.4 Guessers

Guessers vs. Normal Morphological Analyzers

Assuming that the morphotactics and morphophonology have been correctly described, a normal finite-state morphological analyzer will still not recognize a word unless its stem is included in the **lexc** lexicon. As it may take several person-years of work to build up a lexicon with the tens of thousands of stems necessary for broad coverage of real text, it is often useful to define a GUESSER version of your analyzer. Unlike a normal morphological analyzer, wherein the set of known stems is explicitly enumerated, a guesser is designed to analyze words that are based on any PHONOLOGICALLY POSSIBLE STEM. The set of phonologically possible stems is definable, more or less precisely, using regular expressions and scripts. Guessers can be used as a general backup when normal morphological analysis fails, and they can be very useful for suggesting new stems that need to be added to the lexicon.

A GUESSER is a variant of a morphological analyzer that contains all phonologically possible stems.

Constructing a Simple Morphological Guesser

Guessers may be defined and used in various ways; this section describes a fairly straightforward approach to building a guesser that could be used in conjunction with the **lookup** utility. The basic idea of a guesser is that you substitute (or augment) the lexicon (or lexicons) of enumerated stems with a lexicon of phonologically possible stems, usually defined separately in a regular expression or script.

In a typical morphotactic description, your **lexc** file contains scores or even hundreds of LEXICONS, but only a very few represent open classes, like noun stems, verb stems, etc. as in Figure 10.4. In such traditional lexicons you manually enumerate as many real, attested stems of the language as your time and resources permit. The resulting analyzer finds solutions only for words that are based on the enumerated stems.

The standard morphological analyzer and the guesser should share a common


```

LEXICON NounStems
rana      CNN ;
vo        CCN ;
zimaba    CCN ;
...

LEXICON VerbStems
dami      CCV ;
wojo      CCV ;
pi        CCV ;
...

```

Figure 10.4: A Normal Lexicon for Morphology Contains an Explicit Enumeration of Stems for Open Classes like Noun Stems and Verb Stems

```

LEXICON NounStems
^GUESSNOUNSTEM  CCN ;    ! add a dummy entry
rana      CNN ;
vo        CCN ;
zimaba    CCN ;
...

LEXICON VerbStems
^GUESSVERBSTEM  CCV ;    ! add a dummy entry
dami      CCV ;
wojo      CCV ;
pi        CCV ;
...

```

Figure 10.5: In the **lexc** Lexicon, Add Dummy Entries in Each Open Class

```

LEXICON NounStems
^GUESSNOUNSTEM CCN1 ; ! dummy entry for CCN1
^GUESSNOUNSTEM CCN2 ; ! dummy entry for CCN2
rana      CNN1 ;
vo        CCN1 ;
zimaba    CCN2 ;
...

LEXICON VerbStems
^GUESSVERBSTEM CCV1 ; ! dummy entry for CCV1
^GUESSVERBSTEM CCV2 ; ! dummy entry for CCV2
dami      CCV1 ;
wojo      CCV2 ;
pi        CCV2 ;
...

```

Figure 10.6: Add a Dummy Entry for Subclass Having a Distinct Continuation

lexc file. One way to start is to insert a dummy entry in each open class as in Figure 10.5, where **^GUESSNOUNSTEM** and **^GUESSVERBSTEM** are defined as multicharacter symbols. If you have multiple continuation classes for nouns, e.g. CCN1, CCN2, etc., then you probably want to add a GUESS entry for each, as in Figure 10.6.

Now compile your **lexc** grammar in the usual way. It will be just as before, but it will now also include these pseudo-stems that look like **^GUESSNOUNSTEM** and **^GUESSVERBSTEM**, expanded out with all possible prefixes and suffixes. Before composing on any filters or rules, define the set of phonologically possible lexical stems. It is usually most convenient to do this in **xfst** regular expressions or scripts. The crudest model is simply sigma-plus, i.e. something like

```

xfst[n]: clear stack
xfst[0]: define guesstems [a|b|c|d|e|f|g|h|i|j|
k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z]+ %+Guess:0 ;
xfst[0]: push guesstems
xfst[1]: save stack possible-stems.fst

```

modifying the alphabet as necessary to reflect the alphabet of the underlying stems in your particular language. Instead of just sigma-plus, one could use an iteration operator something like $\wedge\{3, 8\}$, which would specify that guessed stems must

have three to eight characters.

```
xfst[0]: define guesstems [a|b|c|d|e|f|g|h|i|j|
k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z]^{3,8} %+Guess:0 ;
```

A more sophisticated modeling of possible stems will be shown below. Note in this example that each guessed stem has the tag `+Guess` on the upper side. Then, assuming that the network from your `lexc` grammar is stored in file `lexc.fst`, continue with

```
xfst[1]: clear stack
xfst[0]: load stack lexc.fst
xfst[1]: substitute defined guesstems for %^GUESSNOUNSTEM
xfst[1]: substitute defined guesstems for %^GUESSVERBSTEM
xfst[1]: define AllInclusive
```

Then apply all the usual filters and rules of the morphological analyzer. This should leave you with an `AllInclusive` FST that contains all your enumerated attested stems plus all possible guessed stems, with each guessed stem having a `+Guess` tag on the top. Now separate out the FST with the real stems from the FST with the guesses using finite-state filters (see Section 6.3.2).

```
xfst[1]: clear stack
xfst[0]: read regex ~$[%+Guess] .o. AllInclusive ;
xfst[1]: save stack MyLanguage.fst

xfst[1]: clear stack
xfst[0]: read regex $[%+Guess] .o. AllInclusive ;
xfst[1]: save stack MyLanguage-guesser.fst
```

The resulting `MyLanguage.fst` should be exactly like your original morphological-analyzer FST, and the new guesser will be stored in `MyLanguage-guesser.fst`. The basic idea is to try to look up each input word first in `MyLanguage.fst`, and then, if and only if that fails, in `MyLanguage-guesser.fst`. That can be accomplished using the simple **lookup** script shown in Figure 10.7.

A Better Definition of Possible Stems

You should try to define the notion of phonologically possible stem as tightly as possible. To improve on the simple sigma-plus solution shown above, let us assume that we know some facts of the phonology and want to model them. For now, let's assume that our language has stems that are always of the form CV, CVCV or

```

analyzer      MyLanguage.fst
guesser      MyLanguage-guesser.fst

analyzer
guesser

```

Figure 10.7: A Simple **lookup** Script that Incorporates a Guesser. The variable `analyzer` is associated with the network in the file `MyLanguage.fst`; and the variable `guesser` is associated with the network in `MyLanguage-guesser.fst`. The minimal lookup strategy is first to try looking up each word in `analyzer`. If that fails, then try to look up the word using `guesser`.

CVCVCV. The following **xfst** script defines `guessstems` appropriately.

```

xfst[n]: clear stack
xfst[0]: define Cons [b|c|d|f|g|h|j|k|l|
m|n|p|q|r|s|t|v|w|x|y|z] ;
xfst[0]: define Vowel [a|e|i|o|u] ;
xfst[0]: define Syllable Cons Vowel ;
xfst[0]: define guessstems Syllable^{1,3} ;

```

There may be consonant clusters to consider (the allowable clusters are usually very limited), restricted sets of consonants that can begin or end a stem, CVC syllables, long vowels, tones, etc. depending on your language and on the conventions you use in your normal **lexc** file for notating stems. Just try to model the language of phonologically possible stems as closely as possible in `guessstems` network, and then perform the indicated substitution operation(s) as indicated above.

Morphological analyzers based on **lookup** and lookup-strategy scripts can of course grow as complicated as necessary, referring to transducers that map encodings, perform normalization of capitalization, allow relaxation of accentuation and other orthographical rules, and incorporate guessing of various kinds.

10.6 Spelling

10.6.1 Spelling Checkers

Spelling checkers are modules usually built into a larger application, like a text editor, and their basic task is to look up all the words in a document, using some kind of language model, and to indicate which words were not found. The surrounding application then typically marks the not-found words, presumably misspellings, for attention from the user.

Many traditional spelling checkers are based on simple wordlists extracted from corpora, and this approach is known to be problematic for highly inflecting languages. A single Finnish verb, for example, may have over 10,000 different inflected/derived forms, and so adding a single new Finnish lexeme to a wordlist is no fun. Luckily, finite-state networks model languages, store millions of words compactly, and can be defined with powerful **lexc** descriptions and **xfst** regular expressions rather than just wordlists. If one has a well-written grammar of Finnish, then adding a new lexeme should be as simple as adding the stem or baseform to the **lexc** description, with the grammar expanding it out automatically into the 10,000+ different forms. Finally, finite-state networks perform acceptance or lookup efficiently, rejecting all unknown words; this makes finite-state networks the perfect technical foundation for spelling checkers.

In practice, developers often start by creating a morphological analyzer, and then extracting the lower-side language network to serve as a spelling checker. Additional words, including those supplied by the user, can be trivially unioned into the network or built into an auxiliary network using applications like **read text** (see Section 7.4.2).

10.6.2 Spelling Correctors

Finite-state methods can also be applied naturally to implement spelling correction. Assume that we have a large-coverage morphological analyzer, and that we again extract the network that accepts the lower-side language and use that as our language model. Whereas spelling checking will simply accept or reject each candidate word, spelling correction should in addition suggest a set of properly spelled words for each presumably misspelled candidate word that does not belong to the known language. The suggested corrections should of course be similar to the misspelled word, reflecting common misspellings and typographical errors.

Spelling correction can be implemented using rule-defined transducers and a suitable **lookup** strategy file. To support the correction facility, the developer writes a set of Replace Rules that describe typical misspellings and typos as optional mappings from good spellings down to bad spellings, e.g. for English you might have rules like

```
p h (->) f           ! e.g. "fone" for "phone"
.o.
```

```

o u g h (->) u | | _ .# . ! thru for through
.o.
o m m o (->) o m o      ! e.g. "accomodation"
.o.
e i (->) i e           ! thier for their
.o.
h t e (->) t h e | | .# . _ .# . ! common typo
.o.
...

```

For South and Central American Spanish, the following rules are a start. Note that in a spelling corrector, deaccentuation is best treated as a kind of spelling error. People who bother to use a spelling checker/corrector are aiming to conform to the formal rules.

```

[ z (->) s , s (->) z ]
.o.
[ c (->) s , s (->) c | | _ [ i | e ] ]
.o.
[ Á (->) A , É (->) E , Í (->) I ,
Ó (->) O , Ú (->) U , Ü (->) U , Ñ (->) N ,
á (->) a , é (->) e , í (->) i , ó (->) o ,
ú (->) u , ü (->) u , ñ (->) n ]
.o.
...

```

There may be dozens of such rules, or hundreds, including elision, epenthesis, and limited cases of metathesis. It takes a while to get into the right mindset for writing such rules, and ordering them correctly, but it's a useful exercise. The rules should be based on a good sample of the real errors made by real people. Of course, the errors can vary by dialect. In South America, people often confuse orthographical **s** and **z**, which are both pronounced /s/. In Spain, however, the phonemes are distinct and such confusion is unlikely.

Assuming that the model of correctly spelled words is stored in `mylanguage.fsm` and that the spelling-correction rules are stored in `correction.fst`, the strategy for spelling checking and correction is then to

1. First apply `mylanguage.fsm` to the candidate word. If found, return Success. The word is good.
2. Else apply `[mylanguage.fsm .o. correction.fst]` in an upward direction to the misspelled word. Return the set of properly spelled suggestions.

Thus if the input word is “fone”, it will not be found in `English.fsm`, but the lookup in `[English.fsm .o. correction.fst]`, where `correction.fst` contains the rule `p h (->) f`, will yield the possible correction “phone”. Whatever the language, the basic network will constrain the set of corrections to words

known to be spelled correctly, and the correction rules will constrain the possible mappings between misspelled words and properly spelled words.

10.7 Beyond Tokenization and Morphological Analysis

It is beyond the scope of this book to treat applications beyond morphological analysis and generation in any detail; the field of finite-state linguistic development deserves a number of specialized books. However, we list here a few of the higher linguistic applications where finite-state techniques have proved useful.

10.7.1 Other Uses for `tokenize`

The `tokenize` utility is not necessarily limited to tokenization; it uses a finite-state transducer to transform one string of input characters into another string of output characters, and so it might be used for a variety of text transformations, including mapping between character encodings, insertion or deletion of markup tags, lower-casing or upper-casing, etc. Such tasks can often be performed by Unix utilities such as `sed` and `gawk`, or by languages like Perl and Python, but transduction may be preferable, depending on the taste of the developers and the difficulty of the transformation to be performed.

The transducer used by `tokenize` must satisfy the formal requirements detailed in Section 10.2.3, page 532, i.e. it must accept the universal language on the lower side and be unambiguous.

10.7.2 Advanced Tokenization and Normalization

Experienced computational linguists will have noticed that the tokenization performed by `tokenize` is fairly simple, performed in isolation and deterministic. Normalization can also be much more complicated, including the rejoining of words that are hyphenated over line breaks. Advanced normalization and non-deterministic tokenization are topics pursued at **Xerox** Research, but they are beyond the scope of this book.

10.7.3 Part-of-Speech Disambiguation

The `lookup` utility outputs one line for each solution, and the set of solutions for each input word is terminated by a blank line. For example, where `English.fst`

walk	walk	+Noun+Sg
walk	walk	+Verb+Pres+Non3sg
the	the	+Det+Def+SP
dog	dog	+Noun+Sg
dog	dog	+Verb+Pres+Non3sg

Figure 10.8: Default Output from **lookup**. The first column shows the input word, the second column the baseform, and the third column the set of analysis tags. The set of analyses for each input word is terminated with a blank line.

is a lexical transducer for English, the Unix command

```
unix> echo "walk the dog" | \
tr -sc "[:alpha:]" "[\n*]" | \
lookup English.fst
```

produces the output shown in Figure 10.8, where both *walk* and *dog* are ambiguous, with noun and verb readings.⁸

A part-of-speech disambiguator, also called a tagger, takes the ambiguous output of a morphological analyzer, examines each ambiguous word in context, and tries to select the single correct analysis for each word. For the trivial example at hand, the desired output of a tagger would be something like

```
walk    walk    +Verb+Pres+Non3sg

the     the     +Det+Def+SP

dog     dog     +Noun+Sg
```

or, in another commonly used format

```
walk/V the/Det dog/N
```

or, in a version of XML,

```
<word cat="N">walk</word>
<word cat="Det">the</word>
<word cat="V">dog</word>
```

⁸Other output formats can be produced by setting flags; see Section 10.3.6, page 543.

Whatever the output, the basic idea is that the ambiguous word *walk* is resolved as a verb and the ambiguous *dog* is resolved as a noun by a part-of-speech disambiguator that examines each work in context and makes a choice.

While part-of-speech disambiguation is beyond the scope of this book, it should be understood that the output of the **lookup** utility could be piped to any disambiguator utility that is written to accept the kind of output that **lookup** produces.

```
cat input_file | \
tokenize tokenizing_fst | \
lookup lookup_fst | disambiguator ....
```

10.7.4 Parsing and Higher Linguistic Processing

Syntactic parsing, which usually involves assigning a tree structure or dependency structure to an input sentence, is also beyond the scope of this book. However, it should be understood that tokenization and morphological analysis are prerequisite modules inside almost any system that performs syntactic parsing, and parsing itself is a required module inside almost any system for natural-language understanding, question answering, discourse analysis, machine translation, etc.

One style of parsing, often known as shallow or robust parsing, can be performed using finite-state transducers. In practice, such robust parsers are often designed to accept input from a part-of-speech disambiguator and to output relatively flat parse trees. Again, the overall system can often be constructed with Unix pipes.

```
cat input_file | \
tokenize tokenizing_fst | \
lookup lookup_fst | disambiguator ... | \
robustparser ...
```

Disambiguators and parsers may, of course, be written in any convenient way and may employ techniques beyond finite-state power.

10.8 Application Summary

Finite-state methods cannot do everything in natural-language processing, but they have been found adequate, elegant and very efficient for lower-level tasks like tokenization, normalization, relaxation, morphological analysis, guessing, spelling checking and correction, part-of-speech tagging, and even some kinds of robust parsing or “chunking”.

The integration of finite-state modules inside larger systems is dependent on runtime code, such as **tokenize** and **lookup**, that can access and apply pre-compiled networks. Finite-state components can often be connected into larger systems via standard Unix pipes. Within the **lookup** utility, the use of lookup-strategy scripts

provides flexibility and simulated compositions that can help keep transducers small.

The **tokenize** and **lookup** utilities included with this book and described in this chapter are adequate for many kinds of testing and prototype applications, but they have their limitations. **Xerox** selectively licenses XeLDA, a much more sophisticated runtime environment, for advanced research and commercial applications.⁹

⁹<http://www.fsmbook.com/>

Chapter 11

Computational Complexity

Contents

11.1 Complexity	563
11.1.1 The Challenge from Barton	563
11.1.2 The Reply from Koskenniemi and Church	564
11.1.3 Practical Consequences	564
11.2 Finite-State Solutions to Constraint Problems	565
11.2.1 Barton’s SAT Problem	565
The Original Problem and Two-Level Solution	565
A twolc Solution	568
An xfst Solution	569
11.2.2 The Einstein Problem	571
Karttunen’s Einstein Solution	572
Beesley’s Solution	575
11.2.3 Einstein II Exercise	577
Description	577
Constraints	577
Question	578

11.1 Complexity

11.1.1 The Challenge from Barton

In presenting his Two-Level implementation of finite-state morphology, Kimmo Koskenniemi made a number of claims about its computational complexity. In his dissertation (Koskenniemi, 1983) he writes, “The computational complexity of the two-level model is minimal, it relies mainly on small finite-state automata. In this way it provides an estimate of the complexity of the ‘real’ morphological and morphophonological processes.” He also writes that “each two-level rule ultimately

corresponds to a finite state automaton, and there exists an efficient algorithm using these automata for both production and analysis of word forms.”

Then in a series of publications in the mid 1980s, G. Edward Barton, Jr. (Barton, 1986b; Barton, 1986a; Barton, 1987) challenged such claims, showing that, in the worst case, Two-Level Morphology could solve problems known to be NP-complete, and therefore inefficient to process. “It follows,” he wrote, “that the finite-state two-level framework itself cannot guarantee computational efficiency.” Barton took this to be a serious fault, requiring some kind of constraints to limit the power of Two-Level Morphology; and many computational linguists accepted and continue to accept his conclusions as damaging criticisms of the Two-Level and more general finite-state paradigms.

11.1.2 The Reply from Koskenniemi and Church

Barton’s challenge was answered by Koskenniemi and Church in 1988 (Koskenniemi and Church, 1988). They argue that Finite-State Morphology has been shown, in practice, to be efficient for processing natural-language words, even in notoriously difficult languages like Finnish. Barton’s NP-complete worst-case example, though cleverly constructed, is not a natural-language problem at all but rather the Boolean Satisfaction (SAT) problem for formulas in the propositional calculus. Where symbols **P**, **Q**, **R** and **S** represent propositions, such as “Kimmo is Finnish” and “Mary is a lawyer”, that are either true or false, the SAT problem is to find the consistent assignments of true or false to each variable such that whole sentences like

$$(P \mid Q \mid R \mid S) \ \& \ \sim P \ \& \ (P \mid Q) \ \& \ (\sim R \mid \sim S) \ \& \ (R \mid P \mid \sim Q)$$

are “satisfied”, having an overall true value. There is, in this example, a single consistent assignment of values that leads to satisfaction: P=False, Q=True, R=True, and S=False.

The SAT problem is indeed NP-complete and generally takes a long time for both human beings and computers to solve. The problem is that such formulas exhibit an unlimited number of “long-distance dependencies”. Natural-languages can display similar long-distance dependencies, such as umlaut and vowel harmony, but the number of such processes in natural-language words appears, in practice, to be strictly limited, with an observed maximum of perhaps two.

11.1.3 Practical Consequences

Koskenniemi and Church argue that Barton’s conclusions are correct but basically irrelevant. Two-Level morphology has proved in practice to be efficient for word-analysis and -generation in natural language, and the same practical results have been shown repeatedly in applications of finite-state morphology at **Xerox**. It should also be pointed out that Barton’s worst-case limitations apply not only to

Two-Level or Finite-State morphology, but to any analyzer that is at least of finite-state power.

11.2 Finite-State Solutions to Constraint Problems

Now that the world is safe again for finite-state natural-language morphology,¹ and now that finite-state methods have been shown adequate to handle the non-linguistic SAT problem, it is both instructive and amusing to devise finite-state solutions to a variety of SAT-like logical-constraint problems.

11.2.1 Barton's SAT Problem

Let's look first at Barton's original SAT problem (Barton, 1986b; Koskenniemi and Church, 1988) and examine solutions using finite-state tools.

The Original Problem and Two-Level Solution

As originally stated by Barton, the propositional sentences or formulas to be analyzed are based on three variables, **x**, **y** and **z** that can take the values **T** (true) or **F** (false). The negation of a variable's value is shown with a prefixed minus sign, e.g. $\neg x$, and the disjunction or OR-ing of values is indicated by concatenation. Thus xyz denotes "x OR y OR z" and $\neg xy$ denotes "(NOT x) OR y". Conjunction is indicated with commas, e.g. $xyz, \neg x$ denotes "(x OR y OR z) AND (NOT x)". Barton's Two-Level solution, with hand-compiled rules expressed in KIMMO-style notation (Antworth, 1990), is shown in Figure 11.1. When input strings such as " $xyz, \neg x, y-z, xz$ " are generated via this grammar, all the satisfaction solutions are returned.

The x-consistency rule, shown as a network in Figure 11.2, ensures that **x** is always given (that is, realized as) the same value across the entire expression. In particular, if **x** is realized as **T** (true) then the machine moves into state 2, and **x** must be consistently realized as **T** from then on; similarly, if **x** is realized as **F** (false) then the machine moves into state 3 and constrains **x** to be realized as **F** from then on. The other two consistency rules similarly constrain the realizations of **y** and **z**.

The satisfaction rule, shown as a diagram in Figure 11.3, constrains the overall expression to be true. Within a disjunction, any true value moves the machine from state 1 to final state 2, where it remains for the rest of the disjunction; the start of a new disjunction, signaled by the comma that indicates AND, puts the machine back in state 1. State 3 handles the negation prefix; iff the following value is false, then the current disjunction is necessarily true, and the machine moves into state 2.

¹Debate continues on the formal descriptive power of finite-state morphology systems, and whether it matters.

```

ALPHABET x y z T F - ,
ANY =
END

"x-consistency" 3 3
    x x =
    T F =
1:  2 3 1
2:  2 0 2
3:  0 3 3

"y-consistency" 3 3
    Y Y =
    T F =
1:  2 3 1
2:  2 0 2
3:  0 3 3

"z-consistency" 3 3
    z z =
    T F =
1:  2 3 1
2:  2 0 2
3:  0 3 3

"satisfaction" 3 4
    = = - ,
    T F - ,
1.  2 1 3 0
2:  2 2 2 1
3.  1 2 0 0

END

```

Figure 11.1: The SAT Problem in KIMMO-Style Notation

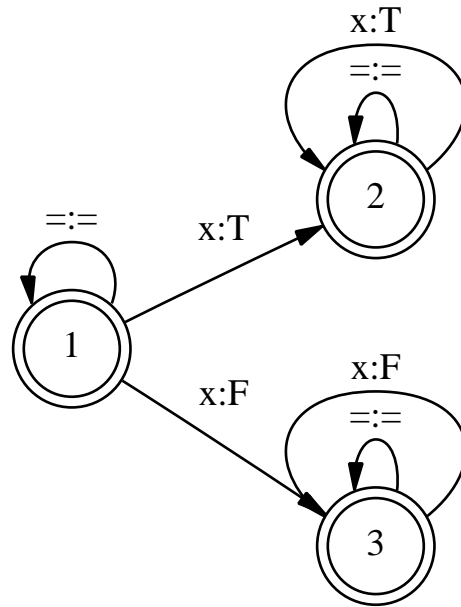


Figure 11.2: The X-consistency Rule as a Network

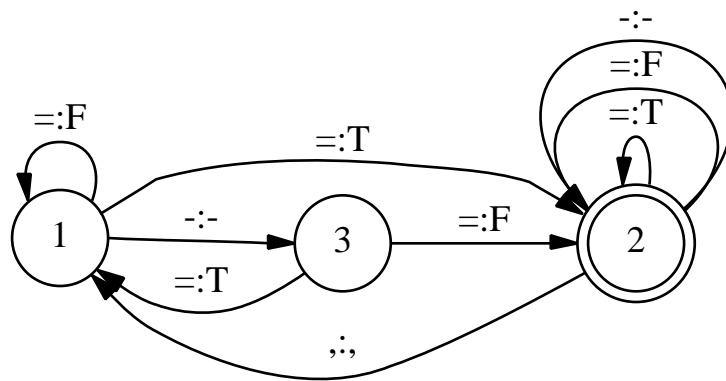


Figure 11.3: The Satisfaction Rule as a Network

A twolc Solution

Karttunen has written a solution to the SAT problem in **twolc**. In a slight complication of the original problem, this solution allows up to four propositional variables: **p, q, r** and **s**. Again, propositional formulas are generated via the grammar to find solutions.

Alphabet

```
p:T  q:T  r:T  s:T
p:F  q:F  r:F  s:F
%-  %, ;
```

Sets

```
Variable = p q r s ;
TruthValue = T F ;
```

Definitions

```
SimpleClause = :T | :F ;
Expr = %- | SimpleClause ;
NoNegation = [#: | %, | SimpleClause] ;
TrueSimpleClause = :T ;
FalseSimpleClause = :F ;
TrueClause = NoNegation TrueSimpleClause
| %- FalseSimpleClause ;
```

Rules

```
"Satisfaction"
%, => TrueClause Expr* _ ;
```

```
"Consistency"
Vx:Ty <= $[Vx:Ty] _ ;
      where Ty in TruthValue
            Vx in Variable ;
```

In this formulation, the comma (the AND operator) is restricted to appearing after a true disjunct, and the consistency rule, employing the convenient **twolc** `where` clause, makes sure that each variable has a consistent value throughout the overall expression.²

²The Consistency rule triggers a set of spurious compiler warnings about left-arrow conflicts. The compiler sees potential problems in contexts where consistency does not hold. It cannot “see ahead” that the rule it is currently working on will have the effect of eliminating all such cases. The warnings can be ignored.

An **xfst** Solution

In yet another variation, which tries to make the input and output languages more readable, Beesley has written the following solution in the form of an **xfst** script. The propositional variables used here are **P**, **Q**, **R** and **S**, and the input formulas use **|** for disjunction, **&** for conjunction, and **-** or **~** for negation. For human convenience and readability, disjuncts can optionally be surrounded with parentheses, and spaces are ignored. Arbitrarily, the grammar is written so that the resulting **FST** is applied in an upward direction (**apply up**) to the input formulas, e.g.

```
xfst[1]: apply up P & Q & -R & -S & ( R | S | P)
```

and the output contains indications of the form $P=T$, with any duplicates weeded out.

```
# Barton's SAT Problem in an xfst Script

# Use 'apply up' on a Boolean satisfaction sentence
#   in up to four variables P Q R S to
#   find satisfaction possibilities
# Use | for disjunctions (parentheses can optionally
#   be placed around disjunctions only)
# Use prefixed ~ or - for negation of a variable,
#   e.g. ~P or -P
# Use & for conjunction of disjunctions
# You can insert spaces for readability

# input strings should look like
# P & Q & R
# ( P | Q | R | S )
# (~P | Q) & R & (S | P) & P & ~S
# ( P | Q | R ) & ~Q & ~P
# ~P & ( Q | R | S | P ) & (~Q | ~S) & Q & ~R
# P & Q & -R & -S & ( R | S | P )
# etc.

# filter to ensure consistency of T/F assignments
#   throughout the input; e.g. you cannot have P=True
#   in one part of the formula and P=False in another

define ConsistencyFilter ~[ $Pt & $Pf ] & ~[ $Qt & $Qf ] &
                        ~[ $Rt & $Rf ] & ~[ $St & $Sf ] ;

# within each disjunction set, the set must not contain an
# ampersand (which separates disjunction sets), and must
```

```

# either begin with a true setting, or contain a true
# setting, not preceded by ~, or contain the negation
# of a false setting

define DisSat ~$[%&]
  &
  [ [ Pt | Qt | Rt | St ] ?*
    |
    $[ [ \%~ [ Pt | Qt | Rt | St ] ]
      |
      [ \%~ [ Pf | Qf | Rf | Sf ] ]
    ]
  ] ;

# In the conjunction of possibly multiple disjunctions,
# all the disjunctions must be satisfied

define AllSat DisSat [ %& DisSat ]* ;

# Modify the upper language for more readable display
# of the variable settings

read regex [ P=T% <- Pt , P=F% <- Pf ,
            Q=T% <- Qt , Q=F% <- Qf ,
            R=T% <- Rt , R=F% <- Rf ,
            S=T% <- St , S=F% <- Sf ,
            0 <- %& , 0 <- \%~
          ]

.o.
0 <- Pt || .#. $Pt _ .o. 0 <- Pf || .#. $Pf _
.o.
0 <- Qt || .#. $Qt _ .o. 0 <- Qf || .#. $Qf _
.o.
0 <- Rt || .#. $Rt _ .o. 0 <- Rf || .#. $Rf _
.o.
0 <- St || .#. $St _ .o. 0 <- Sf || .#. $Sf _
.o.
AllSat
.o.
ConsistencyFilter
.o.
[ Pt | Pf <- P,
  Qt | Qf <- Q,
  Rt | Rf <- R,

```

```

St | Sf <- S ]
.O.
[ 0 <- % ) ,
  0 <- % ( ,
  0 <- % ,
  %~ <- %- ,
  0 <- % | ] ;

```

Note that rules composed on top of the final **FST** remove multiple instances of variables and present the results in a fairly readable format: e.g. the analysis of

```
~P & ( Q | R | S | P ) & ( ~Q | ~S ) & Q & ~R
```

returns the string “P=F Q=T R=F S=F”.

11.2.2 The Einstein Problem

The following problem was allegedly proposed by Albert Einstein, who (also allegedly) claimed that 98% of people could not solve it. This and many other constraint problems can be solved elegantly with finite-state methods.

Here are the facts or constraints to be resolved:

1. There are five houses in a row, each one of a different color.
2. In each house there lives a person, each one of a different nationality.
3. Each person uniquely drinks one kind of beverage, smokes one brand of cigarette, and raises one kind of animal.
4. The Englishman lives in the red house.
5. The Swede raises dogs.
6. The Dane drinks tea.
7. The green house is just to the left of the white house.
8. The owner of the green house drinks coffee.
9. The Pall Mall smoker raises birds.
10. The owner of the yellow house smokes Dunhills.
11. The man who lives in the center house drinks milk.
12. The Norwegian lives in the first house.
13. The man who smokes Blends lives next door to the one who raises cats.

14. The man who raises horses lives next door to the Dunhill smoker.
15. The man who smokes Blue Masters drinks beer.
16. The German smokes Prince cigarettes.
17. The Norwegian lives next door to the blue house.
18. The man who smokes Blends has a neighbor who drinks water.

Problem: Find which person raises fish.

As bizarre as this problem may seem, it can be solved by a simple simultaneous resolution of constraints, and there is in fact just a single valid solution.

Karttunen's Einstein Solution

Karttunen's solution is written in the form of an **xfst** script. As always, an **xfst** script is run by calling the **source** utility.

```
# Solution to the Einstein Problem as an xfst Script

# We need five basic variables:
# Color of the house,
# Nationality of the owner,
# and his favorite Drink,
# Cigarette, and Pet.
# We define each variable as the language of the
# possible values of the variable.

clear stack

define Color [blue | green | red | white | yellow];
define Nationality [Dane | Englishman | German |
    Swede | Norwegian];
define Drink [beer | coffee | milk | tea | water];
define Cigarette [Blend | BlueMaster | Dunhill |
    PallMall | Prince];
define Pet [birds | cats | dogs | fish | horses];

define House Color Nationality Drink Cigarette Pet ;

# With five variables each taking one of five
```

```

# possible values, this gives quite a number of
# possible households,  $5 \times 5 \times 5 \times 5 \times 5 = 3125$ , to be
# exact. A road with five houses next to each other,
#  $\text{House}^5$ , provides an astronomical number of possible
# combinations of colors, nationalities, drinks,
# cigarettes and pets.

# To solve Einstein's puzzle, we represent each of
# the fifteen constraints as a regular language and
# intersect these languages with the initial set of
# all possibilities. If all goes well at the end we
# will know who keeps fish. For example, we can
# interpret 'The Englishman lives in the red house.'
# as  $[\text{red Englishman}]$ . This constraint is trivial
# to encode because in our representation of a house,
# the color and the nationality are adjacent. The
# second constraint, The Swede keeps dogs could be
# represented as  $[\text{Swede Drink Cigarette dogs}]$  but
# we will choose a less verbose formulation,
#  $[\text{Swede } \sim \text{Pet dogs}]$ , that does not explicitly
# list the variables that separate Nationality and
# Pet. The fifteen constraints are shown below.

define C1  $[\text{red Englishman}]$ ;
# The Englishman lives in the red house.
define C2  $[\text{Swede } \sim \text{Pet dogs}]$ ;
# The Swede keeps dogs.
define C3  $[\text{Dane tea}]$ ;
# The Dane drinks tea.
define C4  $[\text{green } \sim \text{Color white}]$ ;
# The green house is just to the left of the white one.
define C5  $[\text{green } \sim \text{Drink coffee}]$ ;
# The owner of the green house drinks coffee.
define C6  $[\text{PallMall birds}]$ ;
# The Pall Mall smoker keeps birds.
define C7  $[\text{yellow } \sim \text{Cigarette Dunhill}]$ ;
# The owner of the yellow house smokes Dunhills.
define C8  $[\text{House}^2 \sim \text{Drink milk } \sim \text{Drink House}^2]$ ;
# The man in the center house drinks milk.
define C9  $[\text{? Norwegian ?*}]$ ;
# The Norwegian lives in the first house.
define C10  $[\text{Blend ? } \sim \text{Pet cats} \mid \text{cats } \sim \text{Cigarette Blend}]$ ;
# The Blend smoker has a neighbor who keeps cats.
define C11  $[\text{horses } \sim \text{Cigarette Dunhill} \mid$ 

```

```

    Dunhill ? ~$Pet horses];
# The man who keeps horses lives next to the Dunhill
# smoker.
define C12 $[beer BlueMaster];
# The man who smokes Blue Masters drinks beer.
define C13 $[German ~$Cigarette Prince];
# The German smokes Prince.
define C14 $[Norwegian ~$Color blue
    | blue ? ~$Nationality Norwegian];
# The Norwegian lives next to the blue house.
define C15 $[Blend ~$Drink water
    | water ? ~$Cigarette Blend];
# The Blend smoker has a neighbor who drinks water.

# All that we need to do to solve the problem is
# to impose the constraints on the row of five
# houses by intersection. The solution below is
# almost correct.

define Solution [House^5 & C1 & C2 & C3 & C4 & C5 &
    C6 & C7 & C8 & C9 & C10 &
    C11 & C12 & C13 & C14 & C15];

# The result is a network with five paths. In four
# of the solutions nobody keeps fish and two people
# have the same kind of pet. We need one final
# constraint, presupposed by the question
# "Who keeps fish?":

define C16 $fish;
# There is someone who keeps fish.

define Solution [House^5 & C1 & C2 & C3 & C4 & C5 &
    C6 & C7 & C8 & C9 & C10 &
    C11 & C12 & C13 & C14 & C15 & C16];

# With C16 added, only one solution remains.
# To make it easier to see, we compose the solution
# with the transducer that adds some prose around the
# pieces along the one remaining path:

define Describe [House -> "In the "... .o.
    Color -> ... " house " .o.
```

```

Nationality -> "the " ... " " .o.
Drink -> "drinks "... ",\n" .o.
Cigarette -> "smokes "... "s, and " .o.
Pet -> "keeps " ... ".\n"] ;

```

We can now see the solution.

```

regex [Solution .o. Describe];
print lower-words

```

```

In the yellow house the Norwegian drinks water,
smokes Dunhills, and keeps cats.
In the blue house the Dane drinks tea,
smokes Blends, and keeps horses.
In the red house the Englishman drinks milk,
smokes PallMalls, and keeps birds.
In the green house the German drinks coffee,
smokes Princes, and keeps fish.
In the white house the Swede drinks beer,
smokes BlueMasters, and keeps dogs.

```

In short, it's the German who keeps the fish.

Beesley's Solution

There are usually many ways to model the same constraint problem. Beesley's solution, also written in the form of an **xfst** script, inserts "Wall" symbols to separate one house from another. In the end, the result is equivalent.

```

clear stack

define Color      green | white | blue | red | yellow ;
define Nationality German | English | Swede | Dane |
                  Norwegian ;
define Beverage   tea | coffee | beer | water | milk ;
define Cigarette  Prince | PallMall | Dunhills |
                  BlueMasters | Blends ;
define Pets       dogs | birds | cats | horses | fish ;

define House      Color Nationality Beverage Cigarette Pets ;

define FiveHouses Wall [ House Wall ]^5 ;

define NoWall     [\\Wall]* ;

```

```

define NextDoor NoWall Wall NoWall ;

read regex [ [ FiveHouses
&
$[ red NoWall English ]
&
$[ Swede NoWall dogs ]
&
$[ Dane NoWall tea ]
&
$[ green NextDoor white ]
&
$[ green NoWall coffee ]
&
$[ PallMall NoWall birds ]
&
$[ yellow NoWall Dunhills ]
&
Wall [ House Wall ]^2 $[ milk ] [ Wall House ]^2 Wall
&
$[ Norwegian ] Wall [ House Wall]^4
&
[ $[ Blends NextDoor cats ] |
  $[ cats NextDoor Blends ] ]
&
[ $[ Dunhills NextDoor horses ] |
  $[ horses NextDoor Dunhills ] ]
&
$[ beer NoWall BlueMasters ]
&
$[ German NoWall Prince ]
&
[ $[ Norwegian NextDoor blue ] |
  $[ blue NextDoor Norwegian ] ]
&
[ $[ Blends NextDoor water ] |
  $[ water NextDoor Blends ] ]
&
$[ fish ]
] .o. ? -> ... % .o. Wall -> "\n" ].1 ;
print words

```

The output of this script is

yellow Norwegian water Dunhills cats
blue Dane tea Blends horses
red English milk PallMall birds
green German coffee Prince fish
white Swede beer BlueMasters dogs

showing again that it's the German who raises fish.

11.2.3 Einstein II Exercise

If you would like to try another puzzle of this kind, here is a variant of Einstein's puzzle from www.riddleaday.com. Once again, the trick is to model the constraints as languages and to find the unique solution via intersection.

Description

There are five cars parked next to each other in the parking garage. Each car is a different color. Each of the five car owners drinks a different beverage, has a different hobby (or participates in a different sport) and owns a different breed of dog. No two owners prefer the same beverage, have the same hobby (or sport) or own the same breed of dog.

Constraints

1. Mark owns the first car.
2. Lucie owns the teal car.
3. Dave drinks sparkling water.
4. The purple car is on the left of the tan car.
5. The one who races cars parks next to the one who drinks espresso.
6. Brenda has a Yellow Lab.
7. The one who golfs, also drinks tea.
8. The purple car owner drinks raspberry mocha.
9. The person who collects stamps also has two poodles.
10. The person who races cars, parks next to the one who has a terrier.
11. The person who has a collie parks next to the one who swims.
12. Susan paints oil paintings.
13. The owner of the center car drinks grape soda.

14. Mark parks his car next to the blue car.
15. The owner of the green car swims.

Question

The question is: Who owns the Bichon Frise? (It's a breed of dog.)

Part IV

Appendices

Appendix A

Graphing Quiz

Draw a network, using circles for states and labeled arrows for arcs, for each of the regular expressions below. Then, if the language is finite, enumerate the strings; if the language is infinite, give a few examples of the strings.

1. a
2. $d\ o\ g$
3. $[d\ |\ o\ |\ g]$
4. $[d\ o\ g\ |\ c\ a\ t\ |\ h\ o\ r\ s\ e]$
5. $[d\ o\ g\ \%+Noun\ \%+Sg]$
6. $[d\ o\ g\ |\ h\ o\ u\ s\ e\ |\ f\ o\ o\ d]$
7. $[d\ o\ g]\ |\ [h\ o\ u\ s\ e\ |\ f\ o\ o\ d]$
8. $[d\ o\ g][h\ o\ u\ s\ e\ |\ f\ o\ o\ d]$
9. $a\ b^*\ c$
10. $a\ b^+\ c$
11. $[h\ a]^*$
12. $[a\ |\ b\ |\ c]^*$
13. $[x\ |\ y\ |\ z]^+$
14. $a\ (b)\ c$
15. $a\ [b\ c]^*\ d$
16. $a\ [b\ |\ c]^+\ (d)$

17. [h u n d | e l e f a n t | k a t]
[e g | e t | i n]* o (j) (n)
18. (m a l | n e) [b o n | a l t]
[e g | e t]+ a (j) (n)
19. [d o g | p i g | d u c k | f r o g]
20. []
21. ?*
22. ~[?*
23. \$[a]
24. ~\$[]
25. [w o r k | f i l l | p a c k]
[s | e d | i n g | []]
26. [b o o k %+Noun:0 %+Sg:0 |
b o o k %+Noun:0 %+Pl:s]
27. [h u n d | e l e f a n t | k a t]
[%+Aug:e 0:g | %+Dim:e 0:t | %+Fem:i 0:n]*
%+Noun:o (%+Pl:j) (%+Acc:n)
28. [b o o k | d o g | d u c k] %+Noun:0
[%+Sg:0 | %+Pl:s]
29. [a i m e:0 r:0] %+Verb:0 %+PresInd:0
[%+1P:e %+Sg:0 |
%+2P:e %+Sg:s |
%+3P:e %+Sg:0 |
%+1P:o %+Pl:n 0:s |
%+2P:e %+Pl:z |
%+3P:e %+Pl:n 0:t]

Write the regular-expression grammars and graph the networks for the following:

1. Write a grammar of 24-hour times, e.g. 06h23, 14h32, 21h00, 23:59. The exact format is somewhat language-dependent.
2. Write a grammar of telephone numbers for your country.
3. If you speak French, write the regular expression and graph the network for the following:

- Describe regular French future and conditional verbs, using lexical-level tags such as +Verb, +Cond, +Fut, +1P, +2P, +3P, +Sg and +Pl. The lexical side should show the infinitive plus tags.
 - Add some irregular cases, such as *devenir*, *être*, *aller* and *faire*.
4. Find other types of strings that have a predictable structure and write regular expressions to characterize them.

Appendix B

Suggested Analysis Tags

Contents

B.1	The Challenge of Tag Choice	586
B.2	Tag-Name Spelling	586
B.2.1	Punctuation Alternatives	586
	Initial Plus Sign	586
	Final Plus Sign	587
	Initial and Final Plus Sign	587
	Other Conventions	587
B.2.2	Principles for Tag-Name Punctuation	588
B.3	Principles for Tag-Name Choice	588
B.3.1	Three Directives	588
	The Prime Directive	588
	The Secondary Directive	588
	The Tertiary Directive	588
B.3.2	Don't Work in Isolation	588
B.3.3	Don't Panic	589
B.4	Suggested Tags to Choose From	589
B.4.1	Major Category Tags	589
B.4.2	Noun Distinctions	589
B.4.3	Verb Distinctions	593
B.4.4	Proper Noun Distinctions	594
B.4.5	Pronoun Distinctions	594
B.4.6	Adjective Distinctions	595
B.4.7	Number-Word Distinctions	595
B.4.8	Article/Determiner/Quantifier Distinctions	597
B.4.9	Adverbs	597
B.4.10	Title Distinctions	597

B.4.11 Punctuation	598
Subcategories of Punctuation	598
Handling Separated Punctuation in lexc	598
Delimiters in Lexical Strings	598
B.5 Miscellaneous Problematic Leftovers	599
B.5.1 Register Tags	599
B.5.2 +Emph	599
B.5.3 +NP	599
B.5.4 +Abbr	599
B.5.5 +Meas	600

B.1 The Challenge of Tag Choice

Xerox finite-state networks store feature information, including part-of-speech category, tense, aspect, mood, person, number and gender, in the form of multicharacter-symbol TAGS such as +Noun, +PresInd, +1P, +Sg, and +Masc. These tags are in fact just symbols, manipulated exactly like the alphabetic symbols *a*, *b*, and *c*, but they have multicharacter print names that hopefully mean something to the developers. There is nothing magic about the selection and spelling of the tags; they must all be chosen by the developers and declared, usually in the Multi-char.Symbols statement (see Section 4.3.4) of the **lexc** source file.

Because natural languages differ so much in the distinctions they make and in the terminology traditionally used to describe those distinctions, imposing rigid rules for defining multicharacter-symbol tags is both impossible and undesirable. However, there is at least *some* benefit to be had if reasonably similar languages adopt, where appropriate, the same tags to mark the same phenomena. If nothing else, this facilitates future maintenance by those who may have to move back and forth from one system to another.

The following guidelines and examples are offered as helpful suggestions.

B.2 Tag-Name Spelling

B.2.1 Punctuation Alternatives

Initial Plus Sign

In this book, most of the examples have employed tags spelled with an initial plus sign, followed by letters and digits, such as +Verb. When reading analysis strings, as in the following Portuguese example, this convention helps the human eye to identify and separate the tags.

```
Upper: cantar+Verb+PresInd+1P+Pl
Lower: cantamos
```

Final Plus Sign

The initial plus-sign convention works well for languages where the analysis strings begin with a baseform, like *cantar*, and continue with tags that map to surface suffixes. In a language with productive prefixation, however, it may be necessary or useful to map the surface prefixes directly to prefix tags, such that prefix tags will precede the baseform in the analysis string. As in the following Esperanto example, **Xerox** linguists often define prefix tags with names like `Neg+` that have a plus sign at the end. The tag `Neg+` maps to the surface prefix *ne*.

```
Upper: Neg+bon+Adj+Pl+Acc
Lower: nebonajn
```

Without the plus sign at the end, the analysis string would look like `Negbon+Adj+Pl+Acc`, with the boundary between the first tag and the stem *bon* visually unclear.

Initial and Final Plus Sign

In many systems, it may be desirable to define analysis strings that contain not only the baseform and tags, but also the lexical form of some or all of the various affixes. In such cases, it may be useful to define tags spelled like `+Neg+`, `+AdjRoot+`, `+AdjSuff+` and `+Pl+`, with a plus sign at the beginning and at the end. That would lead to analyses like this:

```
Upper: ne+Neg+bon+AdjRoot+a+AdjSuff+j+Pl+n+Acc
Lower: nebonajn
```

In general, it should be remembered that the lexical language is a creation of the developer (see Section 6.2.2), and should include whatever (morpho)phonological material, tags, and delimiters are required for readability and for support of the subsequent applications.

Other Conventions

Another possible and respectable convention is to spell all tags with surrounding square brackets, e.g. `[Noun]`, `[Sg]`, `[Pl]`, etc. That would lead to analyses that look like the following:

```
Upper: cantar[Verb][PresInd][1P][Pl]
Lower: cantamos
```

```
Upper: ne[Neg]bon[AdjRoot]a[AdjSuff]j[Pl]n[Acc]
Lower: nebonajn
```

B.2.2 Principles for Tag-Name Punctuation

Whichever convention is chosen, the following recommendations hold:

1. Always include a punctuation character in the spelling of each tagname. This can be used to help find cases where the linguist forgot to declare an intended tag (see Section 7.3.5).
2. Choose a spelling convention that visually separates affixes from baseforms.
3. Be consistent.

For the remainder of this appendix, we will assume that tags are spelled with an initial plus sign. The remaining questions are then how to choose the appropriate distinctions and how to spell the alphabetic part of the tags.

B.3 Principles for Tag-Name Choice

B.3.1 Three Directives

The Prime Directive

Choose morphological-analysis tags that are appropriate for indicating the distinctions in the language being analyzed. Do not try to force your language into a descriptive framework that is foreign to it. If there is an established linguistic vocabulary for describing your language, consider defining tags that evoke that vocabulary.

The Secondary Directive

Where a set of words act the same syntactically, i.e. where a set of words fit into the same syntactic frames, then they should probably be analyzed with the same tags. Where two words act differently in the syntax, they should probably be analyzed with distinct strings of tags. Use tags, and strings of tags, consistently.

The Tertiary Directive

Use the tags and tag orders which have already been used in related systems, unless this violates the Prime Directive or Secondary Directive.

B.3.2 Don't Work in Isolation

It is highly recommended that you *not* work in isolation when choosing tags and tag orders. Make a detailed plan as early as you can, selecting a preliminary tagset and defining tag orders, and present this plan to your colleagues for discussion and approval before you continue with further development. Formalize your tagset and

tag orders as a regular expression in a LEXICAL GRAMMAR (see Section 7.4.1) and use this lexical grammar periodically for testing. You will always need to refine your tagnames and tagset as you progress, but some thoughtful planning at the beginning can help make a more transparent and maintainable system.

B.3.3 Don't Panic

Do not get overly wrought about tag names; remember that

- The tags in the alphabet of a network are typically used and seen only by a few developers. Real customers will probably never see them at all.
- Tags are just multicharacter symbols that do not really mean anything, except as you define them and contrast them with other tags in your system.
- For your own convenience, you want to choose tag names that are informative and yet are not too long to type.
- It is absolutely trivial to change tag names cosmetically, using replace rules (see Section 3.5.5).

B.4 Suggested Tags to Choose From

If you have no good reason to use a different system, we present the following suggested tags to choose from.

B.4.1 Major Category Tags

Major-category tags, shown in Table B.1, correspond roughly to the categories of headwords in a printed dictionary, or to the kinds of distinctions needed for a part-of-speech tagger or a syntactic parser that will use the output of your morphological analyzer. There are good arguments for placing the major category tag either at the very beginning of the tag string or at the very end. The convention of putting them at the beginning has been used for Spanish, Portuguese, Dutch, Italian, Hungarian, Czech and other systems at **Xerox**.

B.4.2 Noun Distinctions

In many languages, nouns may be further distinguished as augmentative or diminutive, resulting in tag strings like +Noun+Aug and +Noun+Dim.

+Aug	augmentative
+Dim	diminutive

Gender distinctions are also very common.

+Noun	noun (<i>house</i>)
+Prop	proper noun (<i>John</i>)
+Verb	verb
+Adj	adjective
+Adv	adverb
+Pron	pronoun
+Art	article (like English <i>the</i> and <i>a</i>)
+Det	determiner (like <i>this</i> , <i>that</i> , <i>those</i>)
+Quant	quantifier (like <i>many</i> , <i>some</i>)
+Conj	conjunction (like <i>and</i> , <i>or</i> , <i>but</i>)
+Prep	preposition
+Title	monsieur, senhor, Herr
+Punc	punctuation mark
+Command	idiomatic orders, usually military: <i>march</i> , <i>fire</i> , <i>shoulder arms</i> , <i>attention</i> , <i>at ease</i> , <i>shut up</i> (often difficult to distinguish from the more general imperative verbs)
+Exclam	exclamations: <i>Damn</i> , <i>Merde</i> , <i>Ouch</i>
+Interj	interjections: <i>what</i> , <i>quoi</i> , <i>right</i>
+Onom	onomatopoeia: <i>bow-wow</i> , <i>au-au</i> , <i>gou-gou</i> , <i>cocoricó</i> , <i>cockle-doodle-do</i> , <i>croak</i> , <i>ribbit</i> , <i>vre ke kex koax koax</i> (what Greek frogs say)
+Num	number (usually alphabetically based)
+Dig	digit-based word
+Prt	particle, a hard to define class, sometimes used for question particles
+Initial	Upper-case letter followed by a period, e.g. <i>Q.</i> as in many names: <i>John Q. Citizen</i>
+Let	an individual letter appearing by itself, e.g. <i>q</i>

Table B.1: Suggested Major-Category Tags

+Nom	nominative case
+Gen	genitive
+Acc	accusative
+Abl	ablative (from)
+Ela	elative (out of)
+All	allative (to)
+Ess	essive (as)
+Par	partitive (part of)
+Com	comitative (with)
+Abe	abessive (without)
+Ine	inessive (in)
+Ill	illative (into)
+Ins	instructive (by)
+Ade	adessive (on)
+Tra	translative (change to)

Table B.2: Case Marking for Nouns

+Masc	masculine
+Fem	feminine
+Neut	neuter
+MF	masculine or feminine, within a system where most nouns are one or the other
+MN	masculine or neuter
+FN	feminine or neuter

Each language has to be handled on its own terms. Dutch required a much more complex gender-marking scheme to accommodate subtle Flemish distinctions (which have mostly been lost in Dutch proper). When in doubt, it is usually better to make distinctions; it is always easy to collapse them later.

Case markings in some languages, like Finnish, are particularly rich, functioning much like appended postpositions. Table B.2 is a list of case distinctions that are significant in at least some languages around the world.

While some languages do not mark nouns for number, others will distinguish singular, dual, plural and sometimes rarer types like paucal (“a few”). In languages like English, that almost always mark a noun as singular or plural, a special tag like +SP can be used to mark *sheep*, *deer*, *grouse* and a handful of others that do not change in the plural.

+Inf	infinitive
+PresInd	present indicative
+PresSubj	present subjunctive
+ImpInd	imperfect indicative
+ImpSubj	imperfect subjunctive
+FutInd	future indicative
+FutSubj	future subjunctive
+Cond	conditional
+Impv	imperative
+Perf	past perfect
+Pluperf	pluperfect
+Gerund	gerund
+PastPart	past participle

Table B.3: Portmanteau Tense/Aspect/Mood Distinctions on Verbs in Romance Languages

+Sg	singular
+Dual	dual
+Tri	triple
+Pauc	paucal
+Pl	plural
+SP	for English <i>sheep</i> , etc.

Noun tags have typically been ordered in the following way:

+Noun+Fem+Sg

+Noun+Dim+Masc+Pl

+Noun+Nom+Masc+Sg

If, in the grammatical tradition of your language, the features of a noun are rattled off in a particular order, e.g. “Masculine-Singular Nominative”, then you might best reflect that order in the tag sequences.

Finally, CLASSIFYING languages divide the whole universe of “things” into a set of morphologically and syntactically significant CLASSES that may be as simple as animate vs. inanimate or may extend, as in Navajo, to distinctions like animate erect vs. animate non-erect, inanimate contained vs. inanimate non-contained, inanimate contained rigid vs. inanimate contained non-rigid, etc. Wherever possible, the tags used to mark such classes should reflect the standard terminology used in the field. In Bantu languages the noun classes are standardly numbered, so the tags should obviously include those numbers, e.g. +C1, +C2, +C3, etc.

B.4.3 Verb Distinctions

In the Romance languages treated so far, tense, aspect and mood were combined into single tags reflecting the realities of the verbal paradigms, as shown in Table B.3. Portuguese has inflected infinitives, and so it needs a tag like +InfFlex to distinguish these from bare infinitives.

In other languages, and especially in agglutinating languages, it might make more sense to use two tags, e.g. +Pres+Ind or +Pres+Subj, separating the tense and mood, if that better reflects the morphotactic system of the language.

+Ind	indicative
+Subj	subjunctive
+Junc	junctive
+Juss	jussive
+Opt	optative
etc.	

You may also need individual aspect tags in addition to or instead of tense tags:

+Perf	perfect
+Imp	imperfect
+Cont	continuative
etc.	

You may also need a system of voice tags:

+Act	active
+Pass	passive
+Middle	middle
+Caus	causative
etc.	

And of course many languages make person distinctions in their verbs:

+1P	first person
+2P	second person
+3P	third person

In some languages, you will need to distinguish between “inclusive we” and “exclusive we” (perhaps +1P+Pl+Incl vs. +1P+Pl+Excl). Yet other languages will require marking +Neg, +Pos, focus, and other distinctions right in the verb itself. Again, the goal is to choose and consistently use tags that best express the significant morphological/syntactic distinctions in your language. It is impossible and undesirable to force all languages into a rigid system—you will have to decide what is significant and devise the most beautiful tag encoding that you can.

B.4.4 Proper Noun Distinctions

A language typically has many subtypes of proper nouns, and the distinctions can often be syntactically significant. For example, given names may typically come before family names, or vice versa, and names for countries, cities and geographical areas may require different syntactic structures.

Unless more distinctions are really needed for the syntax, the following proper-noun subtypes are suggested as a start. One might propose five primary distinctions:

+Prop+Giv	given or first names
+Prop+Fam	family names
+Prop+Place	any geographical/political name
+Prop+Org	any kind of organization, e.g. Xerox
+Prop+Misc	leftovers

Marking leftovers explicitly helps to identify and extract them later if becomes desirable to add more distinctions.

Within the place names, there are five sub-sub-divisions that have been used in some projects:

+Prop+Place+Country	countries
+Prop+Place+City	cities
+Prop+Place+Continent	continents
+Prop+Place+Region	departments, sub-states, provinces; especially in the countries where the language is spoken
+Prop+Place+Usastate	states of the USA
+Prop+Place+Misc	for anything left over

Depending on your language, it may be necessary to mark gender, case and number on proper nouns, e.g. *Asie+Prop+Place+Continent+Fem+Sg*. As usual, the goal is to choose tags that reflect real distinctions in your language and which may be necessary for a subsequent task such as tagging or parsing.

B.4.5 Pronoun Distinctions

Pronoun systems are very language-specific. Case tags are often useful for distinguishing different classes of pronouns.

+Nom	nominative
+Dat	dative
+Acc	accusative
etc.	

Other distinctions, like possessive, may be marked +Gen or +Poss:

+Gen	+Pron+Gen in a highly case-marked language
+Poss	+Pron+Poss for possessive pronouns in English

Yet other distinctions, like reflexive, may be significant in the pronoun system.

+Rel	relative
+Refl	reflexive
+Interrog	interrogative
+Dem	demonstrative
+PrepObj	post-prepositional

Pronouns are usually distinguished by person and number, parallel to verbs, as in +Pron+Nom+3P+Masc+Sg.

B.4.6 Adjective Distinctions

The following distinctions have proved useful:

+Comp	+Adj+Comp	comparative
+Sup	+Adj+Sup	superlative

Instead of +Sup, one linguist felt obliged to use +Int (for intensive). Augmentative/diminutive, gender, number and case tags are also appropriate in languages where the adjectives agree with the nouns they modify.

B.4.7 Number-Word Distinctions

Words representing numbers and number-related concepts come in many forms. The following distinctions have proved useful in many languages:

+Num	for alphabetic strings representing numbers
+Dig	for digit-based string
+Rom	for Roman numerals

Subtypes may include the following:

+Num+Card	for cardinal numbers <i>one</i> and <i>two</i>
+Num+Ord	for ordinal numbers <i>first</i> and <i>second</i>
+Dig+Card	for <i>1, 2, 3</i>
+Dig+Ord	for <i>1st, 2nd, 3rd</i>
+Rom+Card	for <i>I, II, III, IV</i>
+Rom+Ord	for <i>Ie, Iie, IIIe</i>

The main-category tag +Num is typically used for strings of alphabetic characters, e.g. *first* might analyze as +Num+Ord. Number strings based on strings of digits, like *23*, are usually given the main-category tag +Dig with one or more qualifying tags. For digit-based strings, analyses like the following may be appropriate.

Upper: 1996+Dig+Year
Lower: 1996

Upper: 23%+Dig+Percent
Lower: 23%

Upper: 23°+Dig+Degree
Lower: 23°

Upper: 3.+Dig+Item
Lower: 3.

Upper: 3)+Dig+Item
Lower: 3)

Upper: 2nd+Dig+Ord
Lower: 2nd

Upper: 02/07/95+Dig+Date
Lower: 02/07/95

Upper: 555-3496+Dig+Tel
Lower: 555-3496

Upper: \$24+Dig+Curr
Lower: \$24

Upper: 12:24+Dig+Time
Lower: 12:24

Upper: 548-04-1579+Dig+ID
Lower: 548-04-1579

Upper: 38240+Dig+Postal
Lower: 38240

There are seldom any easy answers, and often some dilemmas, when it comes to tag selection. Ordinal numbers are very often marked for gender and number and act exactly like adjectives. In some cases, it might be best simply to treat them as adjectives.

B.4.8 Article/Determiner/Quantifier Distinctions

Articles, determiners and quantifiers will usually need to be subcategorized in various ways, e.g.

Upper: the+Art+Def
Lower: the

Upper: a+Art+Indef
Lower: a

Gender, number, case, etc. tags will be necessary in many languages.

B.4.9 Adverbs

In non-technical school grammars, all the little words that no one knows what to do with are lumped into a category of Adverbs, a very dangerous pseudo-class. Avoid that tendency, trying to make syntactically real distinctions where appropriate. Some adverbs might need only the +Adv tag for your purposes:

Upper: well+Adv
Lower: well

Others are obviously, and fairly productively, derived from adjectives, and this can be shown by convention with tag strings like the following, wherein \hat{DB} is a multicharacter symbol representing a derivational boundary.

Upper: claro+Adj \hat{DB} +Adv
Lower: claramente

Upper: claro+Adj+Sup \hat{DB} +Adv
Lower: clarissimamente

The intent of analysis strings like “claro+Adj \hat{DB} +Adv” is to identify the base-form as an adjective, i.e. something that will be marked as an adjective in a standard dictionary, while marking the whole word as an adverb.

The secondary tags +Dim, +Aug, +Interrog, +Manner, +Extent and +Neg have also been used with +Adv to distinguish various subtypes.

B.4.10 Title Distinctions

Titles may be distinguished by gender, number and other sub-tags.

+Title+Addr	for <i>Mr.</i> , <i>Mrs.</i> , <i>Dr.</i>
+Title+Post	for titles that come after a name, such as <i>Jr.</i> and <i>Esq.</i>

B.4.11 Punctuation

Subcategories of Punctuation

The punctuation tag (+Punc) is intended for analyzing input words that consist of a single punctuation symbol, including periods, commas and exclamation marks. They do show up isolated in texts, and some tokenizers will separate punctuation from words and feed the punctuation tokens separately to the morphological analyzer.

It was once thought useful to distinguish between beginning, middle and ending punctuation, where the positions refer to individual words.

+Punc+Beg	e.g. in English, the left single quote ‘, the left parenthesis, and other punctuation marks that typically come at the beginning of a word
+Punc+Mid	e.g. in English, underscore, hyphen and slash, as in <i>Multi-char_Symbols</i> , <i>twenty-five</i> , and <i>and/or</i>
+Punc+End	e.g. in English, the right single quote ’, the right parenthesis, etc.

This classification has been problematic. Of course, all the linguist can do is indicate the most likely positioning of the punctuation mark relative to a word.

Taggers seem to need different information, in particular they need to know if a punctuation mark typically terminates a whole sentence. In the end, no one really knows how best to analyze isolated punctuation marks within the context of a morphological analyzer. Consult with colleagues and any customers to determine the most useful encoding for subsequent applications.

Handling Separated Punctuation in lexc

As a purely practical matter it is usually best to handle normal alphabetic words in one **lexc** lexicon and individual punctuation marks in a separate **lexc** lexicon; the resulting networks can subsequently be unioned together to form the final lexical transducer. This provides a clean separation between real punctuation marks and the punctuation marks that are included in the spelling of multicharacter symbols. When examining the alphabet of the alphabetic-word network, the presence of punctuation characters can be a useful clue to tracking down undeclared multicharacter symbols (see Section 7.3.5).

Delimiters in Lexical Strings

If you need to insert delimiters in your lexical strings, which might include compound boundaries, the boundaries between the root and the affixes, derivational boundaries, etc., use multicharacter symbols for this purpose. Do not use ordinary punctuation symbols as delimiters in lexical strings.

B.5 Miscellaneous Problematic Leftovers

Every language will provide new challenges for defining a tagset. Here are a few left-over tags that have proved helpful, or troublesome, in the past.

B.5.1 Register Tags

Languages always have a cline of registers, ranging from the vulgar and informal to the formal and courtly. Morphological analyzers are typically written to handle properly spelled text produced by educated native speakers of the standard dialect. However, there is usually a need to handle at least some informal, dialectal, and even vulgar words that occur in texts, and yet to be able to distinguish them from the standard words. The following tags, usually added at the end of otherwise normal tag strings, have been used in some systems.

+Slang	signals that the surface form is popular or informal
+NonStd	signals that the surface form is from a non-standard dialect
+Vulgar	signals that the surface form is vulgar
+Off	signals that the surface form is offensive or not politically correct
etc.	

B.5.2 +Emph

To emphasize written words, Dutch can just add acute accents to words that are normally written without accents—such accented forms can be distinguished by adding an extra +Emph tag on the end of the normal tags.

B.5.3 +NP

The problematic tag +NP was used in a **Xerox** Portuguese analyzer to mark some proper nouns that act as a full noun phrase and resist the addition of the definite article. Unlike in English, many Portuguese and Catalan proper nouns can take a definite article, e.g. Portuguese *O Jorge* (“(the) George”) or Catalan *La Maria* (“(the) Mary”).

B.5.4 +Abbr

There is a certain tendency for beginning developers to mark all abbreviations as just +Abbr or something similar, as if it were a part-of-speech. This is of course useless for subsequent processes like tagging that need to know how words behave syntactically. If a tag like +Abbr is used, then it needs to be added, perhaps at the end, to a more normal set of tags that indicates the major part-of-speech category

and other standard distinctions. Your analyzer might handle upper:lower string pairs like the following:

Upper: IBM+Prop+Org+Sg+Abbr
Lower: IBM

Upper: RSVP+Command+Abbr
Lower: RSVP

Upper: TGIF+Interj+Abbr
Lower: TGIF

Upper: v.+Adv+Extent+Abbr
Lower: v.

There is a never-resolved question whether the lexical side should spell out the abbreviation, mapping, e.g., *TGIF* to *Thank God It's Friday* or *v.* to *very*. The **lexc** grammar that handles abbreviations can of course be written either way.

B.5.5 +Meas

+Meas is a problematic tag that has been used to mark various measure terms, e.g. *km*, *cm*, *k* (for kilo), etc. These might be better marked +Noun . . . +Abbr with other appropriate tags between +Noun and +Abbr reflecting how the words behave syntactically. If +Meas is used, it might be subdivided into +Meas-+Distance, +Meas+Size, +Meas+Weight, +Meas+Temp, etc.

Appendix C

Makefiles

Contents

C.1 Introduction	601
C.1.1 The Goal of this Appendix	601
C.1.2 What do Makefiles Do?	602
C.1.3 Makefiles are Another Formalism to Master	602
C.2 Example: A Simple Morphological Analyzer	603
C.2.1 Two Source Files	603
C.2.2 Building the Lexical Transducer by Hand	603
C.2.3 Building the Lexical Transducer with a Makefile	604
C.3 Example: Separating Source, Intermediate, and Binary Files	609
C.4 twolc and Makefiles	611
C.5 quit-on-fail	612
C.6 Conclusion	612

C.1 Introduction

C.1.1 The Goal of this Appendix

This appendix illustrates how to use makefiles to help build and maintain your applications. The examples assume a Unix-like operating system, but a free version of the **make** application is also available for Windows from the Cygwin Project.¹

We will assume, for illustration, that the application is a straightforward finite-state morphological analyzer, compiled from **lexc** and **xfst** sources into a single lexical transducer, but makefiles can be valuable or even vital in most development projects.

¹See <http://sources.redhat.com/cygwin/>. For improved Unix-like look and feel, this Windows version of **make** also comes with other tools (if you ask for them in the install process) like **sed** and **sort** that can facilitate writing makefiles.

C.1.2 What do Makefiles Do?

Makefiles automate and optimize the process of building your source files into a lexical transducer. They are Good Things.

In any non-trivial morphological analyzer, you will have a number of source files (i.e. files that you edit) that may be wordlists, **lexc** source files, **xfst** regular-expression files, **xfst** scripts or **twolc** source files. To build (or rebuild) the morphological analyzer, you typically need to call **lexc**, **xfst** and perhaps also **twolc**, possibly multiple times, to compile the source files into finite-state networks, and then to combine those networks together via composition, union and other finite-state operations to create the final lexical transducer.

Different files depend on each other. If you edit a source file, then it will need to be recompiled into a network, and every part of the final lexical transducer that depends on that new network will need to be rebuilt as well, often in a very critical order. It should be obvious that recompiling any non-trivial system by hand, relying on the developer's memory of which source files have been changed and how the overall system fits together, is problematic.

Makefiles automatically keep track of which files depend on which other files, and they can call **lexc**, **xfst**, **twolc** and Unix command-line utilities as necessary to rebuild the final lexical transducer. If your makefile is properly written, then you can edit one or more of the source files, enter the Unix command **make**, and the makefile will take care of everything, invoking all necessary operations in the correct order, and not recompiling anything that doesn't need to be recompiled.

```
unix> make
```

If nothing has changed, and everything is up to date, then a call to **make** simply prints out a message that all is well.

The GNU version of **make** is the one we use, and on our system it happens to reside in `/opt/gnu/bin/make`. The alias **gmake** is often defined to invoke the GNU version of **make**, wherever it might be stored.

```
unix> gmake
```

The examples below will use **gmake** and the GNU flavor of makefile syntax.

C.1.3 Makefiles are Another Formalism to Master

As valuable as makefiles are, they require the developer to master yet another formalism, and this is why we avoided introducing makefiles in the body of the book. Makefiles and the associated **make** or **gmake** application also vary subtly on different systems, and you may well need to consult your local systems-programmer gurus to translate the following examples to work in your environment. Almost everyone needs help getting started with makefiles, but once you have a working example to study, it is relatively easy to copy, edit and expand it as your project progresses.

C.2 Example: A Simple Morphological Analyzer

C.2.1 Two Source Files

We will begin by assuming that the application is a morphological analyzer for Mylang (the mythical “my language”), and that there is one **lexc** source file called `mylang-lex.txt`, containing the lexicon and morphotactic description, and a regular-expression file called `mylang-rules.regex` containing alternation rules. As shown in Figure C.1, the **lexc** source file is to be compiled into a network using **lexc**, and the alternation rules are to be compiled into a network using the **xfst** regular-expression compiler. The resulting rule network is then composed on the lower side of the lexicon network to form the final lexical transducer.

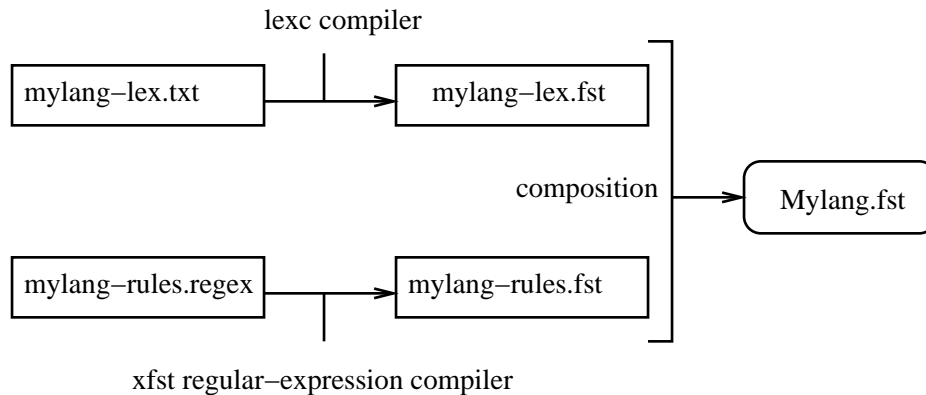


Figure C.1: Outline of a Simple Morphological Analyzer. The source files `mylang-lex.txt` and `mylang-rules.regex` are compiled by **lexc** and **xfst** respectively into intermediate networks `mylang-lex.fst` and `mylang-rules.fst`, which are then combined by composition to form the final lexical transducer `Mylang.fst`.

C.2.2 Building the Lexical Transducer by Hand

Before we dive into a makefile, let’s review how such a system would be built or rebuilt by hand. The developer might first invoke **xfst**, entering the command `compile-source mylang-lex.txt` followed by `save-source mylang-lex.fst`, followed by `quit`.

```

unix> lexc
lexc> compile-source mylang-lex.txt
lexc> save-source mylang-lex.fst
lexc> quit
unix>
  
```

Assuming that there were no errors, the compiled lexicon network will be stored in the file `mylang-lex.fst`. The name of the output file could of course be

any string, but giving it the same basic name as the source file, but with the `.fst` extension, is a useful convention.

To compile the rules, the developer would then invoke `xfst`, entering the commands `read regex < mylang-rules.regex`, `save stack mylang-rules.fst` and finally `quit`.

```
unix> xfst
xfst[0]: read regex < mylang-rules.regex
xfst[1]: save stack mylang-rules.fst
xfst[1]: quit
unix>
```

Again, the saving of the result as `mylang-rules.fst` is just a useful mnemonic convention.

Finally, with the two component networks already compiled and saved to file, we can invoke `xfst` a second time to compose them together into the final lexical transducer, which we'll save to file as `Mylang.fst`.

```
unix> xfst
xfst[0]: read regex @"mylang-lex.fst"
.o.
@"mylang-rules.fst" ;
xfst[1]: save stack Mylang.fst
xfst[1]: quit
unix>
```

These same `xfst` and `lexc` commands will be built into the makefile that we are about to write.

Canny readers will have noticed that it is not absolutely necessary to save `mylang-rules.fst` and invoke `xfst` a second time. However, breaking the task deliberately into such discrete parts shows the steps more cleanly and translates into a better makefile.

Note that if we subsequently edit the `lexc` file, but not the rules, then the rule file does not have to be recompiled at all. Similarly, if the rule file is edited, but not the `lexc` file, then the recompilation of the `lexc` file would just be a waste of time. In a large system, some of the processing steps may be very time-consuming, so the ability of a well-written makefile to avoid unnecessary recompilations can be a big time-saver. Editing either source file will always require its own recompilation and the recompilation of the final lexical transducer. The order is of course vital: the recompilation of the final lexical transducer must be done after the component files have been recompiled.

C.2.3 Building the Lexical Transducer with a Makefile

The alternative to building the lexical transducer by hand is to write a makefile that reflects all the proper dependencies and invokes all the necessary operations, in the

correct order. Syntactically, the makefile is just a plain text file composed of goals built on the following template.

```
goal: subgoals ...
      unix-command
      ...
```

The *goal* is very often the name of a file to be produced, and the *subgoals* are the names of other files or goals that the current goal is dependent on. When a goal is invoked, **gmake** first checks its subgoals to see if they are up to date. If they are indeed all up to date, then the current goal is considered up to date as well, and the commands in the body of the rule are not performed. It is up to the makefile writer to make sure that the dependencies are properly specified.

If, however, any subgoal is not up to date, then it is first brought up to date, and the current goal is then recomputed by invoking the commands in the body of the goal. (Syntactically, each command in the body of the goal must be indented with a tab.) If the subgoal is the name of a source file, then the current goal will be recomputed if the goal file doesn't exist at all or if the source file has a more recent modification date than the goal file. In practice, this means that a goal will be recomputed if you have subsequently edited any of the source files it depends on.

The commands in the body of a goal must be indented with a TAB character, not spaces, and this is an invisible trap waiting to catch the unwary.

It's time to look at a real example. Here's one way to build `Mylang.fst` in a GNU makefile, which is named literally `Makefile` and should reside in the same directory as the two source files.

```
# The First (Default) Goal is to build Mylang.fst
# This goal depends on mylang-lex.fst and
# mylang-rules.fst being up to date
```

```
Mylang.fst: mylang-lex.fst mylang-rules.fst
    @echo
    @echo "*** Building Mylang.fst ***" ;
    @echo
    @printf "read regex @"mylang-lex.fst" \
    .o. @"mylang-rules.fst" ; \n\
    save stack Mylang.fst \n\
    quit \n" > /tmp/mylang-script
    @xfst < /tmp/mylang-script
    @rm -f /tmp/mylang-script
```

```

# The Second Goal is to build mylang-lex.fst
# This goal depends on mylang-lex.txt, the lexc
# source file

mylang-lex.fst: mylang-lex.txt
    @echo
    @echo "*** Building mylang-lex.fst ***"
    @echo
    @printf "compile-source mylang-lex.txt \n\
save-source mylang-lex.fst \n\
quit \n" > /tmp/mylang-script
    @lexc < /tmp/mylang-script
    @rm -f /tmp/mylang-script

# The Third Goal is to build mylang-rules.fst
# This goal depends on mylang-rules.regex, the
# regular-expression source file

mylang-rules.fst: mylang-rules.regex
    @echo
    @echo "*** Building mylang-rules.fst ***"
    @echo
    @printf "read regex < mylang-rules.regex \n\
save stack mylang-rules.fst \n\
quit \n" > /tmp/mylang-script
    @xfst < /tmp/mylang-script
    @rm -f /tmp/mylang-script

# Finally, the clean goal can be invoked manually to
# erase all the non-source files. As in this case,
# the goal name does not have to be a filename.

clean:
    @rm -f *.fst

```

The goals in a makefile can be invoked individually by giving their names as arguments to **gmake**, e.g.

```

unix> gmake clean
unix> gmake mylang-rules.fst

```

If **gmake** is invoked without any overt argument, then the topmost goal in the file, here `Mylang.fst`, is the default; if the makefile is properly written, it will invoke all the other subgoals as needed.

As we saw above in the manual method, we cannot build `Mylang.fst` until `mylang-lex.fst` and `mylang-rules.fst` are up to date, and so this first goal lists them appropriately as subgoals.

```
# The First (Default) Goal is to build Mylang.fst
# This goal depends on mylang-lex.fst and
# mylang-rules.fst being up to date

Mylang.fst: mylang-lex.fst mylang-rules.fst
    @echo
    @echo "*** Building Mylang.fst ***" ;
    @echo
    @printf "read regex @"mylang-lex.fst" \
.o. @"mylang-rules.fst" ; \n\
save stack Mylang.fst \n\
quit \n" > /tmp/mylang-script
@xfst < /tmp/mylang-script
@rm -f /tmp/mylang-script
```

Looking more closely at the text just before the first goal, the pound-sign simply introduces comments that continue to the end-of-line. In the body of the goal, the **echo** statements are standard Unix commands that echo their string argument to the terminal. The first three commands therefore print newlines and a message that helpfully informs us of the progress of the makefile. The fourth command, **printf**, writes its string argument to `/tmp/mylang-script`. This string represents three commands to be performed by **xfst**, and in the makefile it includes the backslash to insert literal double-quotes and newlines (`\n`), and to break the string argument over multiple lines for legibility. The `/tmp` directory into which **printf** writes is accessible to all users and is usually an appropriate place to store such a temporary script; consult your gurus for the local policy. The name of the script file, here `mylang-script`, is chosen by the user.

The fifth command calls **xfst**, and the left angle-bracket tells **xfst** to read its commands from the same script file that we just created. Finally, a call to the standard Unix **rm** command removes the temporary script. All of the Unix commands in this example are preceded with the at-sign (`@`), which is optional and suppresses the echoing of the Unix commands themselves during execution of the makefile.

In some environments, you can use the **echo** utility instead of **printf** to write the commands into the temporary script file. However, some flavors of Unix shell, in particular the C shell, the Korn shell and the Bourne shell, have built-in **echo** commands that do not recognize escaped characters. In such systems, you should stick with **printf** or invoke a different version of **echo**, typically by specifying the explicit path `/usr/bin/echo`. Ask your local gurus how **echo** and **printf** work in your environment, and manually examine the files that they create when you start writing makefiles.

The goal that builds `mylang-lex.fst` is very similar to what we have seen in the first goal. In this case, the commands written to the temporary file are **lexc** commands, and the **lexc** application is of course invoked to execute them.

```
# The Second Goal is to build mylang-lex.fst
# This goal depends on mylang-lex.txt, the lexc
# source file

mylang-lex.fst: mylang-lex.txt
    @echo
    @echo "*** Building mylang-lex.fst ***"
    @echo
    @printf "compile-source mylang-lex.txt \n\
save-source mylang-lex.fst \n\
quit \n" > /tmp/mylang-script
    @lexc < /tmp/mylang-script
    @rm -f /tmp/mylang-script
```

The file `mylang-lex.fst` depends on the source file `mylang-lex.txt`, and **gmake** will rebuild `mylang-lex.fst` if and only if it doesn't exist at all or if the modification date of **mylang-lex.txt** is more recent.

The goal that builds `mylang-rules.fst` is more of the same, except that this time the goal depends on the source file `mylang-rules.regex` and again calls **xfst**. There are no surprises here.

```
# The Third Goal is to build mylang-rules.fst
# This goal depends on mylang-rules.regex, the
# regular-expression source file

mylang-rules.fst: mylang-rules.regex
    @echo
    @echo "*** Building mylang-rules.fst ***"
    @echo
```



```

@printf "read regex < mylang-rules.regex \n\
save stack mylang-rules.fst \n\
quit \n" > /tmp/mylang-script
@xfst < /tmp/mylang-script
@rm -f /tmp/mylang-script

```

C.3 Example: Separating Source, Intermediate, and Binary Files

Beginning developers usually mix source files, intermediate files and final-result “binary” files all together promiscuously in the same directory—expert developers learn to keep them in separate directories. In particular, keeping the source files in a dedicated source directory greatly facilitates backups, version control, creation of tar-files, etc.

A recommended directory structure for Mylang is shown in Figure C.2. The morph (morphology) directory under Mylang is just one of several daughters that may eventually appear, and it has three daughter directories: src, int, and bin. The source files and the makefile will be stored in Mylang/morph/src, intermediate files like `mylang-lex.fst` and `mylang-rules.fst` in Mylang/morph/int, and the final lexical transducer `Mylang.fst` will be stored in Mylang/morph/bin.

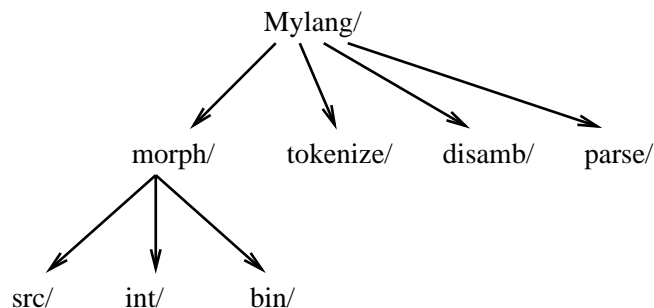


Figure C.2: It is recommended that source files be kept in a dedicated directory `src/`, intermediate files in `int/`, and final “binary” files in `bin/`.

While such a file structure may seem harder to work in, the hard parts can be written just once in the makefile. The following makefile is assumed to reside in the `morph/src/` directory. It creates and accesses the intermediate files in the sister `int/` directory, and it writes the final result in the sister `bin/` directory.

```

# Makefile Modified to Use src/, int/ and bin/

# The First (Default) Goal is named Mylang.fst;
# It has a subgoal ../bin/Mylang.fst
# This allows the user to type 'gmake Mylang.fst'

```

```
# rather than 'gmake ../bin/Mylang.fst'

Mylang.fst: ../bin/Mylang.fst

# This goal depends on the intermediate files
# mylang-lex.fst and mylang-rules.fst being
# up to date

../bin/Mylang.fst: ../int/mylang-lex.fst \
    ../int/mylang-rules.fst
    @echo
    @echo "*** Building Mylang.fst ***" ;
    @echo
    @printf "read regex @"../int/mylang-lex.fst\" \
    .o. \
    @"../int/mylang-rules.fst\" ; \n\
    save stack ../bin/Mylang.fst \n\
    quit \n" > /tmp/mylang-script
    @xfst < /tmp/mylang-script
    @rm -f /tmp/mylang-script

# The Second Goal is to build mylang-lex.fst
# This goal depends on mylang-lex.txt, the lexc
# source file, being up to date

mylang-tex.fst: ../int/mylang-lex.fst

../int/mylang-lex.fst: mylang-lex.txt
    @echo
    @echo "*** Building mylang-lex.fst ***"
    @echo
    @printf "compile-source mylang-lex.txt \n\
    save-source ../int/mylang-lex.fst \n\
    quit \n" > /tmp/mylang-script
    @lexc < /tmp/mylang-script
    @rm -f /tmp/mylang-script

# The Third Goal is to build mylang-rules.fst
# This goal depends on mylang-rules.regex, the
# regular-expression source file, being up to date

mylang-rules.fst: ../int/mylang-rules.fst

../int/mylang-rules.fst: mylang-rules.regex
```

```

@echo
@echo "*** Building mylang-rules.fst ***"
@echo
@printf "read regex < mylang-rules.regex \n\
save stack ../int/mylang-rules.fst \n\
quit \n" > /tmp/mylang-script
@xfst < /tmp/mylang-script
@rm -f /tmp/mylang-script

clean:
@rm -f ../bin/* ../int/*

```

Note in this version that the `clean` goal simply erases all the files in the intermediate and binary directories.

C.4 twolc and Makefiles

The **twolc** language is not much used these days at **Xerox**, having been largely replaced by the newer replace rules in **xfst**. But where **twolc** is still used, makefiles can of course contain goals that invoke the **twolc** compiler. Let us assume that the **twolc** source file is named `mytwolc-rules.txt` and that the rules are to be compiled and intersected, with the binary result saved to file as `mytwolc-rules.fst`. Done manually, the commands would be

```

unix> twolc
twolc> read-grammar mytwolc-rules.txt
twolc> compile
twolc> intersect
twolc> save-binary mytwolc-rules.fst
twolc> quit
unix>

```

Here is the same set of commands, recast as a makefile goal:

```

# Goal to compile and intersect a file of twolc rules

mytwolc-rules.fst: mytwolc-rules.txt
@echo
@echo "*** Updating mytwolc-rules.fst *** "
@echo
@printf "read-grammar mytwolc-rules.txt \n\
compile \n\
intersect \n\
save-binary mytwolc-rules.fst \n\

```

```
quit \n" > /tmp/mylang-script
@twolc < /tmp/mylang-script
@rm -f /tmp/mylang-script
```

C.5 quit-on-fail

When rebuilding large systems via makefiles, it may be desirable for **lexc** and **xfst** to quit when they encounter an error. **lexc** provides the switch **quit-on-fail**, **OFF** by default, which can be toggled **ON** (see Section 4.4.4), e.g.

```
mylang-lex.fst: mylang-lex.txt
@echo
@echo "*** Building mylang-lex.fst ***"
@echo
@printf "quit-on-fail \n\
compile-source mylang-lex.txt \n\
save-source ../int/mylang-lex.fst \n\
quit \n" > /tmp/mylang-script
@lexc < /tmp/mylang-script
@rm -f /tmp/mylang-script
```

Similarly, **xfst** provides an interface variable, also called **quit-on-fail**, that can be set to **ON** for the same effect, e.g.

```
mylang-rules.fst: mylang-rules.regex
@echo
@echo "*** Building mylang-rules.fst ***"
@echo
@printf "set quit-on-fail ON \n\
read regex < mylang-rules.regex \n\
save stack ../int/mylang-rules.fst \n\
quit \n" > /tmp/mylang-script
@xfst < /tmp/mylang-script
@rm -f /tmp/mylang-script
```

C.6 Conclusion

Makefiles are Good Things. Large morphological analyzers and similar applications may have dozens of different source files and very intricate interdependencies. But once you have created and debugged a makefile for your project, you should thereafter be able to edit your source files at will and simply invoke **gmake** to have the whole system rebuilt automatically.

Appendix D

Solutions to Exercises

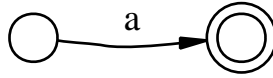
Contents

D.1	Graphing Quiz	614
D.2	The Better Cola Machine Diagram	621
D.3	The Better Cola Machine Network	621
D.4	Brazilian Portuguese Pronunciation	624
	D.4.1 Portuguese in xfst	624
	D.4.2 Portuguese Pronunciation in twolc	628
D.5	The Monish Language	631
	D.5.1 One Solution to Monish Analysis	631
	D.5.2 Monish Guesser-Analyzer	632
	D.5.3 Monish Rules in twolc	634
D.6	Tag Moving	635
	D.6.1 Tag Moving in a Regular Expression	635
	D.6.2 Tag Moving with twolc Rules and lexc Composition .	635
	D.6.3 Tag Moving with twolc Rules and xfst Composition .	637
D.7	Esperanto Nouns	637
D.8	Esperanto Adjectives	638
D.9	Esperanto Nouns and Adjectives	639
D.10	Esperanto Nouns and Adjectives with Upper-Side Tags . .	640
D.11	Esperanto Nouns, Adjectives and Verbs	641
D.12	Einstein II Problem	644

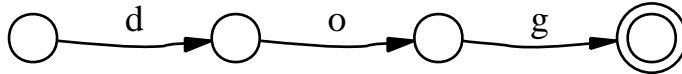
This appendix includes solutions to selected exercises. It must be noted that there are many possible solutions to each problem.

D.1 Graphing Quiz

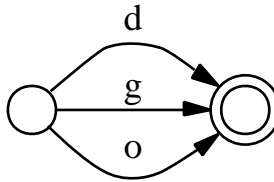
The graphing quiz is in Appendix A, page 581.



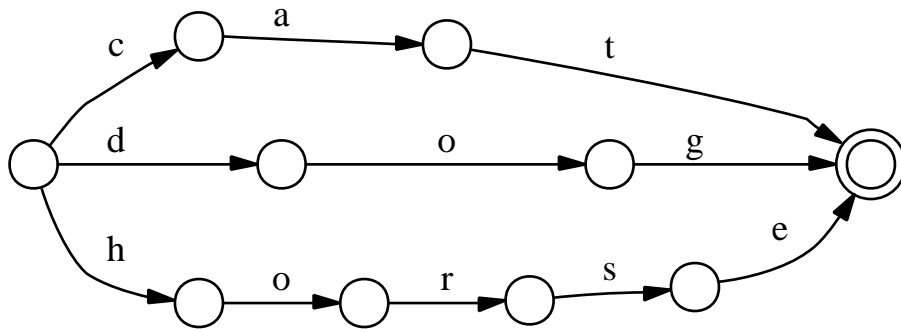
1. Network: a



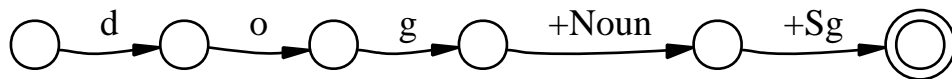
2. Network: d o g



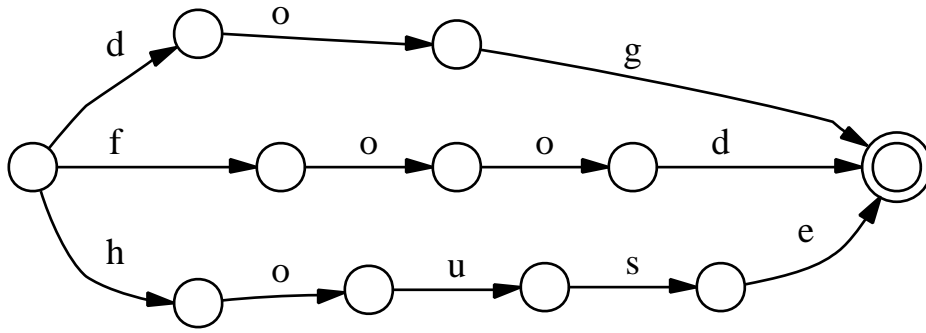
3. Network: [d | o | g]



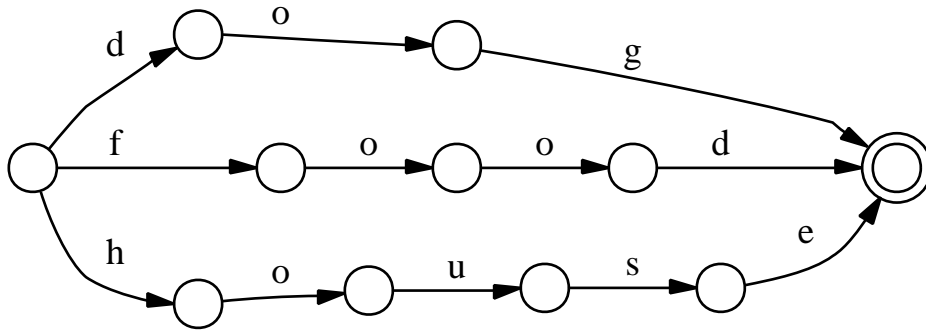
4. Network: [d o g | c a t | h o r s e]



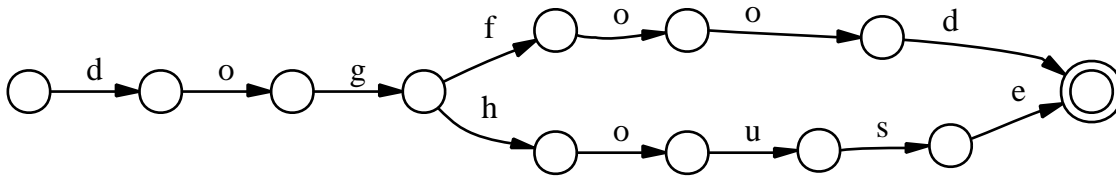
5. Network: [d o g % +Noun % +Sg]



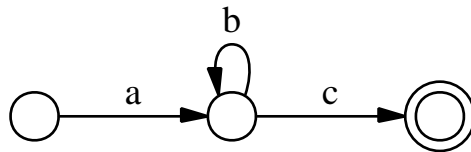
6. Network: [dog | house | food]



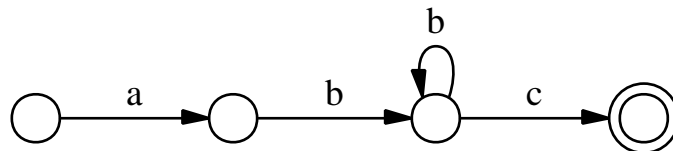
7. Network: [dog] | [house | food]



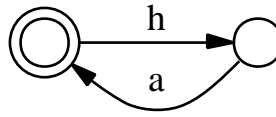
8. Network: [dog] [house | food]



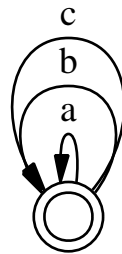
9. Network: a b* c



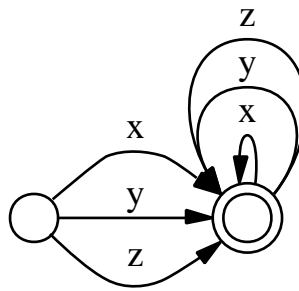
10. Network: a b+ c



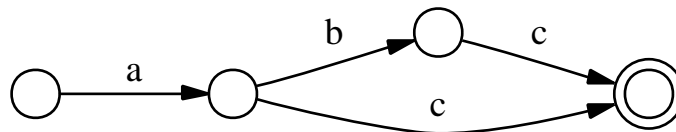
11. Network: $[h a]^*$



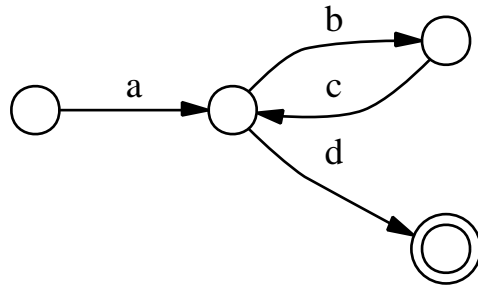
12. Network: $[a | b | c]^*$



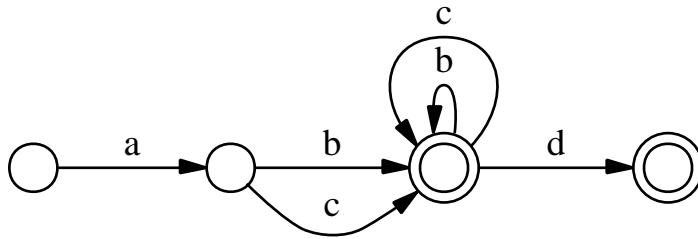
13. Network: $[x | y | z]^+$



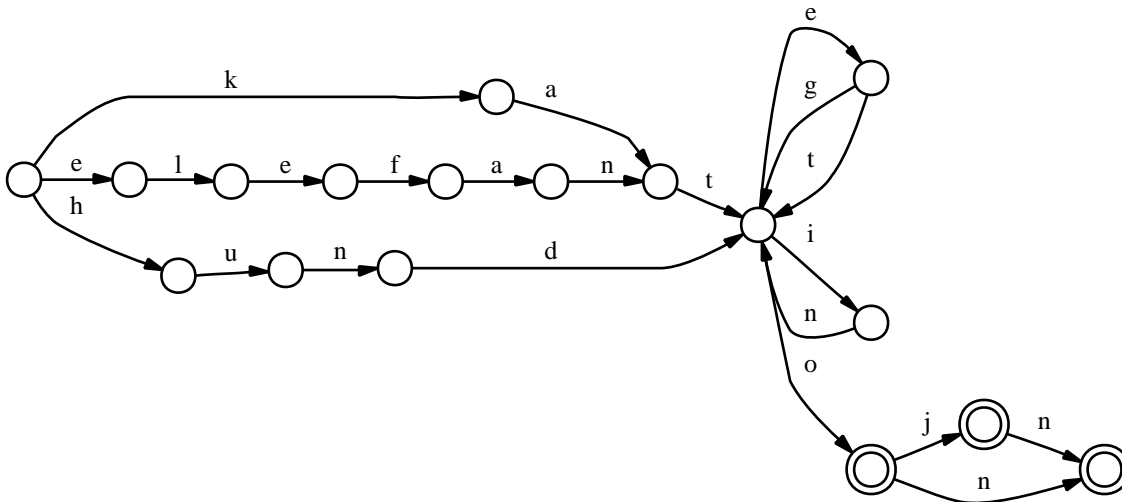
14. Network: $a(b c)$



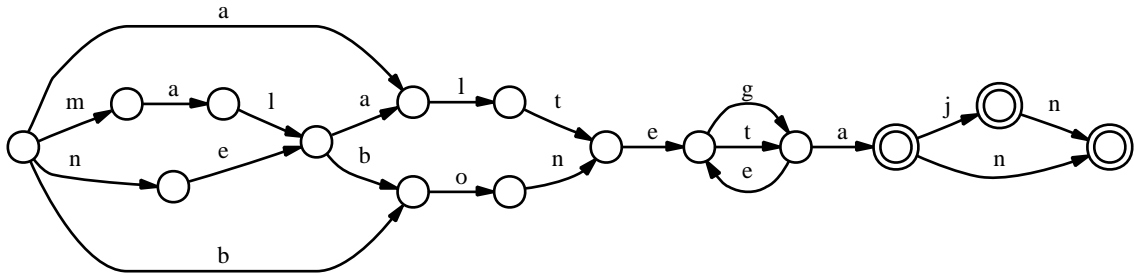
15. Network: $a [b c]^* d$



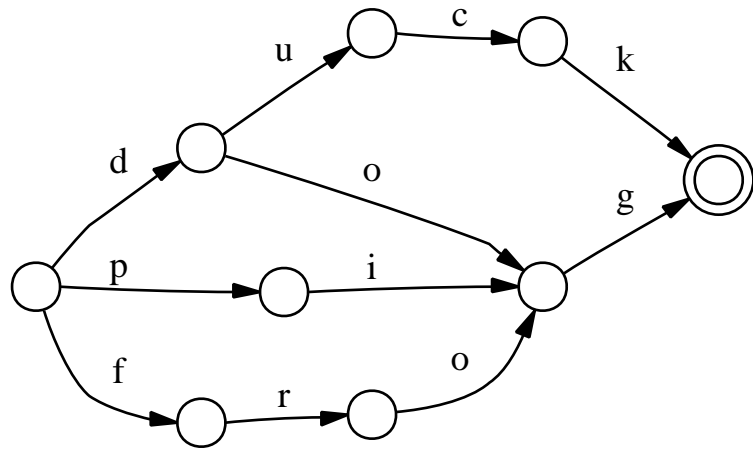
16. Network: $a [b | c]^+ (d)$



17. Network: $[hund | elephant | kat] [eg | et | in]^* o (j) (n)$



18. Network: $(m a l | n e) [b o n | a l t] [e g | e t] + a (j) (n)$



19. Network: $[d o g | p i g | d u c k | f r o g]$



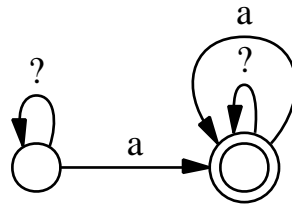
20. Network: $[\]$



21. Network: $?^*$



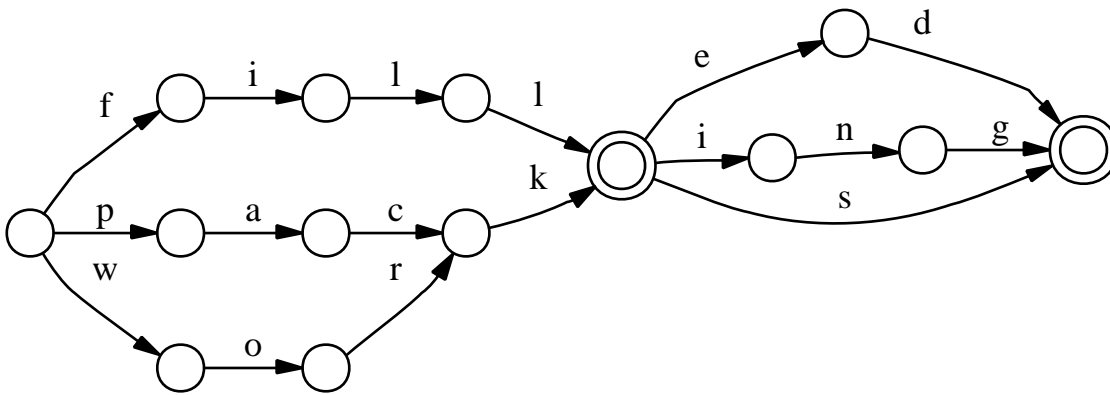
22. Network: $\sim [?^*]$



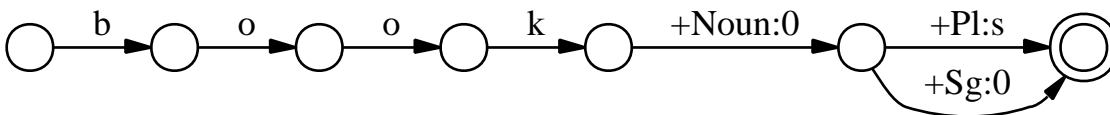
23. Network: $\{a\}$



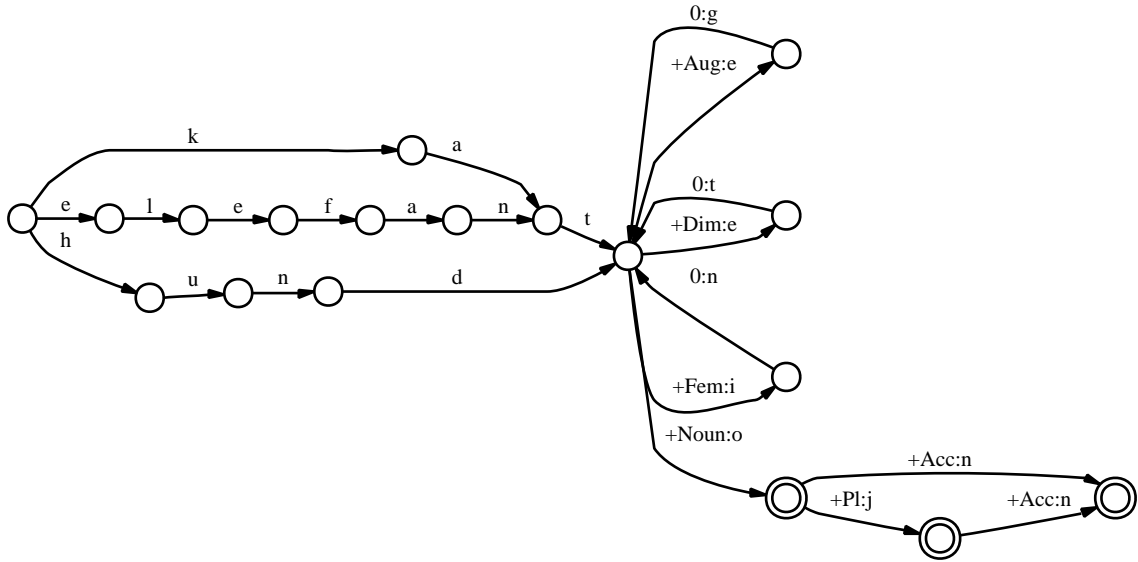
24. Network: $\sim\{\}$



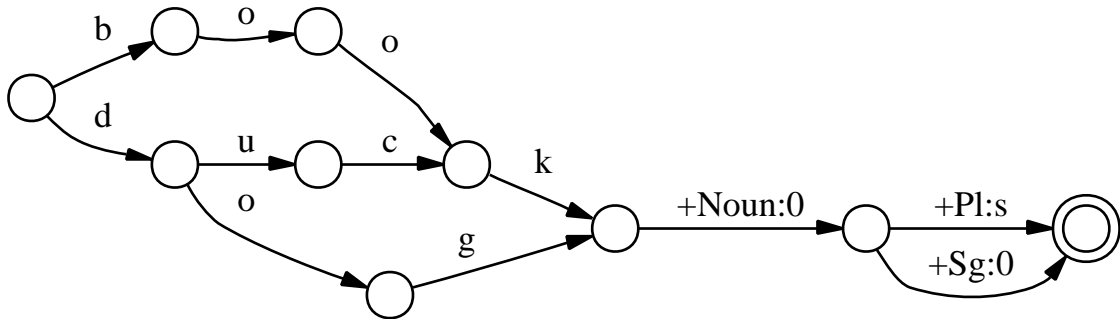
25. Network: $[work|fill|pack][s|ed|ing|]$



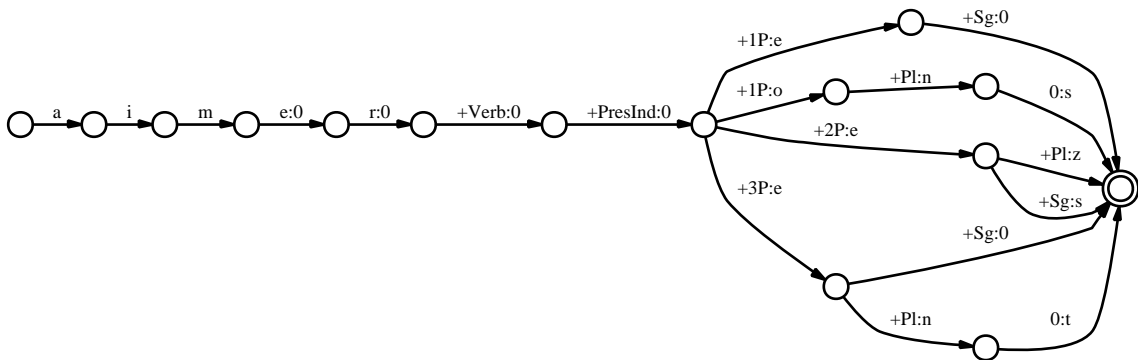
26. Network: $[book\%+Noun:0\%+Sg:0\ | \ book\%+Noun:0\%+Pl:s]$



27. Network: [h u n d | e l e f a n t | k a t] [%+Aug:e 0:g | %+Dim:e 0:t | %+Fem:i 0:n]* %+Noun:o (%+Pl:j) (%+Acc:n)



28. Network: [b o o k | d o g | d u c k] %+Noun:0 [%+Sg:0 | %+Pl:s]



29. Network: [a i m e:0 r:0] %+Verb:0 %+PresInd:0 [%+1P:e %+Sg:0 | %+2P:e %+Sg:s | %+3P:e %+Sg:0 | %+1P:o %+Pl:n 0:s | %+2P:e %+Pl:z |

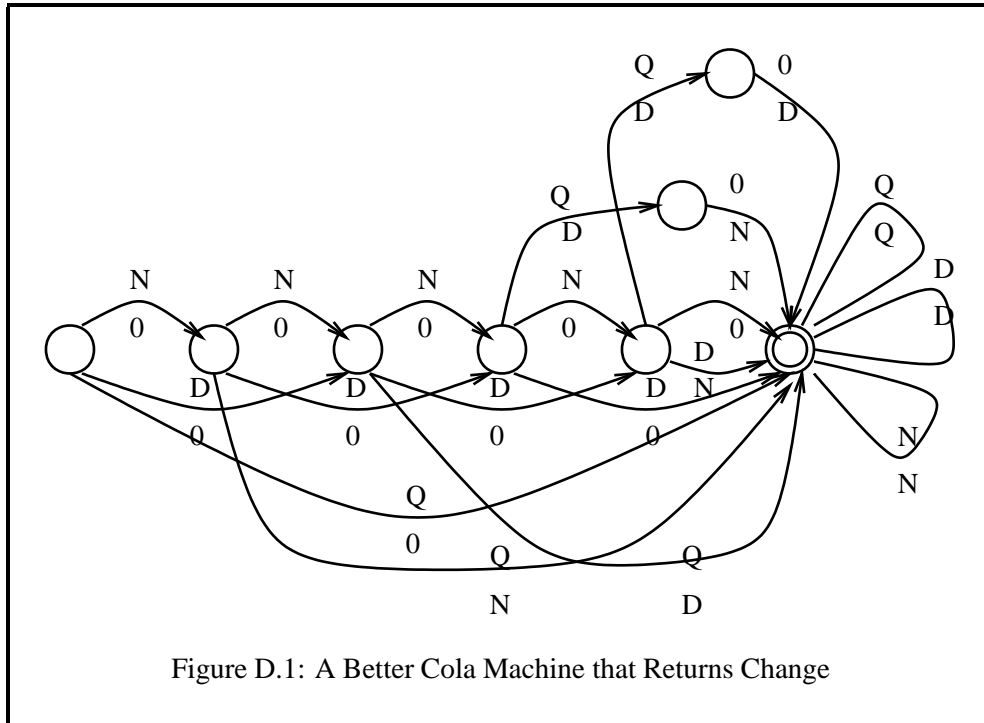


Figure D.1: A Better Cola Machine that Returns Change

`%+3P:e %+Pl:n 0:t]`

D.2 The Better Cola Machine Diagram

Figure D.1 shows one possible solution to the Better Cola Machine exercise in Section 1.10.1, on page 40. Note that this machine indicates the inputs (coins) on the upper side and the outputs (change) on the lower side of each arc.

D.3 The Better Cola Machine Network

In the Better Cola Machine exercise in Section 1.10.1, the reader is invited to construct by hand a transducer that accepts a sequence of nickels (**N**), dimes (**D**), and quarters (**Q**) and gives in return a cola for 25 cents. In Section 2.6.1, on page 79, the task is to construct a regular expression from which the transducer can be compiled automatically. We give the solution below to the latter exercise.

The upper language in the relation we want to describe consists of strings in which the three symbols may occur in any order: $[N|D|Q]^*$ and the value of the string must be exactly 25 cents. As we have already seen, this language can be defined by the regular expression $[[D \rightarrow N^2, Q \rightarrow N^5] \cdot \circ \cdot N^5] \cdot u$.

To define a transducer that maps such a string to “[COLA]”, we can just leave out the upper-side extraction operator $\cdot u$ and compose the relation with another

relation that maps a sequence of five nickels into a cola: $[N^5 \rightarrow "[COLA]"]$. The solution to the first task is given in Figure D.2.

```
[ D -> N^2, Q -> N^5 ]
      .o.
      N^5
      .o.
N^5 -> "[COLA]"
```

Figure D.2: First solution

If the value of the original coin sequence is worth more than five nickels, the transducer in Figure D.2 gives no output. To remedy this problem, we can simply relax the constraint that limits the value of the input string to five nickels. If we replaced the second component, $[N^5]$, by $[N^5]^*$, the resulting transducer would accept any sequence of coins whose value is some multiple of five nickels. But we choose the replacement N^* to allow strings such as “NNDDQNNN” that are worth more than one cola but do not necessarily translate into a multiple of five nickels. Figure D.3 shows the new solution.

```
[ D -> N^2, Q -> N^5 ]
      .o.
      N*
      .o.
N^5 @-> "[COLA]"
```

Figure D.3: Second solution

Note that we have to introduce the left-to-right, longest-match replace operator, $@->$, in Figure D.3 to guarantee a unique outcome in the case that the input string is worth more than one cola. Any extra money is returned to the user in the form of nickels.

As a favor to the user, let us make a little improvement. We will pay the refund using the smallest possible number of coins. For example, the overpayment of 20 cents will be returned as two dimes instead of four nickels. Figure D.4 presents the new solution. This regular expression compiles into a transducer of 12 states and 39 arcs. The effect of the N^* component in this cascade of compositions is that the upper language of the relation is restricted to strings that are made up of N s and the two other coin symbols that are mapped to N . If the N^* component is removed,

```

[ D -> N^2, Q -> N^5 ]
  .o.
  N*
  .o.
  N^5 @-> "[COLA]"
  .o.
  N^2 @-> D

```

Figure D.4: Third solution

the upper language of the resulting relation will be the universal language. Any unknown symbols will simply be mapped to themselves. For example, the upper-language input “QXQ” would be mapped to “[COLA]X[COLA]”, representing the return of two colas and the unknown coin. Perhaps this would be the best policy for real-life cola machines.

Allowing unknown symbols in the upper-language input gives rise to a new problem. We would like to map, say, “NDXD” to “X[COLA]”, recognizing that the nonadjacent coins “ND . . D” add up to five nickels. But the relation $[N^5 @-> "[COLA]"]$ in Figure D.4 presupposes an uninterrupted sequence of coins. It has to be modified. With the help of the *ignore* operator / we can describe strings that contain five nickels and possibly some other symbols. $[N^5/\backslash N]$ denotes the set of strings that contain exactly five Ns interspersed with any number of other symbols. Recall that $\backslash N$ denotes the term complement of N, all single symbol strings other than “N”.

```

[ D -> N^2, Q -> N^5 ]
  .o.
  N^5/\N @-> ... "[COLA]"
  .o.
  N -> 0 || - $"[COLA]"
  .o.
  N^2/\N @-> ... D
  .o.
  N -> 0 || - $D

```

Figure D.5: Final solution

Because the nickels should be mapped to a cola while the other symbols should

pass through unchanged, it is convenient to describe the nickels-to-cola mapping as a composition of two simple relations: $[N^5/\backslash N @-> \dots "[COLA]" \cdot \circ \cdot N -> 0 \ || \ - \$ "[COLA]"]$. The first part inserts a “[COLA]” suffix after a string that contains exactly five nickels possibly interrupted by other tokens. The second part deletes all the nickels that have been “consumed” in the process. A similar split has to be introduced in converting a sequence of two extra nickels to a dime because the two nickels need not be adjacent. Figure D.5 gives the final solution. The robust cola machine compiled from the expression in Figure D.5 contains 27 states and 116 arcs. It effectively counts the real coins in the input and returns the appropriate number of colas with any extra change due. All unknown coins are simply passed through unchanged. For example, for the input “XNDQYZDNWDN” the customer receives two colas, two dimes of change and the four unusable input symbols: “X[COLA]YZ[COLA]WDD”.

D.4 Brazilian Portuguese Pronunciation

D.4.1 Portuguese in xfst

The Brazilian Portuguese Pronunciation exercise is presented in Section 3.5.4, page 149. The first solution shown here uses an `xfst` script to define `Vowel`, which is used in the context of a rule, and then specifies the cascade of rules directly in one regular expression. Note especially the order of the replace rules—some of the relative ordering of rules is critical, and the rest of the relative ordering is arbitrary. An alternative solution, using a `define` statement for each rule, is shown below.

```
# Solution
# Southern Brazilian Portuguese Pronunciation
# using an xfst script file
# scripts are run from xfst using the 'source <filename>'
#      command

echo <<Defining Vowel>>

define Vowel [ a | e | i | o | u
               | á | é | í | ó | ú
               | â | ê |           ô
               | ã           | õ
               | à
               |           ü
               ] ;

echo <<Vowel is defined>>
echo <<Compiling the Rules>>

# the square brackets around the following rules are not
# technically necessary
```



```

read regex
[ s -> z || Vowel _ Vowel ] # this rule must come before
                              # the [ ç -> s ] rule and
                              # the [ s s -> s ] rule

.o.
[ ç -> s ]
.o.
[ c h -> %$ ] # a special case, h is part of the digraph 'ch'
              # such special cases need to be
              # handled "first"

.o.
[ c -> s || _ [ e | i | é | í | ê ] ]
              # c -> s is another special case, reflecting
              # historical phonological changes as Latin
              # evolved into Portuguese

.o.
[ c -> k ] # the default--Latin c always represented /k/
.o.
[ s s -> s ]
.o.
[ n h -> N ] # special case, h is part of the digraph 'nh'
.o.
[ l h -> L ] # special case, h is part of the digraph 'lh'
.o.
[ h -> 0 ] # all other hs are silent
.o.
[ r r -> R ]
.o.
[ r -> R || .#. _ ]
.o.
[ e -> i || _ (s) .#. , .#. p _ r ]

# this e -> i rule must apply before the d -> J and
# the t -> C rules

.o.
[ o -> u || _ (s) .#. ]
.o.
[ d -> J || _ [ i | í ] ]
.o.
[ t -> C || _ [ i | í ] ]

# the two rules above could be abbreviated equivalently
# as one
# [ d -> J, t -> C || _ [ i | í ] ];
# they must be ordered after the e -> i rule

.o.

```

```
[ z -> s || _ .# . ]
;

echo <<Successful Completion>>
echo <<The Network is on Top of the Stack>>
```

The following alternative solution defines a variable for each individual rule and then composes the variable values together. This approach can facilitate any necessary reordering of the rules during testing.

```
# Alternative Solution
# Southern Brazilian Portuguese Pronunciation
# using an xfst script file;
# scripts are run from xfst using the 'source <filename>'
#      command

echo <<Defining Vowel>>

define Vowel [ a | e | i | o | u
               | á | é | í | ó | ú
               | â | ê |      | ô
               | ã |      |      | õ
               | à
               |      |      |      | ü
               ] ;

echo <<Vowel is defined>>
echo <<Compiling the Rules>>

echo <<Defining Individual Rules>>

define Rule1 [ s -> z || Vowel _ Vowel ] ;
               # this rule must come before
               # the [ ç -> s ] rule and
               # the [ s s -> s ] rule

define Rule2 [ ç -> s ] ;

define Rule3 [ c h -> %$ ] ;
               # a special case, h is part of the digraph 'ch'
               # such special cases need to be
               # handled "first"

define Rule4 [ c -> s || _ [ e | i | é | í | ê ] ] ;
               # c -> s is another special case, reflecting
               # historical phonological changes as Latin
               # evolved into Portuguese
```

```

define Rule5 [ c -> k ] ;
    # the default--Latin c always represented /k/

define Rule6 [ s s -> s ] ;

define Rule7 [ n h -> N ] ;
    # a special case, h is part of the digraph 'nh'

define Rule8 [ l h -> L ] ;
    # a special case, h is part of the digraph 'lh'

define Rule9 [ h -> 0 ] ;
    # all other hs are silent
    # (no phonological realization)

define Rule10 [ r r -> R ] ;

define Rule11 [ r -> R || .#. _ ] ;

define Rule12 [ e -> i || _ (s) .#. , .#. p _ r ] ;
    # this e -> i rule must apply before the d -> J and
    # the t -> C rules

define Rule13 [ o -> u || _ (s) .#. ] ;

define Rule14 [ d -> J || _ [ i | í ] ] ;

define Rule15 [ t -> C || _ [ i | í ] ] ;
    # the two rules above could be abbreviated
    # equivalently as one
    # [ d -> J, t -> C || _ [ i | í ] ]

define Rule16 [ z -> s || _ .#. ] ;

# now that all the rules are individually defined, they
# can be composed together in the proper order

read regex Rule1 .o. Rule2 .o. Rule3 .o. Rule4 .o. Rule5 .o.
    Rule6 .o. Rule7 .o. Rule8 .o. Rule9 .o. Rule10 .o.
    Rule11 .o. Rule12 .o. Rule13 .o. Rule14 .o.
    Rule15 .o. Rule16 ;

# the advantage of such an approach is that any necessary
# editing of the order of the individual rules during
# testing is easier

echo <<Successful Completion>>
echo <<The Network is Left on Top of the Stack>>

```

D.4.2 Portuguese Pronunciation in twolc

The Portuguese Pronunciation exercise, using **twolc** rules, is proposed on page 334. Here is one possible **twolc** solution.

```

Alphabet h:0 ç:s c:s k t d l s z r n
      a e i o u
      á é í ó ú
      â     ô
      ã     õ
      à
      ü ;
! c appears only on the upper side, no c:c possible
! h is always realized as 0
! ç is always realized as s

Sets
      Vowel = a e i o u
      á é í ó ú
      â     ô
      ã     õ
      à
      ü ;
! all symbols that appear in Sets must be overtly
! declared in the Alphabet section;
! this is a 'feature' of twolc

Rules

"e:i"
e:i <=> _ (s) .# . ;
      .#. p _ r ;

! note that the e:i rule has two contexts, each
! terminated with a semicolon; the syntax and
! semantics of twolc rules are different from
! the syntax and semantics of the replace rules
! in xfst

"o:u"
o:u <=> _ (s) .# . ;

"c:k"
c:k <=> _ [ a | o: | u |
      á | ó | ú | â | ô | ã | õ | à | ü ] ;

! the right context o: specifies only the upper
! side because the rule just above maps some
! non-stressed final o vowels to u. A right

```

```
! context of just o would be interpreted as o:o,
! which is too specific for the rule to work
! correctly on examples like
! Upper:  oco
! Lower:  oku
```

```
"h digraphs"
```

```
[ l:L | n:N | c:%$ ] <=> _ h: ;
```

```
! L, N and $ appear only on the surface ;
! h never appears on the surface. The Alphabet
! specifies h:0, but not h, and so twolc will
! therefore assume (correctly) that h appears
! only on the upper side
```

```
"palatalization"
```

```
[ t:C | d:J ] <=> _ [ :i | :í ] ;
```

```
! C and J appear only on the surface.
! This palatalization occurs when the following
! surface vowel is /i/, hence the notation :i and :í
```

```
"ss"
```

```
s:0 <=> s _ ;
```

```
"s:z"
```

```
s:z <=> Vowel: _ Vowel: ;
```

```
! There are three possible realizations for s, being
! s:0, s:z and s:s
! This requires two rules and the declaration of
! s (i.e. s:s) in the Alphabet.
! With rules defined to control the s:0 and s:z cases,
! the s:s realization is the traditional 'elsewhere'
! case.
```

```
"z:s"
```

```
z:s <=> _ .#. ;
```

```
! There are two possible realizations for z, being
! z:s and z:z
! This requires one rule and the declaration of
! z (i.e. z:z) in the Alphabet.
! With the z:s rule defined, the z:z realization
! is the traditional 'elsewhere' case.
```

```
"r:R"
```

```
r:R <=> .#. _ ;
      _ r: ;
```

```

! R appears only on the surface

"r:0"
r:0 <=> r: _ ;

! There are three possible realizations for r
!   r:R, r:0 and r:r.

! end of twolc source file

```

Assuming that the source file is in `port-pronun-twolc.txt` the following commands should be invoked to compile the rules.

```

twolc> read-grammar port-pronun-twolc.txt
twolc> compile

```

The **read-grammar** command reads in the source file and checks for purely syntactic errors, printing any necessary warnings and error messages. After any necessary correction, when **read-grammar** reads the file cleanly, then the **compile** command should be invoked to compile the rules. During the compilation of the grammar shown above, the compiler will print the following semantic warning:

```

*** Warning: Unresolved <= conflict with respect
to 'r:R' vs. 'r:0'
   between "r:R"
   and "r:0"

**** Conflicting pair(s) of contexts:
   _ r: ;
   r: _ ;
Left context example:   r
Right context example: r

```

The warning identifies the two conflicting rules `r:R` and `r:0` by their unique names, as indicated in the source file, and it identifies the specific contexts that are in left-arrow conflict. The first context, `_ r:`, has the universal relation as the left context and `r:` as the right context. The second context, `r: _`, has `r:` as the left context and the universal relation as the right context. As the rule compiler clearly shows, the rules will be in conflict when the lexical context is `r` on the left and `r` on the right.

In other words, the two rules conflict for the input word “`rrr`”, or for any input word containing the substring “`rrr`”; for such a word, there will be no output. In this case, the conflict doesn’t matter because the sequence “`rrr`” never occurs in properly spelled Portuguese words. So ignore the warning in this case. (See Section 5.5 for ways to resolve rule conflicts, where necessary.)

The irrelevant warning message can be suppressed by “installing” a reference lexicon (e.g. a lexical transducer) or a pseudo-lexicon of the language in question, which in this case is Portuguese. When the **twolc** compiler sees that the language of the lower side of the lexicon does not contain any string that contains the sequence “rrr”, it will suppress the warning message. Installing a pseudo-lexicon compiled from the regular expression $\sim\$\{r\ r\ r\}$ would be sufficient to suppress the message in this case; see page 342.

Once the rules have been compiled, they can be tested using the command **lex-test**. This command prompts you to manually enter individual lexical words and shows what the rules produce as output.

```
twolc> lex-test
```

If all the lexical test words have been typed into the file `portwordlist`, with one word per line, then they can all be run through the rules at once using the command **lex-test-file**.

```
twolc> lex-test-file portwordlist
```

twolc will then prompt you for the name of the output file and print the output for each word, also showing how the lexical and surface strings are aligned.

D.5 The Monish Language

D.5.1 One Solution to Monish Analysis

```
# One solution to The Monish Language
# This xfst script is limited to compiling the lexicon
# and saving it to file; the rules are written
# in a separate regex file

clear stack

define FrontV [ i | e | é | ä ] ;          # Front Vowels

define BackV [ u | o | ó | a ] ;          # Back Vowels

define MorphoV [ %^U | %^O | %^Ó | %^A ] ;

    # Morphophonemic Vowels

define Root [ r u u z o d |                # verb roots
t s a r l ó k | n t o n ó l | b u n o o t s |
v é s i i m b | y ä ä q i n | f e s é é n g ] ;

define Suff1 [ %+Int .x. [ %^U %^U k ] ] ;
```

```

# intensifier

# aspect marker
define Suff2 [ %+Perf .x. [ %^O n ] ] | # perfective
[ %+Imperf .x. [ %^O m b ] ] | # imperfective
[ %+Opt .x. [ %^U d d ] ] ; # optative

# confidence of the speaker
define Suff3 [ %+True .x. [ %^A n k ] ] | # true
[ %+Belief .x. [ %^A %^A v %^O t ] ] | # belief
[ %+Doubt .x. [ %^U %^U z ] ] | # credulous
[ %+False .x. [ %^O q ] ] ; # false

# person & number
define Suff4 [ %+1P %+Sg .x. %^A %^A b %^A ] |
[ %+2P %+Sg .x. %^O m %^A ] |
[ %+3P %+Sg .x. %^U v v %^U ] |
[ %+1P %+Pl %+Excl .x. %^A %^A b %^O r %^A ] |
# +Excl == exclusive
[ %+1P %+Pl %+Incl .x. %^A %^A b %^U g %^A ] |
# +Incl == inclusive
[ %+2P %+Pl .x. %^O m %^O r %^A ] |
[ %+3P %+Pl .x. %^U v v %^O r %^U ] ;

read regex [ Root (Suff1) Suff2 (Suff3) Suff4 ] ;
save stack monish-lex.fst
clear stack

```

This above script will save `monish-lex.fst` to file. One way to complete the example is to write the necessary replace rules in a separate regular-expression file, compile them, and then use `xfst` to compose rules on the bottom of the lexicon.

```

# this is monish-rul.regex, a regular-expression file

[ %^U -> u , %^O -> o , %^O -> ó , %^A -> a || BackV ?* _ ]
.ó.
[ %^U -> i , %^O -> e , %^O -> é , %^A -> ä ] ;

```

D.5.2 Monish Guesser-Analyzer

Another way to approach the problem is to define the rules right in the script and perform the composition of the lexicon and the rules in the script as well. The following script, which is also modified to be a guessing analyzer, takes this approach. Note that the definition of `Root` has been modified to accept anything that looks like a valid Monish root.

```

# The Monish Guesser
# Note the definition of Root to cover any string that
# looks like a valid Monish root

```



```

clear stack

define FrontV [ i | e | é | ä ] ;          # Front Vowels

define BackV [ u | o | ó | a ] ;          # Back Vowels

define MorphoV [ %^U | %^O | %^Ó | %^A ] ;

    # Morphophonemic Vowels

# this was the original definition of Root
# define Root [ r u u z o d |                # verb roots
# t s a r l ó k | n t o n ó l | b u n o o t s |
# v é s i i m b | y ä ä q i n | f e s é é n g ] ;

# this is the new version of Root which includes all
# strings that look like valid Monish roots.

define Root [ [ ? - [ BackV | MorphoV ] ]+ & $[ FrontV ] ] |
[ [ ? - [ FrontV | MorphoV ] ]+ & $[ BackV ] ] ;

define Suff1 [ %+Int .x. [ %^U %^U k ] ] ; # intensifier

# aspect marker
define Suff2 [ %+Perf .x. [ %^O n ] ] |     # perfective
[ %+Imperf .x. [ %^Ó m b ] ] |           # imperfective
[ %+Opt .x. [ %^U d d ] ] ;              # optative

# confidence of the speaker
define Suff3 [ %+True .x. [ %^A n k ] ] | # true
[ %+Belief .x. [ %^A %^A v %^O t ] ] |   # belief
[ %+Doubt .x. [ %^U %^U z ] ] |         # credulous
[ %+False .x. [ %^Ó q ] ] ;             # false

# person & number
define Suff4 [ %+1P %+Sg .x. %^A %^A b %^A ] |
[ %+2P %+Sg .x. %^Ó m %^A ] |
[ %+3P %+Sg .x. %^U v v %^U ] |
[ %+1P %+Pl %+Excl .x. %^A %^A b %^O r %^A ] |
    # +Excl == exclusive
[ %+1P %+Pl %+Incl .x. %^A %^A b %^U g %^A ] |
    # +Incl == inclusive
[ %+2P %+Pl .x. %^Ó m %^O r %^A ] |
[ %+3P %+Pl .x. %^U v v %^O r %^U ] ;

define Verbs [ Root (Suff1) Suff2 (Suff3) Suff4 ] ;
define Rules [ %^U -> u , %^O -> o , %^Ó -> ó ,

```

```

%A -> a || BackV ?* _ ]
.o.
[ %U -> i , %O -> e , %Ó -> é , %A -> ä ] ;

# here the lexicon (Verbs) and Rules are composed
# together by the script

read regex Verbs .o. Rules ;
echo << saving monish-guesser.fst to file >>
save stack monish-guesser.fst

echo << monish-guesser.fst is also left on the stack >>

```

D.5.3 Monish Rules in twolc

This is one way to write the Monish alternation rules in **twolc**.

```

Alphabet
    %U:i %O:e %Ó:é %A:ä
    i e é ä
    u o ó a ;

! In the Alphabet section the realizations of
! the underspecified vowels as front vowels are
! declared. The realization of the same
! underspecified vowels as back vowels is
! declared implicitly in the rule below. The
! feasible pairs like %U:u could also be
! declared overtly in the Alphabet section,
! without affecting the result.

Sets
    BackV = u o ó a ;

Rules

"Back Rule"
[ %U:u | %O:o | %Ó:ó | %A:a ] <=> BackV :* _ ;

```

The rule states that the underspecified vowels will be realized as back vowels always and only in words that contain back vowels. Given the defined Alphabet, the grammar will realize all underspecified vowels not in the given context as front vowels. One could also write the grammar with a rule to realize the underspecified vowels as front vowels in the appropriate context, leaving all other cases to be realized as back vowels.

```

! This is another way to write the Monish alternation
! rules in twolc.

```

Alphabet

```
%^U:u %^O:o %^Ó:ó %^A:a
i e é ä
u o ó a ;
```

```
! In the Alphabet section the realizations of
! the underspecified vowels as front vowels are
! declared. The realization of the same
! underspecified vowels as back vowels is
! declared implicitly in the rule below. The
! feasible pairs like %^U:i could also be
! declared overtly in the Alphabet section,
! without affecting the result.
```

Sets

```
FrontV = i e é ä ;
```

Rules

```
"Front Rule"
[ %^U:i | %^O:e | %^Ó:é | %^A:ä ] <=> FrontV :* _ ;
```

D.6 Tag Moving

D.6.1 Tag Moving in a Regular Expression

Here is one solution to the Tag Moving exercise on page 351. Assume that the lexicon, with strings like “Neg+healthy+Adj” on the upper side, has been compiled and stored in `adj-lexc.fst`. The rule composed directly above the lexicon inserts a new suffix-like tag, +Neg, at the end of strings that contain a prefix tag Neg+. The topmost rule then deletes the original Neg+ tag.

```
xfst[0]: read regex
0 <- "Neg+"
.o.
"+Neg" <- [...] || $["Neg+"] _ .#.
.o.
@"adj-lexc.fst" ;
```

xfst offers these upward-oriented left-arrow replace rules, which are intuitively ideal for rules that are to be composed on the top of a lexicon transducer.

D.6.2 Tag Moving with **twolc** Rules and **lexc** Composition

Here is one solution to Tag Moving exercise on page 351. **twolc** rules offer only a downward orientation, so we will first need to write the rule(s) upside-down, to map a string like “Neg+healthy+Adj” downward to “healthy+Adj+Neg”. Assume that the following grammar is written in file `neg-twolc.txt`.

```
! file neg-twolc.txt
```

```
Alphabet Neg%+:0 ;
```

```
Rules
```

```
"Insert Neg Suffix Tag"
```

```
0:%+Neg <=>  $[Neg%+:] \0:%+Neg _ .#. ;
```

```
! N.B. it is important to specify a left context
! that contains Neg%+: rather than just Neg%+,
! which would be interpreted as Neg%+:Neg%+
```

```
! the \0:%+Neg in the immediate left context
! prevents twolc from trying to insert an infinite
number of new tags
```

In **twolc**, you should be able to compile and test the grammar with the following instructions:

```
twolc> read-grammar neg-twolc.txt
twolc> compile
twolc> lex-test Neg+worthy+Adj
           worthy+Adj+Neg
```

The output of **lex-test** should show “worthy+Adj+Neg” with the negative tag effectively moved to the end of the word. Then quit from **lex-test** mode, save the network to the binary file `neg-twolc.fst`, and quit **twolc**.

```
Lexical string ("q" = quit): q
twolc> save-binary neg-twolc.fst
twolc> quit
```

Then to compose the rules on the lexicon using **lexc**, perform the following **lexc** instructions.

```
lexc> read-source adj-lexc.fst
lexc> lookup unhealthy
Neg+healthy+Adj
lexc> invert-source
lexc> read-rules neg-twolc.fst
lexc> compose-result
lexc> invert-result
lexc> lookup unhealthy
Use (s)ource or (r)esult? [r]: r
healthy+Adj+Neg
```

Note that because the **twolc** rules were necessarily written upside down, the lexicon must first be inverted, so that the rules can apply to the correct side. That leaves the result inverted, so we then invert the result to get the final solution.

D.6.3 Tag Moving with twolc Rules and xfst Composition

This is one solution to the exercise on page 351. Assuming that we have the **lexc** lexicon and the **twolc** rules already compiled and stored as `adj-lexc.fst` and `neg-twolc.fst` respectively, we can compose them in **xfst**. As the **twolc** rules had to be written upside-down, we simply need to invert the rule transducer and compose it on the upper-side of the lexicon, as in the following **xfst** session:

```
xfst[0]: read regex
["@neg-twolc.fst"].i
.o.
@"adj-lexc.fst" ;
xfst[1]:
```

Another equivalent solution, performing the inversion and composition on the **xfst** stack, is the following:

```
xfst[0]: load adj-lexc.fst
xfst[1]: load neg-twolc.fst
xfst[2]: invert net
xfst[2]: compose net
```

D.7 Esperanto Nouns

One solution to the Esperanto Noun exercise on page 230.

```
LEXICON Root      ! corresponds to the start state
      Nouns ;

LEXICON Nouns     ! noun roots
bird    Nmf ;
hund    Nmf ;
kat     Nmf ;
elefant Nmf ;

LEXICON Nmf
in      Nmf ;      ! feminine; note the loop back to Nmf
et      Nmf ;      ! diminutive; note the loop
eg      Nmf ;      ! augmentative; note the loop
      Nend ;      ! escape path to Nend

LEXICON Nend
o       Number ;  ! required noun suffix
```

```

LEXICON Number
j      Case ;      ! optional number suffix
      Case ;

```

```

LEXICON Case
n      # ;         ! optional accusative case suffix
      # ;

```

D.8 Esperanto Adjectives

This is one solution to the Esperanto Adjectives exercise on page 234.

```

! esp-adjs-lex.txt

LEXICON Root
      Adjectives ;      ! start with an adj root
      AdjPrefix ;      ! or a prefix

LEXICON AdjPrefix
mal    Adjectives ;
ne     Adjectives ;

LEXICON Adjectives      ! adjective roots
bon    Adj ;
long   Adj ;
jun    Adj ;
alt    Adj ;
grav   Adj ;

LEXICON Adj             ! looped
eg     Adj ;           ! augmentative
et     Adj ;           ! diminutive
      Adjend ;        ! escape

LEXICON Adjend
a      Number ;       ! required adjective suffix

LEXICON Number
j      Case ;         ! optional plural suffix
      Case ;

LEXICON Case
n      # ;           ! optional accusative-case
      # ;           ! suffix

```

D.9 Esperanto Nouns and Adjectives

This is one solution to the Esperanto Nouns and Adjectives exercise on page 236.

```

LEXICON Root
      Nouns ;
      Adjectives ;

LEXICON Nouns
      NounRoots ;
ge      NounRoots ; ! optional ge prefix
          ! ("male and female")

LEXICON NounRoots ! noun roots
bird     Nmf ;
hund     Nmf ;
kat      Nmf ;
elefant  Nmf ;

LEXICON Nmf
in       Nmf ;           ! feminine; note the loop back to Nmf
et       Nmf ;           ! diminutive; note the loop
eg       Nmf ;           ! augmentative; note the loop
          Nend ;         ! escape path to Nend
          Adjend ;       ! escape path to Adjend for
          ! derivation of noun to adjective

LEXICON Nend
o        Number ;       ! required noun suffix

LEXICON Adjectives
          AdjPrefixes ; ! start with a prefix
          AdjRoots ;    ! or an adjective root

LEXICON AdjPrefixes
mal      AdjRoots ;
ne       AdjRoots ;

LEXICON AdjRoots ! adjective roots
bon      Adj ;
long     Adj ;
jun      Adj ;
alt      Adj ;
grav     Adj ;

LEXICON Adj ! looped
eg       Adj ;           ! augmentative
et       Adj ;           ! diminutive
          Adjend ;       ! escape to Adjend
ec       Nend ;         ! escape to Nend for derivation

```

```

! of adjective to noun

LEXICON Adjend
a      Number ;      ! required adjective suffix

LEXICON Number
j      Case ;        ! optional plural suffix
      Case ;

LEXICON Case
n      # ;           ! optional accusative-case
      # ;           ! suffix

```

D.10 Esperanto Nouns and Adjectives with Upper-Side Tags

This is one solution to the Esperanto Nouns and Adjectives with Upper-Side Tags exercise on page 244.

```

Multichar_Symbols +Noun +Adj +NSuff +ASuff +Nize
                  +Pl +Sg +Acc MF+
                  +Aug +Dim +Fem Op+ Neg+

```

```

LEXICON Root
      Nouns ;
      Adjectives ;

```

```

LEXICON Nouns
      NounRoots ;
MF+:ge NounRoots ;      ! optional ge prefix
                        ! ("male and female")

```

```

LEXICON NounRoots      ! noun roots
bird      Nmf ;
hund      Nmf ;
kat       Nmf ;
elefant   Nmf ;

```

```

LEXICON Nmf
+Noun:0 AugDimFem ;

```

```

LEXICON AugDimFem
+Fem:in AugDimFem ;      ! feminine; note the loop back to
                          ! AugDimFem
+Dim:et AugDimFem ;      ! diminutive; note the loop
+Aug:eg AugDimFem ;      ! augmentative; note the loop
      Nend ;              ! escape path to Nend

```



```

    Adjend ;           ! escape path to Adjend for
                       ! derivation of noun to adjective

LEXICON Nend
+N Suff:o  Number ;   ! required noun suffix

LEXICON Adjectives
  AdjPrefixes ;      ! start with a prefix
  AdjRoots ;         ! or an adjective root

LEXICON AdjPrefixes
Op+:mal AdjRoots ;
Neg+:ne AdjRoots ;

LEXICON AdjRoots      ! adjective roots
bon  Adj ;
long Adj ;
jun  Adj ;
alt  Adj ;
grav Adj ;

LEXICON Adj
+Adj:0 AugDimNize ;

LEXICON AugDimNize    ! looped
+Aug:eg AugDimNize ; ! augmentative
+Dim:et AugDimNize ; ! diminutive
      Adjend ;       ! escape to Adjend
+Nize:ec  Nend ;     ! escape to Nend for derivation
                       ! of adjective to noun

LEXICON Adjend
+ASuff:a  Number ;   ! required adjective suffix

LEXICON Number
+Pl:j    Case ;      ! optional plural suffix
+Sg:0    Case ;

LEXICON Case
+Acc:n   # ;         ! optional accusative-case
                       ! suffix
      # ;

```

D.11 Esperanto Nouns, Adjectives and Verbs

This is one solution to the Esperanto Nouns, Adjectives and Verbs exercise on page 245.

```
Multichar_Symbols +Noun +Adj +NSuff +ASuff +Nize
```

+Pl +Sg +Acc MF+
 +Aug +Dim +Fem Op+ Neg+
 +Verb +Cont +Inf +Pres +Past +Fut +Cond +Subj

LEXICON Root
 Nouns ;
 Adjectives ;
 Verbs ;

LEXICON Nouns
 NounRoots ;
 MF+:ge NounRoots ; ! optional ge prefix
 ! ("male and female")

LEXICON NounRoots ! noun roots
 bird Nmf ;
 hund Nmf ;
 kat Nmf ;
 elefant Nmf ;

LEXICON Nmf
 +Noun:0 AugDimFem ;

LEXICON AugDimFem
 +Fem:in AugDimFem ; ! feminine; note the loop back to
 ! AugDimFem
 +Dim:et AugDimFem ; ! diminutive; note the loop
 +Aug:eg AugDimFem ; ! augmentative; note the loop
 Nend ; ! escape path to Nend
 Adjend ; ! escape path to Adjend
 ! (noun to adjective)

LEXICON Nend
 +NSuff:o Number ; ! required noun suffix

LEXICON Adjectives
 AdjPrefixes ;
 AdjRoots ; ! start with a prefix or
 ! an adjective root

LEXICON AdjPrefixes
 Op+:mal AdjRoots ;
 Neg+:ne AdjRoots ;

LEXICON AdjRoots ! adjective roots
 bon Adj ;
 long Adj ;
 jun Adj ;
 alt Adj ;

```

grav    Adj ;

LEXICON Adj
+Adj:0  AugDimNize ;

LEXICON AugDimNize      ! looped
+Aug:eg AugDimNize ;    ! augmentative
+Dim:et AugDimNize ;    ! diminutive
      Adjend ;          ! escape to Adjend
+Nize:ec  Nend ;        ! escape to Nend, derivation
      ;                  ! of adjective to noun

LEXICON Adjend
+ASuff:a  Number ;      ! required adjective suffix

LEXICON Number
+Pl:j    Case ;          ! optional plural suffix
+Sg:0    Case ;

LEXICON Case
+Acc:n   # ;            ! optional accusative-case
      ;                  !suffix
      # ;

LEXICON Verbs
      VerbPrefixes ;
      VerbRoots ;

LEXICON VerbPrefixes
Op+:mal  VerbRoots ;
Neg+:ne  VerbRoots ;

LEXICON VerbRoots
don      V ;            ! give
est      V ;            ! be
pens     V ;            ! think
dir      V ;            ! say
fal      V ;            ! fall

LEXICON V
+Verb:0  Aspect ;

LEXICON Aspect
+Cont:ad  Vend ;
      Vend ;

LEXICON Vend
+Inf:i    # ;
+Pres:as  # ;

```

```
+Past:is      # ;
+Fut:os      # ;
+Cond:us     # ;
+Subj:u      # ;
```

D.12 Einstein II Problem

Here is one solution to the Einstein II problem on page 577. Each car is modeled as a concatenation of name, color, beverage, hobby and dog, in that order. The five cars, parked in a row, are modeled as a concatenation of car models, with a **Line** symbol (like a painted parking line) between cars and at the beginning and end.

```
! A Solution to the Einstein II problem.

! Model each car as a concatenation of
! Name, Color, Beverage, Hobby and Dog, in that order.

! Model the line of cars as a concatenation of five
! car strings with a Line symbol between cars and
! at the beginning and end.

clear stack
define Name    Mark | Lucie | Dave | Brenda | Susan ;
define Color  teal | purple | tan | blue | green ;
define Beverage sparklingwater | espresso | tea |
               raspberrymocha | grapesoda ;
define Hobby   stampcollecting | carracing | golf |
               swimming | painting ;
define Dog     yellowlab | poodles | terrier |
               collie | bichonfrise ;

define Car Name Color Beverage Hobby Dog ;
define FiveCars Line [Car Line]^5 ;

define SameCar [\Line]* ;
define NextTo SameCar Line SameCar ;

! as defined here, X NextTo Y means that
! X is on the left and Y on the right;
! Y NextTo X is the reverse

read regex [ [ FiveCars
.o.
Line ${Mark} [Line Car]^4 Line
```

```

.o.
$[Lucie SameCar teal]
.o.
$[Dave SameCar sparklingwater]
.o.
$[purple NextTo tan]
.o.
$[ [carracing NextTo espresso] | [ espresso NextTo car-
racing ] ]
.o.
$[Brenda SameCar yellowlab]
.o.
$[tea SameCar golf]
.o.
$[purple SameCar raspberrymocha]
.o.
$[stampcollecting SameCar poodles]
.o.
$[ [carracing NextTo terrier] | [terrier NextTo carrac-
ing] ]
.o.
$[ [collie NextTo swimming] | [swimming NextTo collie] ]
.o.
$[ Susan SameCar painting ]
.o.
Line [Car Line]^2 $[grapesoda] [Line Car]^2 Line
.o.
$[ [Mark NextTo blue] | [blue NextTo Mark] ]
.o.
$[ green SameCar swimming ]
.o.
$[bichonfrise]
] .o. Line -> "\n" .o. ? -> ... % ].1 ;
print words

```

The output of this script is

```

Mark green espresso swimming terrier
Dave blue sparklingwater carracing collie
Lucie teal grapesoda stampcollecting poodles
Susan purple raspberrymocha painting bichonfrise
Brenda tan tea golf yellowlab

```

showing that it is Susan who owns the Bichon Frise.

Bibliography

- Antworth, E. L. (1990). *PC-KIMMO: a two-level processor for morphological analysis*. Number 16 in Occasional publications in academic computing. Summer Institute of Linguistics, Dallas.
- Armstrong, S. (1996). Multext: Multilingual text tools and corpora. In Feldweg, H. and Hinrichs, E. W., editors, *Lexikon und Text*, pages 107–112. Max Niemeyer Verlag, Tuebingen.
- Barton, Jr., G. E. (1986a). Computational complexity in two-level morphology. In *Proceedings of the Conference, 24th Annual Meeting of the Association for Computational Linguistics*, pages 53–59. Columbia University.
- Barton, Jr., G. E. (1986b). Constraint propagation in KIMMO systems. In *Proceedings of the Conference, 24th Annual Meeting of the Association for Computational Linguistics*, pages 45–52. Columbia University.
- Barton, Jr., G. E. (1987). The complexity of two-level morphology. In Barton, G. E., Berwick, R., and Ristad, E., editors, *Computational Complexity and Natural Language*. MIT Press, Cambridge, MA.
- Beesley, K. R. (1989). Computer analysis of Arabic morphology: A two-level approach with detours. In *Third Annual Symposium on Arabic Linguistics*, Salt Lake City. University of Utah. Published as Beesley, 1991.
- Beesley, K. R. (1990). Finite-state description of Arabic morphology. In *Proceedings of the Second Cambridge Conference on Bilingual Computing in Arabic and English*. No pagination.
- Beesley, K. R. (1991). Computer analysis of Arabic morphology: A two-level approach with detours. In Comrie, B. and Eid, M., editors, *Perspectives on Arabic Linguistics III: Papers from the Third Annual Symposium on Arabic Linguistics*, pages 155–172. John Benjamins, Amsterdam. Read originally at the Third Annual Symposium on Arabic Linguistics, University of Utah, Salt Lake City, Utah, 3-4 March 1989.
- Beesley, K. R. (1996). Arabic finite-state morphological analysis and generation. In *COLING'96*, volume 1, pages 89–94, Copenhagen. Center for Sprogteknologi. The 16th International Conference on Computational Linguistics.

- Beesley, K. R. (1998a). Arabic morphological analysis on the Internet. In *ICEMCO-98*, Cambridge. Centre for Middle Eastern Studies. Proceedings of the 6th International Conference and Exhibition on Multi-lingual Computing. Paper number 3.1.1; no pagination.
- Beesley, K. R. (1998b). Arabic morphology using only finite-state operations. In Rosner, M., editor, *Computational Approaches to Semitic Languages: Proceedings of the Workshop*, pages 50–57, Montréal, Québec. Université de Montréal.
- Beesley, K. R. (1998c). Arabic stem morphotactics via finite-state intersection. Paper presented at the 12th Symposium on Arabic Linguistics, Arabic Linguistic Society, 6-7 March, 1998, Champaign, IL.
- Beesley, K. R., Buckwalter, T., and Newton, S. N. (1989). Two-level finite-state analysis of Arabic morphology. In *Proceedings of the Seminar on Bilingual Computing in Arabic and English*, Cambridge, England. No pagination.
- Beesley, K. R. and Karttunen, L. (2000). *Finite-State Morphology: Xerox Tools and Techniques*. Submitted to Cambridge University Press.
- Blåberg, O. (1994). *The Ment Model—Complex States in Finite State Morphology*. Number 27 in Ruul. Uppsala University, Uppsala.
- Black, A., Ritchie, G., Pulman, S., and Russell, G. (1987). Formalisms for morphographemic description. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics*, pages 11–18.
- Carter, D. (1995). Rapid development of morphological descriptions for full language processing systems. In *Proceedings of the Seventh Conference of the European Chapter of the Association for Computational Linguistics*, pages 202–209.
- Chomsky, N. and Halle, M. (1968). *The Sound Pattern of English*. Harper and Row, New York.
- Dalrymple, M., Doron, E., Goggin, J., Goodman, B., and McCarthy, J., editors (1983). *Texas Linguistic Forum, Vol. 22*. Department of Linguistics, The University of Texas at Austin, Austin, TX.
- Echols, J. M. and Shadily, H., editors (1989). *An Indonesian-English Dictionary*. Cornell University Press, Ithaca, New York.
- Eisner, J. (1997). Efficient generation in primitive Optimality Theory. In *Proceedings of the 35th Annual ACL and 8th EACL*, pages 313–320, Madrid. Association for Computational Linguistics.

- Frank, R. and Satta, G. (1998). Optimality theory and the generative complexity of constraint violability. *Computational Linguistics*, 24(2):307–316.
- Gajek, O., Beck, H. T., Elder, D., and Whittemore, G. (1983). LISP implementation. In Dalrymple, M., Doron, E., Goggin, J., Goodman, B., and McCarthy, J., editors, *Texas Linguistic Forum, Vol. 22*, pages 187–202. Department of Linguistics, The University of Texas at Austin, Austin, TX.
- Grimley-Evans, E., Kiraz, G. A., and Pulman, S. G. (1996). Compiling a partition-based two-level formalism. In *COLING'96*, Copenhagen. cmp-lg/9605001.
- Halle, M. and Clements, G. N. (1983). *Problem Book in Phonology: A Workbook for Introductory Courses in Linguistics and in Modern Phonology*. MIT Press, Cambridge, MA.
- Harris, Z. (1941). Linguistic structure of Hebrew. *Journal of the American Oriental Society*, 62:143–167.
- Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Reading, Massachusetts.
- Hudson, G. (1986). Arabic root and pattern morphology without tiers. *Journal of Linguistics*, 22:85–122. Reply to McCarthy:1981.
- Johnson, C. D. (1972). *Formal Aspects of Phonological Description*. Mouton, The Hague.
- Kager, R. (1999). *Optimality Theory*. Cambridge University Press, Cambridge, England.
- Kaplan, R. M. and Kay, M. (1981). Phonological rules and finite-state transducers. In *Linguistic Society of America Meeting Handbook, Fifty-Sixth Annual Meeting*, New York. Abstract.
- Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.
- Karttunen, L. (1983). KIMMO: a general morphological processor. In Dalrymple, M., Doron, E., Goggin, J., Goodman, B., and McCarthy, J., editors, *Texas Linguistic Forum, Vol. 22*, pages 165–186. Department of Linguistics, The University of Texas at Austin, Austin, TX.
- Karttunen, L. (1993). Finite-state lexicon compiler. Technical Report ISTL-NLTT-1993-04-02, Xerox Palo Alto Research Center, Palo Alto, CA.
- Karttunen, L. (1995). The replace operator. In *ACL'95*, Cambridge, MA. cmp-lg/9504032.

- Karttunen, L. (1996). Directed replacement. In *ACL'96*, Santa Cruz, CA. cmp-
lg/9606029.
- Karttunen, L. (1998). The proper treatment of optimality in computational phonology. In *FSMNL'98. International Workshop on Finite-State Methods in Natural Language Processing*, Bilkent University, Ankara, Turkey. cmp-
lg/9804002.
- Karttunen, L. and Beesley, K. R. (1992). Two-level rule compiler. Technical Report
ISTL-92-2, Xerox Palo Alto Research Center, Palo Alto, CA.
- Karttunen, L., Chanod, J.-P., Grefenstette, G., and Schiller, A. (1996). Regular
expressions for language engineering. *Journal of Natural Language Engi-
neering*, 2(4):305–328.
- Karttunen, L., Kaplan, R. M., and Zaenen, A. (1992). Two-level morphology with
composition. In *COLING'92*, pages 141–148, Nantes, France.
- Karttunen, L. and Kempe, A. (1995). The parallel replacement operation in finite-
state calculus. Technical Report MLTT-021, Xerox Research Centre Europe,
Grenoble, France. <http://www.xrce.xerox.com/publis/mltt/mltttech.html>.
- Karttunen, L., Koskenniemi, K., and Kaplan, R. M. (1987). A compiler for
two-level phonological rules. In Dalrymple, M., Kaplan, R., Karttunen, L.,
Koskenniemi, K., Shaio, S., and Wescoat, M., editors, *Tools for Morpholog-
ical Analysis*, volume 87-108 of *CSLI Reports*, pages 1–61. Center for the
Study of Language and Information, Stanford University, Palo Alto, CA.
- Karttunen, L., Uszkoreit, H., and Root, R. (1981). Morphological analysis of
Finnish by computer. In *Proceedings of the 71st Annual Meeting of SASS*.
Albuquerque, New Mexico.
- Kataja, L. and Koskenniemi, K. (1988). Finite-state description of Semitic mor-
phology: A case study of Ancient Akkadian. In *COLING'88*, pages 313–315.
- Kay, M. (1987). Nonconcatenative finite-state morphology. In *Proceedings of the
Third Conference of the European Chapter of the Association for Computa-
tional Linguistics*, pages 2–10.
- Kempe, A. and Karttunen, L. (1996). Parallel replacement in finite-state calculus.
In *COLING'96*, Copenhagen. cmp-[lg/9607007](http://www.cmp-ig.org/9607007).
- Kiraz, G. A. (1994a). Multi-tape two-level morphology: a case study in Semitic
non-linear morphology. In *COLING'94*, volume 1, pages 180–186.
- Kiraz, G. A. (1994b). Multi-tape two-level morphology: a case study in Semitic
non-linear morphology. In *Proceedings of the 15th International Conference
on Computational Linguistics*, Kyoto, Japan.

- Kiraz, G. A. (1996). Semhe: A generalised two-level system. In *Proceedings of the 34th Annual Meeting of the Association of Computational Linguistics*, Santa Cruz, CA.
- Kiraz, G. A. (2000). Multitiered nonlinear morphology using multitape finite automata: A case study on Syriac and Arabic. *Computational Linguistics*, 26(1):77–105.
- Kiraz, G. A. and Grimley-Evans, E. (1999). Multi-tape automata for speech and language systems: A prolog implementation. In Champarnaud, J.-M., Maurel, D., and Ziadi, D., editors, *Automata Implementation*, volume 1660 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, Germany.
- Kisseberth, C. (1969). On the abstractness of phonology. *Papers in Linguistics*, 1:248–282.
- Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. E. and McCarthy, J., editors, *Automata Studies*. Princeton University Press.
- Koskenniemi, K. (1983). Two-level morphology: A general computational model for word-form recognition and production. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki.
- Koskenniemi, K. (1984). A general computational model for word-form recognition and production. In *COLING'84*, pages 178–181.
- Koskenniemi, K. (1986). Compilation of automata from morphological two-level rules. In Karlsson, F., editor, *Papers from the Fifth Scandinavian Conference on Computational Linguistics*, pages 143–149.
- Koskenniemi, K. and Church, K. (1988). Complexity, two-level morphology and Finnish. In *COLING'88*, pages 335–339.
- Lavie, A., Itai, A., Ornan, U., and Rimon, M. (1988). On the applicability of two level morphology to the inflection of Hebrew verbs. In *ALLC III*, pages 246–260.
- Mallon, M. (1999). Inuktitut linguistics for technocrats. Technical report, Ittukuluuk Language Programs, Iqaluit, Nunavut, Canada.
- McCarthy, J. J. (1981). A prosodic theory of nonconcatenative morphology. *Linguistic Inquiry*, 12(3):373–418.
- McCarthy, J. J. (2002). *The Foundations of Optimality Theory*. Cambridge University Press, Cambridge, England.
- Mohri, M. (1997). Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

- Partee, B. H., ter Meulen, A., and Wall, R. E. (1993). *Mathematical Methods in Linguistics*. Kluwer, Dordrecht.
- Petitpierre, D. and Russel, G. (1995). MMORPH — the multext morphology program. Technical report, Carouge, Switzerland. Multext Deliverable 2.3.1.
- Prince, A. and Smolensky, P. (1993). Optimality Theory: Constraint interaction in generative grammar. Technical report, Rutgers University, Piscataway, NJ. RuCCS Technical Report 2. Rutgers Center for Cognitive Science.
- Pulman, S. (1991). Two level morphology. In Alshawi, H., Arnold, D., Backofen, R., Carter, D., Lindop, J., Netter, K., Pulman, S., Tsujii, J., and Uskoreit, H., editors, *ET6/1 Rule Formalism and Virtual Machine Design Study*, chapter 5. CEC, Luxembourg.
- Ritchie, G., Black, A., Pulman, S., and Russell, G. (1987). The Edinburgh/Cambridge morphological analyser and dictionary system (version 3.0) user manual. Technical Report Software Paper No. 10, Department of Artificial Intelligence, University of Edinburgh.
- Ritchie, G., Russell, G., Black, A., and Pulman, S. (1992). *Computational Morphology: Practical Mechanisms for the English Lexicon*. MIT Press, Cambridge, Mass.
- Roche, E. and Schabes, Y. (1997). Introduction. In *Finite-State Language Processing*, chapter 1, pages 1–65. MIT Press, Cambridge, Massachusetts.
- Ruessink, H. (1989). Two level formalisms. In *Utrecht Working Papers in NLP*, volume 5.
- Schützenberger, M.-P. (1961). A remark on finite transducers. *Information and Control*, 4:185–196.
- Sproat, R. (1992). *Morphology and Computation*. MIT Press, Cambridge, MA.
- Street, J. C. (1963). *Khalkha Structure*. Mouton, The Hague, Netherlands.

Index

Symbols

!..... *see* lexc, comment
 452
 ?..... *see* lexc, command, ?
 @..... 443, 444, 452
 [..... 587
 [..]..... 179
 # *see* xfst, comment, *see* lexc, end of
 word
 @ *see* Flag Diacritic, spelling
 Lexical Grammar 410, 429
 Lexical Transducer 410
 <..... *see* lexc, angle bracket
 >..... *see* lexc, angle bracket
 |51, *see* regular expression, operator,
 |
 \ 50
 () 49
 * *see* regular expression, operator, *,
 49
 + 49
 - 52, *see* regular expression,
 operator, -
 69
 . / 51
 . # 49
 . i 53
 . l 52
 . o 53
 . r 53
 . u 52
 . x 52
 / 51
 0 16, 48, 239
 ; *see* semicolon, *see* regular
 expression, semicolon

? : ? 49
 ? . *see* regular expression, symbol, ?
 [. .] 49, 68
 [] 49
 \$ 51
 % *see* lexc, character, literal-
 ized, *see* regular expression,
 special character, %
 & 51, *see* regular expression,
 operator, &
 ~ 50
 ~ *see* regular expression, operator, ~
 {} 50, *see* regular expression,
 operator, {}
 ANY 48
 (->) 67
 (<-) 67
 -> 67
 <- 67
 ^ < 50
 ^ > 50
 ^ { , } 51
 ^ 50
 — *see* replace rule, context
 separator

Numbers

0 152

A

abbreviation 529, 532, 533, 599
 abstraction 519
 accentuation 547
 relaxation 551
 accusative . 231, *see* case, accusative
 add properties 199

- add-props *see* lexc, command,
add-props
- adjective vi, 248, 259, 595
comparative 249
superlative 249
- adverb 597
- agglutinating 482
- agglutinative 230, 479
- algorithm 40
- alias 204
- allcap *see* capitalization, all
- allomorph 468
- alphabet . . **10**, 11, 12, 278, 328, 331,
410, 414, 418
check 423
checking 418, 423
corrupted 421
debug 421
problem 421
- alphabet, sigma 102
- ALPNET 504
- Alpnet vi
- ALPS vi
- alternation . . . viii–x, **x**, **30**, 213, 279,
289, 308, 329
- ambiguities *see* lexc, command,
ambiguities, 294
- ambiguous 533
- ampersand . . . *see* regular expression,
operator, &
- analysis 12–14, 17,
18, 144, *see* xfst, command,
apply up, 145, 216, *see* lexc,
command, lookup, *see* xfst,
command, apply up, **282**, 284,
429, 441
- angle bracket 137, 139
- angle brackets 134
- Antworth, Evan . . 212, 309, 479, 487
- any symbol 58
- application 34
- application routines 441
- applications 528
- apply
multicharacter symbols 64, 191,
421
apply down **123**, *see* xfst, apply down
tokenization 64, **191**, **421**
apply up **123**, *see* xfst, apply up, 501
tokenization 64, **191**, **421**
- Arabic . . vi, ix, xi, 13, 265, 293, 377,
444, 454, 480, 550, 551
Modern Standard 501
- arc 5, 8, 11, 13, 14, 42, 47
- arc label 47
- archiphoneme 168, 258
- archive 378
- arity 193
- article 597
- ASCII 16
- aspect 168, 261, 593
- associative 50
- asterisk *see* regular expression,
operator, *, 240
- at-sign 443, 444, 452
- audience vii
- augmentative 225, 230, 234
- autosegmental phonology 502
- Aymara vi, 280
- B**
- backslash-slash rules 178
- backtrack **16**, 455, 456
- backtracking 454
- backup 378, 433
- backward . . . 17, 129, 145, 282, 286
- Bambona 155–166, 247
- banner . . *see* lexc, command, banner
- bar *see* regular expression, operator, |
- Barton
G. Edward 563, 565
- baseformviii, x, 15, 16, 38, 382, 410,
427, 428, 543
choosing 382
- Basque xi, 482
- Beesley
Kenneth R. 569
- Beesley, Kenneth 504

- Beesley, Kenneth R. . . . v, **vi**, 317, 318
 begin-script *see* lexc, command,
 begin-script
 bidirectional x, 17, 282
 bimachine 77
 Bläberg, Olli 468
 Boolean logic 46
 bootstrapping 486
 brace
 curly *see* regular expression,
 operator, { }
 bracket
 dotted *see* dotted brackets
 bracketing *see* regular expression,
 operator, []
 bracketing rules 180
 brackets
 square *see* square brackets
 Brazil 150
 brute force 284
- C**
- C. v, 212, 227, 309, 317
 capitalization 547
 all 547, **549**
 initial **547**, 548, 550
 cardinal 595
 cascade . . . 37, *see* rule, cascade, 283
 pipe 286
 case 159, 265, 443, 446, 591
 accusative 235
 cat *see* Unix, utility, cat, 532
 category 261
 character
 special 419, 444, 452
 chart 379
 check the alphabet 418
 check-all *see* lexc, command, check-
 all, *see* lexc, command, check-
 all
 Chinese 529
 Chomsky, Noam 309
 chunker iii
 chunking . . . *see* parsing, shallow, 561
- Church
 Kenneth 564
 circle 5, 8
 double 9
 circular 273
 circumfixation . . . 456, 457, **469**, 491
 circumflex 240, 425
 cleanup net 199
 clear . *see* xfst, command, clear stack
 clear stack . *see* xfst, command, clear
 stack
 closure 56
 co-occurrence *see* morpheme,
 co-occurrence, 444
 Cognitive Science vi
 cola machine *see* exercise, cola
 machine, 9, 10
 comma 240
 comment
 twolc *see* twolc, comment
 Common Lisp v, 212, 317
 commutative 51
 compact sigma 199
 comparative 595
 compile-replace 478–526
 debugging 523
 flag diacritics 465, 524
 no-retokenize mode 524
 retokenize 524
 size 525
 usage notes 518
 compile-source . *see* lexc, command,
 compile-source, *see* lexc, com-
 mand, compile-source
 complement 94, 240
 language 98
 symbol 98
 complementation 48, 50
 complete net 199
 completion *see* lexc, command,
 completion
 compose 377
 compose net . . . *see* xfst, compose net

- compose-result . *see* lexc, command,
compose-result, *see* lexc, compose-
result
- compose net 143
- composition vii, 30,
see regular expression, op-
erator, .o., *see* xfst, com-
mand, compose net, 39, 53,
142, 155, 254, 264, **282**, xfst286,
284–291, 384, 385, 440, 461,
463, 466, 602
- flag diacritics 463–465
- intersecting 254, 289
- lexc 289
- simulated 541
- size 397
- virtual 540
- compound 468
- compounding . . . x, 22, 248–250, 410
- compression x, 39
- concatenation 29, *see* xfst,
concatenate net, 30, 39, 50,
see regular expression, op-
erator, concatenation, **128**, 157,
223, 264, 267, 418, 479, 482
- concatenative 482
- confidence 169
- constraint 440–442, 451, 454
- constraint satisfaction 515
- consume 13
- containment 51
- context-sensitive power 525
- continuation class *see* lexc,
continuation class, 265
- contraction 18
- copyright ii
- core system 385
- corpora *see* corpus
- corpus 410, 432
- counting 19
- coverage 416, 433, 437
- creative laziness 284, 407
- crossproduct 52,
see regular expression, op-
erator, .x., *see* xfst, com-
mand, crossproduct, *see* xfst,
command, crossproduct net
- CSLI v
- curly brace . . *see* regular expression,
operator, {}
- curly braces . . . 64, 95, 97, 130, 168
- CVS 378
- ## D
- Danish xi
- date 529, 532
- deaccentuation 540, 541, 558
- debugging 410, 429
compile-replace 523
- declarative 212
- delimiter 598
- dependency 602
long-distance 564
- deprecated 442
- derivation x, 597
- derived networks 199
- destination state 47
- determiner 597
- determinize net 199
- detouring 504
- development tools 528
- diacritic *see* Flag
Diacritic; twolc, Diacritics,
442, *see* twolc, Diacritics
- dialect 376, **385**, 599
- dictionary 12
- dime 8–10, 40, 41
- diminutive 230, 234
- dimmer 7
- directory structure 609
- disambiguation 528
- disambiguator ix
- discontiguous dependencies 264, *see*
separated dependencies
- discourse 561
- distribution list iv
- documentation xii
- dotted brackets 68, 179, 180

double vertical-bar rules 173
 double-arrow rules 186
 double-backslash rules 177
 double-slash rules 173
 down 282, 284
 duplicates *see* lexc, command,
 duplicates, 293
 Dutch vi, ix, x, 317, 377

E

Eastern Europe xi
 echo . . 532, *see* xfst, command, echo
 edit properties 199
 Einstein
 Albert 571
 Einstein II 577
 Einstein Problem 571
 solution 572, 575
 eliminate flag . . . *see* xfst, command,
 eliminate flag, 463, *see* xfst,
 eliminate flag
 elision 466, 558
 emacs viii, ix, xi, 321
 emphasis 599
 empty language *see* language, empty
 empty set *see* set, empty
 empty string *see* string, empty
 encoding 540, 550, 559
 end-script *see* lexc, command,
 end-script
 English vii, ix, x, 13, 283, 317
 enumeration 21, 22, 431
 epenthesis 179, 558
 epenthesis rules 179
 Epistemics vi
 epsilon **16**, 40, 48, 92,
 109, *see* string, empty, 152,
 see string, empty, 329, 441,
 442, 448, 454, 458, 472
 epsilon-remove net 199
 equivalence . . *see* xfst, command, test
 equivalent
 errata iv

Esperanto . . 225, 228, 244, 245, 265,
 273, 279, 467
 exclusive 169
 exclusive we 593
 exercise viii
 Bambona 155, 247
 better cola machine 40
 Brazilian Portuguese Pronuncia-
 tion 334
 cola machine 8
 Einstein II 577
 Esperanto adjectives 234
 Esperanto nouns 228
 Esperanto nouns and adjectives
 with upper-side tags . . . 244
 Esperanto verbalization . . . 247
 Esperanto verbs 245
 Irish Gaelic lenition 296
 kaNpat 141–145, 333
 Monish 166
 Pseudo-Arabic 273
 Simplified Germanic compound-
 ing 248
 softdrink machine 41
 Southern Brazilian Portuguese 149
 exercises iv
 explode 94
 explosion . . . 95, 130, 418, 419, 432,
 461, 525
 extract-surface-forms . *see* lexc, com-
 mand, extract-surface-forms

F

failure 410, 411
 failures . *see* lexc, command, failures,
 293, 411, 413, 416, 417
 fan 6
 FAQ iv
 father-in-law 43
 feature 18, 442, 455, 586
 feature register 442
 feature unification . . . 440, 441, 443
 features 440
 feminine *see* gender, feminine

- file 112
 - binary 113
 - prolog 120
 - regular expression 112
 - text 115
 - wordlist 115
- filter... viii, 265, 267, 271–273, 282, 440, 441, 461, 466
- final state *see* state, final
- finite **4, 7**
- finite-state
 - benefits 39
- finite-state automata 8
- finite-state automaton 280
- Finite-State Calculus ... iii, viii, 251
- finite-state calculus 38, 39
- finite-state grammar ix
- finite-state machine 8
- finite-state network *see* network
- finite-state relation 280
- finite-state transducer... *see* FST, 280
- Finnish .. xi, 468, 471, 482, 557, 564
- Flag Diacritic 440–475
 - C operator... *see* Flag Diacritic, operator, C
 - clear 457
 - D operator... *see* Flag Diacritic, operator, D
 - declaration..... 444
 - eliminate flag *see* xfst, eliminate flag
 - errors 452
 - feature 443
 - flag-aware 442, 457
 - implementation 441, 444
 - N operator... *see* Flag Diacritic, operator, N
 - neutral 455–458
 - non-monotonic 457
 - obey-flags 458, 472
 - one-sided 451
 - operation 454
 - operator
 - C 457, 458
 - D 456
 - N 455
 - P 455, 468
 - R 456, 467
 - U 455, 457
 - P operator... *see* Flag Diacritic, operator, P
 - preserve 463
 - R operator... *see* Flag Diacritic, operator, R
 - spelling 441, 443, 444, 455
 - awkward 453
 - template 455
 - two-sided ... 442, 451, 453, 463
 - U operation 443
 - U operator *see* Flag Diacritic, operator, U, *see* Flag Diacritic, operator, U
 - unification .. 443, 455, 457, 469
 - value 443
- Flag Diacritics... 254, 256, 290, 291
 - backtracking 454
 - composition 463–465
 - limit 450
 - obey 453
 - special 291
- flag diacritics
 - compile-replace 465, 524
 - performance 463
 - unification 454
- flag-aware *see* Flag Diacritics, flag-aware
- flag-is-epsilon *see* xfst, switch, flag-is-epsilon, *see* xfst, flag-is-epsilon
- flexibility 384
- flexible 376
- formal 599
- formal languages *see* language, formal
- formal linguistics 380
- formal power 525
- free-order morphemes 395

- French . . . vii, ix, x, 12, 13, 283, 317, 466
 frequency 433
 fsc v
 fst
 format
 tabular 309
- G**
- Gaál, Tamás v
 Gaeilge 296
 Gaelic 296
 gawk . . . *see* Unix, utility, gawk, 559
 gender 261
 feminine 230, 235
 generation 17,
 18, 40, 144, *see* xfst, com-
 mand, apply down, 145, *see*
 lexc, command, lookdown,
 see xfst, command, apply down,
 282, 284, 286, 429, 430, 441
 German ix, x, 13, 22, 377
 grammar
 lexical . . . *see* Lexical Grammar
 Grenoble v, vi
 grep *see* Unix, utility, grep
 guesser . 18, 172, 416, 526, 528, **552**
 construction 552
 guessing 561
- H**
- Halle, Morris 309
 hardware 376, 377
 harmony 469
 hashing 19, 200
 Hebrew 377
 help . . 202, *see* lexc, command, help
 lexc . . *see* lexc, command, help
 history . . *see* lexc, command, history
 housekeeping operations 199
 Hungarian 482
 hyphen 137, 491
- I**
- identity relation *see* relation, identity
 idiosyncrasy 212, 259, 397
 ifsm v
 ignoring 51
 illegal input 10
 inclusive 169
 inclusive we 593
 Indiana University vi
 Indonesian 479, 489
 infinitive 15
 infix 258, 377
 infixation x, 264, 479, 488, 491
 inflection x
 infrastructure 377
 initcap *see* capitalization, initial
 initial 533
 input 5
 tokenization 421
 inspect net 194
 inspection 194
 installation xi
 Institiúid Teangeolaíochta Éireann 296
 integration 528, 561
 Intel iii, 377
 intensifier 168
 interdigitated 501
 interdigitation . x, 264, 377, 478–480
 interface viii
 INTERLISP v
 InterLisp 317
 intermediate 280
 intersect 94, 377
 intersecting composition *see*
 composition, intersecting
 intersecting-composition . . . 254, *see*
 composition, intersecting
 intersection 20, 27,
 see regular expression, op-
 erator, &, *see* xfst, intersect
 net, 39, 48, 51, 126, 128, 240,
 287, 339, 505
 Semitic roots 513
 stems 503
 introduction vii
 Inuktitut 479

- inverse 53
 inverse replacement 67
 invert-result . . . *see* lexc, command,
 invert-result
 invert-source . . . *see* lexc, command,
 invert-source
 Inxight 441, 444
 Irish 289
 Irish Gaelic 296
 irregularity .. viii, 212, 283, 376, 397
 Italian vi, ix, x, 318
 iterate 92
 iteration 49, 240
 ITT vii
- J**
- Japanese x, 529
 Johnson, C. Douglas v, 36, 146, 309,
 312
- K**
- kaNpat 329
 Kaplan, Ronald v, 310, 312
 Kaplan, Ronald M. 317
 Karttunen
 Lauri 572
 Karttunen, Lauri v, **vi**, 212, 317, 318,
 441
 Kay, Martin v, 310, 312
 Kempe, André v
 KIMMO vi, 212
 Kleene 46
 Kleene plus .. *see* regular expression,
 operator, +
 Kleene star .. *see* regular expression,
 operator, *
 Kleene's theorem 46
 Kleene-plus 49
 Kleene-star 49
 Koskenniemi 38
 Kimmo 563, 564
 Koskenniemi, Kimmov, vi, 146, 212,
 308, 313, 317, 482, 503
- L**
- label 410
 label alphabet 189
 label net 199
 labels *see* lexc, command, labels, *see*
 xfst, command, print labels
 language **10**, 11, 21, 48, 278
 empty 21
 formal viii, 39
 intermediate 284, 285
 lexical **283**
 lower 280
 lower-side 157
 regular 47, 280
 surface **283**
 universal 22, 139
 upper 280, 427
 upper-side 157
 language intermediate 284
 Latin 242, 244
 laziness *see* creative laziness
 left-arrow rules 184
 lenition 296
 level 283
 lexc iii, v, **viii**, ix, xi, 30, 35,
 38, 146, 212–305, 319, 381,
 412, 414, 418, 419, 442, 463,
 467, 471, 528
 alternation 227
 angle bracket . . . 240, 419, 420
 angle brackets
 special character 240
 banner 214
 character
 literalized 217
 special 217
 checking 255
 command 252
 ? 214, 252, 257
 add-props 254, **254**, 274
 banner 257
 begin-script 257, **257**
 check-all .. **255**, 256, 291–295
 compile source 298

- compile-source 214, 218, 220, 231, **253**, 289
- completion 257
- compose result 289, 290
- compose-result 214, 253, **254**, 255, 289
- end-script 257, **257**
- extract-surface-forms *see* xfst, command, lower-side, **254**
- help 214, 252, **257**
- help all 257
- history 257
- invert-result *see* xfst, command, invert net, **254**
- invert-source *see* xfst, command, invert net, **254**
- labels 257
- lookdown 216, 218, 232, **255**, 298, 410, 453
- lookup 216, 218, 232, **255**, 298, 410, 453, 529
- merge-result **253**
- merge-source **253**
- props 257, 275, 295
- quit 216, 219, 232, **258**
- random 219, **255**
- random-lex **255**
- random-surf **255**
- read rules **253**
- read-result **253**
- read-rules 253, 289, 290
- read-source 253, **253**, 289
- reset-props 254, **254**, 274
- result-to-source **253**
- run-script 257, **257**
- save-props **254**, 275
- save-result **254**
- save-source 214, 216, 219, 232, 253, **253**
- source-to-result **254**
- status 252, 257, 295, 458
- storage 257
- test equivalent 225
- command menu 252
- commands 252, *see* lexc, command, ?
- comment 217
- compile 258
- compile-source 472
- compiler 212
- compose-result 463
- composition 289, 463
- continuation class 221–223, 233, 260, 264, 277
 - # 223
- display 257
- duplicates *see* lexc, switch, duplicates, *see* lexc, switch, ambiguities
- END 217
- end of word *see* lexc, continuation class, #
- entry 216
 - continuation class **217**
 - end of word 217
 - form 217
- errors 275
- explosion 217, 221, 241, 242, 244
- failures *see* lexc, switch, failures
- file
 - script **257**
- form 277
- help 257
- input-output 253
- integration 213, 278
- interface 213, 251
 - invocation 213
- invocation 216
- LEXICON 216, 221, 222, 233
 - Root 222, 223
- LEXICON Root 216, **217**
- lookup 472
- loop 225
- makefile 608
- Multichar_Symbols 242, 244, 261
- obey-flags 458

- operator
 - % 241
- optionality 225
- pitfall 242
- prompt 214
- properties **254**
- property list 254
- register 251, 252, 254, 289
 - Result 289
 - result 253
 - Rules 289
 - rules 253
 - Source 289
 - source 253
- regular expression 240
- singles . *see* lexc, switch, singles
- special character
 - literal . *see* lexc, operator, %, 242
- strategies 258
- strategy 213
- sublexicon 258
- switch 255
 - ambiguities 255, **255**
 - duplicates 255, **256**
 - failures 255, **256**
 - obey-flags **256**
 - print-space **256**
 - quit-on-fail **256**, 612
 - show-flags **256**
 - singles 255, **256**
 - time **256**
- switches 252, 458
- symbol
 - multicharacter 242
 - separate 217
- testing 255
- traps 258
- union 227
- upper:lower 238
- xfst 258
- lexical 35, 149, 242
- Lexical Grammar 283, 383, 427–430
 - documentation 430
- lexical grammar *see* Lexical Grammar
- lexical language 17, 383
- lexical string 16, 17
- Lexical Transducer . . ix, x, 283, 285, 286
- lexical transducer . . . **16**, 18, 38, 280, 384, 547
- lexicographer 38, 284
- lexicography 221, 233, 236, 283, 418
- LEXICON *see* lexc, LEXICON
- lexicon iii, viii, 35, 39, 212, 552
- lexicon compiler *see* lexc
- license 377
 - commercial xi
 - non-commercial xi
- licensing iv
- LIFO *see* xfst, stack, LIFO
- light switch 5
- Lingsoft 309
- linguistic planning 380
- linguistics 258
- Linguistics Institute of Ireland . . 296
- Linux iii, xii, 377
- long-distance 440
- long-distance dependencies *see* separated dependencies
- long-distance dependency . . 466, 564
- longest match 535
- longest-match rules 177
- lookdown 17, 18, 144, *see* lexc, command, lookdown, *see* lexc, command, lookdown, 282, 286, *see* lexc, lookdown
- lookup 12, 17, 18, 144, *see* xfst, command, apply down, *see* xfst, command, apply up, *see* lexc, command, lookup, *see* lexc, command, lookup, 281, *see* lexc, lookup
 - multicharacter symbols 545
- lookup file *see* lookup utility, lookup strategy script

- lookup script *see* lookup utility, lookup strategy script
 - lookup strategy file *see* lookup utility, lookup strategy script
 - lookup strategy script *see* lookup utility, lookup strategy script
 - lookup utility 411, 412, **539**, 528–562
 - lookup strategy script 540
 - specification 540
 - virtual composition 540
 - output 542
 - lower-casing 559
 - lower-side 157
 - lower-side language 17
 - lower-side net *see* xfst, command, lower-side net
 - lowercase 471
- M**
- machine translation 561
 - Macintosh iii, 377
 - makefile 379, 601–612
 - lexc 608
 - twolc 611
 - xfst 608
 - Malay vi, 479, **489**
 - managing 375
 - map 17
 - mapping 280
 - marking 69
 - markup rules 180
 - markup tags 559
 - measure 600
 - member 21
 - members 21
 - membership 20
 - memory 440, 442
 - merge . 253, *see* union, 505, **505**, 513
 - merge-result *see* lexc, command, merge-result
 - merge-source *see* lexc, command, merge-source
 - meta-morphotactic description . . 483
 - metalanguage viii, 22, 91, 129
 - metathesis 479, 558
 - Microlytics vi
 - Microsoft xii
 - Microsoft Word xi
 - minimize 377
 - minimize net 199
 - minus *see* xfst, minus net, *see* regular expression, operator, –, 126
 - misspelling 414
 - Mitsubishi vii
 - Modern Standard Arabic 501
 - module 376
 - Mongolian vii, 145, 278, 468
 - Monish 166–173
 - Montague Grammar vi
 - mood 261, 593
 - morpheme . **x**, 30, 141, 221, 258, 442
 - boundary 141
 - co-occurrence 440, 444
 - incompatible 444, 466, 467
 - morphemes
 - free order 395
 - morphological analyzer iii, viii, ix
 - morphological generator ix
 - morphology **x**
 - morphophoneme 141, 168
 - morphosyntax *see* morphotactics, 30, 222, 279
 - morphotactics . viii, **x**, **x**, 30, 38, 170, 222, 279, 412
 - formal power 525
 - non-concatenative 478–526
 - multi-word token 529, 533, 535, **537**
 - Multichar_Symbols . . 418, 420, 422, 424, 442
 - failure to declare 419, 420
 - inadvertent declaration 420
 - multicharacter symbol *see* symbol, multicharacter, *see* Multichar_Symbols
 - multicharacter symbols . *see* symbol, multicharacter, 486
 - apply 64, 191, 421

lookup 545
 multiple applications 384
 multiple contexts 65
 multiword tokens *see* token,
 multiword

N

name net 199
 natural language 11, 18
 Navajo vii
 needle in a haystack 417, 436
 negation 50
 network viii, ix, **4, 8**, 18, 47
 properties 198
 size .. x, xii, 440, 441, 461, 466,
 474
 network diagrams **47**
 nickel 8–10, 40, 41
 no-retokenize 519
 non-concatenative morphotactics 478–
 526
 non-final state 13, *see* state, non-final
 normalization 528, 547, 550, 561
 advanced 559
 Norwegian xi
 not-found words 410
 noun 248, 259, 589
 noun phrase ix
 NP-complete 564
 number 159, 261, 533, 591
 plural 231, 235
 number-word mapping 200
 numbers 595

O

O’Croinin, Donncha 296
 obey-flags *see* lexc, command,
 obey-flags
 open class 552
 operating system 377
 operator precedence 62
 Optimality Theory 316
 optional replacement 67

optionality 49, *see* regular expression,
 operator, (), 240
 ordinal 595
 orthography . viii, 149, 150, 376, 466
 OS X iii, 377
 overgeneration .. 248, 250, 269, 271,
 273, 278, 441, 483
 filter 395
 removing 395
 overrecognition 250, 417, 441
 overriding 406

P

parallel rules 147
 PARC iii, v–vii, 212, 441
 parentheses 49, 92, *see* regu-
 lar expression, operator, (),
 206
 parser ix, 18
 shallow iii
 parsing 528, 561
 robust 561
 shallow 19, 561
 part-of-speech 261
 part-of-speech disambiguation .. 528,
 559
 part-of-speech tagging 561
 path 14, 17, 188
 PATR-II vi
 pattern 258
 PC-KIMMO 212, 309, 318, 332
 penny 10
 performance 461, 463
 period 452
 Perl 414, 436, 559
 person 261, 593
 perspicuity 221
 phonology viii, x, 149
 pipe . *see* UNIX, pipe, 410, 530–532,
 539, 541, 545, 561
 place name 594
 planning 375, 384
 plural *see* number, plural
 plus sign 240, 242, 425

- political correctness 376
- polysynthetic 479
- pop . . . *see* xfst, command, pop stack
- pop stack . . . *see* xfst, command, pop stack
- Portuguese . . . vi, ix, x, 13, 149, 150, 317, 385, 551
- possessive 594
- Poulos, George 496
- pound sign . . . *see* lexc, end of word
- PowerPC iii, 377
- precedence . . *see* regular expression, precedence
- prefix 225, 258
- preposition 159
- prerequisites viii
- print commands 187
- print defined 193
- print label-tally 190
- print longest-string 188
- print longest-string-size 188
- print lower-words 188
- print net . . . *see* xfst, command, print net, 193
- print nth-lower 200
- print nth-upper 200
- print num-lower 200
- print num-upper 200
- print random-lower 188
- print random-upper 188
- print sigma **189**
- print sigma-tally 190
- print size 192
- print stack 193
- see* xfst, command, print stack 102
- print upper-words 188
- print words **187**
- print-space . 188, *see* lexc, command, print-space, 427
- priority union **406**, 397–406
- procedure 40
- projection 33, *see* regular expression, operator, .u, .l, *see* xfst, command, upper-side net, lower-side net, 52
- prolog 120
- pronoun 594
- pronunciation 150
- proper name 416, 471, 529
- proper noun 594, 599
- properties 198, 274
- props . *see* lexc, command, props, *see* lexc, command, props
- prune net 199
- Pseudo-Arabic 265
- pseudo-baseform 428
- punctuation 414, 529, 533, **598**
- Python 559
- Q**
- quantifier 597
- quarter 8–10, 40, 41
- question answering 561
- question marker 468
- quit *see* lexc, command, quit
- quit-on-fail *see* lexc, command, quit-on-fail, 612, *see* lexc, switch, quit-on-fail, *see* xfst, variable, quit-on-fail
- R**
- radicals 501
- RAM 377
- random . *see* lexc, command, random
- random-lex *see* lexc, command, random-lex
- random-surf *see* lexc, command, random-surf
- rate 410
- read lexc . . . *see* xfst, command, read lexc, *see* xfst, command, read lexc
- read properties 198
- read text . . . 201, *see* xfst, command, read text
- read-result *see* lexc, command, read-result

- read-rules *see* lexc, command,
 - read-rules
- read-source *see* lexc, command,
 - read-source
- recompilation 604
- reduplication 264, 479, 491
 - fixed length 479, 487
 - full stem 478, 479
 - full-stem 489
 - restricted 487
- register 376, 599
- regression testing 414, 430, 433
- regular expression viii,
 - 22, **46**, 47, 91–99, 106–112, 125–127, 419, 420
 - ? 92
 - ANY *see* regular expression, operator, ?
 - assumption 241
 - bracketing 96
 - compiler 88
 - file 112
 - lexc 240
 - operator
 - @ 287
 - | 94, 240, 419
 - () 92, 240
 - * 92, 240, 419
 - + 92, 157, 240, 419
 - 94
 - .o 143, 286, 287
 - .x 107, **108**, 157
 - : 157
 - [] 92
 - & 94, 126, 419
 - ~ 94
 - { } 94
 - ^ 92, 419
 - concatenation 94
 - precedence 96
 - semicolon 89
 - special character 240
 - % 452
 - symbol 91, 241
 - ? 48
 - 0 **109**
 - multicharacter 92
 - regular expression file *see* xfst, file,
 - regular expression
 - regular expression, operator, * 30
 - regular language *see* language, regular
 - regular relation 47
 - relation **22**, 43, 157, 278, *see*
 - finite-state relation
 - identity 25, 48
 - infinite 24
 - relative 43
 - relaxation 561
 - accentuation 551
 - spelling 551
 - replace rule viii, 38, 136, **136**, 141,
 - 142, 145–147, 149, 163, 212, 247, 279, 280, 284, 287, 289, 296, 308, 318, 319
 - arrow 139
 - context 138
 - # 140
 - beginning of word 140
 - end of word 140
 - context separator 139
 - filter 272
 - replace rules 30, 136, 173, 264
 - requirements xi
 - reset-props *see* lexc, command,
 - reset-props
 - restriction 65
 - result-to-source *see* lexc, command,
 - result-to-source
 - retokenize 519
 - compile-replace 524
 - reverse 53
 - rewrite rule iii, 36, 309
 - rewrite rule, context-sensitive 309
 - rheostat 7
 - right-arrow rules 173
 - robot 8
 - robust parsing *see* parsing, robust

- rodent 8
 - Romanian vii
 - root 157, 258
 - Arabic **501**
 - Semitic 480
 - rotate stack *see* xfst, command, rotate stack
 - round parentheses 49
 - rule . *see* replace rule, *see* twolc, 283
 - cascade 143, 145, 146, 319
 - compiler v, **317**
 - longest-match **177**
 - order 146, 319
 - parallel 319
 - replace *see* replace rule, *see* replace rule, 319
 - two-level 308
 - rule order 334
 - rules
 - backslash-slash 178
 - double vertical-bar 173
 - double-arrow 186
 - double-backslash 177
 - double-slash 173
 - hand-compilation 317
 - left-arrow 184
 - longest-match 177
 - parallel 147
 - right-arrow 173
 - shortest-match 178
 - run-script *see* lexc, command, run-script
 - runtime code 528
- S**
- SAT 564, 565
 - solution 568
 - satisfaction 564
 - save-props *see* lexc, command, save-props
 - save-result *see* lexc, command, save-result
 - save-source *see* lexc, command, save-source, *see* lexc, command, save-source, *see* lexc, command, save-source
 - Schiller, Anna 533
 - Scottish Gaelic 296
 - script .. *see* xfst, file, script, 257, 429
 - sed 559
 - semicolon 88, 217
 - Semitic 480
 - separated dependencies ... 264, **265**, 267, 279
 - separated dependency 445
 - set 20, **21**, 202
 - empty 21
 - infinite 22
 - ordered 22
 - universal 22
 - set theory 20
 - shallow parsing .*see* parsing, shallow
 - shortest-match rules 178
 - show 202
 - show-flags *see* lexc, command, show-flags
 - sigma 58, 59, **189**, *see* xfst, command, print sigma
 - sigma alphabet 59, 102, 189
 - sigma net 199
 - sigma-plus 49
 - sigma-star 49
 - SIL 309
 - simple automaton 47
 - sin
 - commission 416
 - partial omission .. 417, 418, 435
 - sin of commission 412, 417
 - sin of omission 411
 - sin of partial omission 417, 435
 - single insertion 68
 - singles .. *see* lexc, command, singles, 294
 - size 397, *see* network, size; xfst, size, 441, 448, 541, 547, 562
 - size explosion 525
 - slang **388**, 599
 - snapshot 379, 433

- Solaris iii, xii, 377
- solution
- better cola machine 621
 - Brazilian Portuguese 624
 - Einstein II 644
 - Esperanto adjectives 638
 - Esperanto nouns 637
 - Esperanto nouns and adjectives 639
 - Esperanto nouns and adjectives with tags 640
 - Esperanto nouns, adjectives, verbs 641
 - Monish 631
- solutions 613–645
- sort *see* Unix, utility, sort
- sort net 199
- source . . *see* Unix, command, source
- source file 602
- source-to-result . *see* lexc, command, source-to-result
- Spanish . . . vi, vii, ix, x, 11–18, 283, 317, 436, 551, 558
- special character . 240, *see* character, special, 452
- special symbols 419
- speed 461, 463
- spell checker 12, 18
- spell checking 431
- spelling x, 547, **557**
- tag 586
- spelling checker 557
- spelling checking 528, 561
- spelling correction 528, 561
- spelling corrector 557
- spelling reform 385, **387**
- Sproat, Richard 482
- square bracket 49
- square brackets . 49, 92–94, 96, 140, 179, 181, 206
- SRI vi
- stack . *see* xfst, stack, *see* xfst, stack, 442
- Stanford v
- start state *see* state, start, 47
- state **4**, 5
- final 9, 11, 14, 17, 42
 - non-final 42
 - start . . 8, 11–14, 17, 42, 47, 217, 223
- status *see* lexc, command, status, *see* lexc, command, status
- stem
- phonologically possible . . . 552, 554, **555**
 - Semitic 505
- storage . *see* lexc, command, storage, *see* memory
- string
- empty . 21, *see* epsilon, 109, *see* epsilon, 152, 223, 225, 239
 - intermediate 297
 - pair 329
- substitute defined 195, *see* xfst, command, substitute defined
- substitute label 194
- substitute symbol 195
- substitution 194–198
- substring net 199
- subsumption 355
- subtraction 28, *see* regular expression, operator, –, *see* xfst, minus net, 39, 48, 52, 431
- suffix 158, 168, 222, 231, 258
- Summer Institute of Linguistics 309, 318
- SUN xii
- SUNOS xii
- superlative 595
- suppletion 212, 283
- surface 35, 149, 242
- surface form 410
- surface language 17
- surface string 17
- Swedish xi
- symbol . . . **10**, 11, 13, 14, 17, 42, 91
- delimiter 18

- multicharacter . . . 15, 16, 18, 92, 98, 155, 213, 276, 418, *see* Multichar_Symbols, 441, 442, 458, 586
 - spelling 242, 425
 - pair 322
 - single 418
 - tag 427
 - symbol matching 12
 - symbol pair 48
 - symbol tokenization 191, 421
 - system chart 379
- T**
- tag 15, 16, 92, 149, 264, *see* symbol, multicharacter, 428, 543
 - abbreviation 599
 - adjective 595
 - adverb 597
 - article 597
 - aspect 593
 - case 591
 - choice 586–600
 - delimiter 598
 - derivation 597
 - determiner 597
 - directives 588
 - emphasis 599
 - major category 589
 - measure 600
 - modify 184, 589
 - mood 593
 - name 264
 - noun 589
 - number 591
 - numbers 595
 - order 427, 428
 - person 593
 - place name 594
 - pronoun 594
 - proper noun 594, 599
 - punctuation 586, 598
 - quantifier 597
 - spelling 586
 - square brackets 587
 - tagset 427
 - title 597
 - verb 593
 - voice 593
 - tag:possessive 594
 - Tagalog 377, 479, **487**
 - tagger iii, ix, 18, 122, 560
 - tagging 418
 - tags
 - order 383
 - selection 383
 - tagset 427
 - Tapanainen, Pasi v
 - template 480
 - tense 261
 - term complementation 50
 - term negation 50
 - test
 - alphabet
 - checking 418
 - boolean 187
 - regression 341
 - test equivalent . . . *see* xfst, command, test equivalent
 - testing 233, 409–437
 - automated 410
 - by hand 410
 - failure 411
 - language
 - lexical 427
 - surface 430
 - lexicon 430, 435
 - minus 427
 - negative 418
 - omission 411
 - positive 418
 - previous version 433
 - regress . . . *see* regression testing
 - regression 433–435
 - subtraction 427
 - wordlist 430, 431
 - Texas
 - University 318

- text editor xi
- Thai 529
- The Stack *see* xfst, stack
- three-way switch 7
- tilde *see* regular expression, operator,
~
- time .. *see* lexc, command, time, 529
- title 597
- toggle 6
- token 18, 410, 529
- tokenization . 18, 191, 414, 421, 528,
529, 561
 - advanced 559
 - finite-state 530
 - input strings 421
 - non-deterministic 532
- tokenize
 - apply down 64, 191, 421
 - apply up 64, 191, 421
 - input 64, 191, 421
- tokenize utility 411, 528–562
 - help 531
 - options 539
 - usage 531, 539
 - version 531, 539
- tokenizer ... iii, viii, ix, 18, 122, 410
- tokenizing transducer . 530, 532, **532**,
535
- tone spreading 71
- tr *see* Unix, utility, tr
- training vii
- transducer 8, 14, **14**, 16, 17, 47, 280,
310
 - bidirectional 146
 - lexicon 319
 - rule 319
- transition 5, 13
- transliteration 550
- trie 315
- Turkish xi, 482
- turn stack ... *see* xfst, command, turn
stack
- Two-Level Morphology .. v, vi, 313,
317, 318, 482, 487, 503
- two-level rules 38
- TwoL 212, 309, 332, 503
- twolc iii, v, vii, ix, **ix**, xi, 30,
38, 146, 212, 213, 251, 279,
280, 284, 289, 290, 308–372,
412, 418, 419, 528
 - alphabet 322–323
 - Alphabet section . 322, 334, 336
 - banner 332
 - bracket 326
 - character
 - special 327
 - command
 - compile... 291, 332, 333, 339
 - help 342
 - install-binary 339
 - install-lexicon 341–342
 - intersect 291, 339
 - lex-test 332, 339
 - lex-text 333
 - lex-text-file 340
 - pair-test 339, 340
 - pair-test-file 340
 - quit 333
 - read-grammar . 332, 333, 339
 - save-binary 290, 291, 332, 339
 - save-tabular 332
 - uninstall-lexicon 342
 - comment 330
 - compiler 320
 - complement 328
 - concatenation 326
 - contain 327
 - context 331
 - Definitions section 337
 - Diacritics 442
 - Diacritics section 337
 - file
 - binary 289
 - file format 332
 - header 336–337
 - hints 335–336
 - history 312
 - ignore 327

- interface 331–333, 339–342
 - intersection 328
 - introduction 308
 - invoke 331
 - iteration 327
 - Kleene Plus 327
 - Kleene Star 327
 - makefile 611
 - menu 332
 - methodology 329–330
 - minus 328
 - negation 327
 - optionality 326
 - rule 323–328
 - conflict 341
 - context 325–328
 - left-arrow conflict 341
 - local variable 337
 - multiple contexts 337
 - operator 324–325
 - syntax 323
 - rule order 334
 - Rule section 336
 - rule syntax 337–339
 - Sets section 336
 - source file 321
 - special character 330
 - symbol 325
 - symbol pair 330
 - syntax 321–339
 - timing 342
 - tracing 342
 - transducer
 - special case 328
 - union 326
 - word boundary 328
 - zero 326, 331
 - twolc vs. Replace Rules 330–331
 - twosided flag-diacritics 465
 - typographical error 416
- U**
- umlaut 71, 72, 177, 564
 - unconditional replacement 67
 - undefine 105
 - undergeneration 278
 - underlying 35
 - understanding 561
 - UNICODE xi
 - Unicode 550
 - unification vi, *see* Flag Diacritic, unification, 440, *see* Flag Diacritic, unification, 454
 - union 20, 25, *see* regular expression, operator, |, *see* xfst, union net, 26, 39, 51, 94, 126, 128, 158, 240, 253, 287, 384, 385, *see* priority union, 602
 - uniq *see* Unix, utility, uniq
 - Univ. of Edinburgh vi
 - Univ. of Groningen vii
 - Univ. of Texas at Austin vi
 - Univ. Paris VII vii
 - universal language 533
 - UNIX viii, 122
 - pipe 285, 286
 - utility
 - cat 285
 - sort 285
 - uniq 285
 - Unix 412, 413, 416
 - command
 - source 416
 - tr 416
 - utility
 - cat 412
 - gawk 412
 - grep 412
 - sort 413
 - tr 412
 - uniq 413
 - up 282, 284
 - upper-casing 559
 - upper-side 157
 - upper-side language 17
 - upper-side net *see* xfst, command, upper-side net

uppercase 471
 URL iv

V

verb 245, 259, 593
 version control 378, 433, 434
 vertical bar . . *see* regular expression,
 operator, |, 240
 vi viii, xi, 321
 virtual composition 540, *see*
 composition, virtual
 vocalization 480
 voice 593
 Volapük 228
 vowel 155, 157, 166
 back 166, 171
 front 166, 171
 underspecified 168
 vowel harmony 71,
 72, 166, 173–176, 458, 468,
 469, 564
 vulgar 385, **388**, 599

W

website iv
 Welsh 296
 whitespace 533
 Windows iii
 Windows 2000 377
 Windows NT 377
 word **10**
 word boundary 537
 word formation . . *see* morphotactics
 word-number mapping 200
 wordlist 115, 410, 430–432, 602
 compilation 431
 words *see* xfst, command, print
 words
 foreign 416
 write properties 199
 write text . . *see* xfst, command, write
 text

X

XeLDA 529, 562

xemacs viii, xi
 Xerox ii, iii, vi–xii, 4, 16, 30, 38, 39,
 212
 xfst iii, v, **viii**, ix, xi, 38,
 84–206, 212, 218, 219, 251,
 296, 320, 412, 418, 419, 442,
 463, 473, 528
 apropos 85
 banner 84
 batch mode 546
 command
 add properties 199
 alias 204
 apply down 89, 107, 138, 144,
 145, 150, 161, 219, 220, 410,
 453, 458
 apply up . . . 89, 107, 138, 144,
 145, 161, 219, 220, 410, 453,
 458
 apropos 85
 cleanup net 199
 clear stack 90
 compact sigma 199
 complete net 199
 compose net 85, 127, 128, 143,
 271, 286–288, 290, 298, 463,
 464
 concatenate net 127–130, 143
 crossproduct net 127
 define 88, 89, 104, 125
 determinize net 199
 echo 533
 edit properties 199
 eliminate flag . 441, 461–463,
 474
 epsilon-remove net 199
 exit 87
 help 85, 202
 intersect net 127
 invert net 127
 label net 199
 load stack . 114, 219, 220, 271
 lower-side 424
 lower-side net 277

- minimize net 199
- minus net 127, 128, 130, 143, 287
- name net 199
- negate net 127
- one-plus net 127
- pop stack 90
- print defined 193
- print labels 277, 423
- print longest-string 188
- print longest-string-size .. 188
- print lower-words 188
- print net 102, 126, 193
- print nth-lower 200
- print nth-upper 200
- print num-lower 200
- print num-upper 200
- print random-lower . 188, 434
- print random-upper . 188, 434
- print sigma 424
- print size 434
- print stack 102, 193
- print upper-words 188
- print words 89, 97, 101, 102, 111, 126, 132, 187, 188, 434
- prune net 199
- quit 87
- read lexc 213, 258, 299
- read properties 198
- read regex 88, 89, 107, 122, 128, 136, 144, 150, 163, 218, 220, 271, 298, 320, 427
- read text 115, 201, 431
- read-result 254
- read-source 254
- rotate stack 148
- save stack 113
- set 202
- show 202
- sigma net 199
- size 474
- sort net 199
- source 121, 122, 427, 429
- substitute defined . . . 195, 555
- substitute label 194
- substitute symbol 195
- substring net 199
- test equivalent 166, 220
- test lower universal 533
- turn stack 149
- twosided flag-diacritics .. 465
- undefine 105
- union net 127
- upper-side 424
- upper-side net 277
- write properties 199
- write text 432, 434, 435
- command-line flags . . . 121, 460
 - f 121
 - l 122
- command-line options 204
- command|print size **192**
- comment 113
- composition 286, 463
- exit 87
- file 112
 - binary **113**
 - regular expression .. 112, **122**, 170
 - script . 87, 121, 122, **122**, 136, 165, 166, 427, 473
 - source 170
- flag
 - e 205, 546
 - f 205
 - flush 205
 - l 204
 - pipe 205
 - q 205, 546
 - stop 205, 546
- flags 204
 - h 204
 - v 204
- help 85
- invoking 84
- makefile 608

prompt 84, 90
 quit 87
 script .. *see* xfst, file, script, 429
 stack ... 88, **100**, 107, 127–149,
 220, 252, 271, **287**, 288, 423
 clear *see* xfst, command, clear
 stack
 LIFO 104, 131
 pop .. *see* xfst, command, pop
 stack
 visualize 102
 starting 84
 startup script 204
 switch
 flag-is-epsilon 291
 traps 289
 variable
 print-space 188, 427
 quit-on-fail 612
 Xola 429
 XRCE iii, v–vii, xii

Y

Yampol, Todd v, 212, 317
 Yoruba vii

Z

Zamenhof, L.L. 228
 zero *see* string, empty, 329, 331, 340
 Zing 225