Generation of Patterns with the OPatGen Program

User Guide

David Antoš

xantos (at) fi.muni.cz
http://www.fi.muni.cz/~xantos/patlib

This is a documentation of the OPatGen (version 1.0) word hyphenation generator. OPatGen takes list of hyphenated words and creates patterns to use in TeX. OPatGen is able to handle UTF-8 encoding.

1 Introduction

The OPatGen program takes a list of hyphenated words and creates patterns that can be loaded into TEX to enable word hyphenation. OPatGen is a complete reimplementation of Frank Liang's PatGen program. It brings following advantages.

- Full Unicode (UTF-8) support, independent on system Unicode support.
- Big alphabet handling.
- Dynamic data structures, it reduces the "capacity exceeded" problem.
- "Unlimited" number of hyphenation values.
- Easier modifications.

If none of the highlights above is important for you, you may use the PatGen program instead. It is quicker. Reading the guide will not be waste of time as we cover the pattern generating topic inside-out and we refer to differences between PatGen and OPatGen programs. The differences from user's point of view are quite small. And if you run into difficulties with PatGen, you may easily switch into OPatGen paying with longer runtime only.

2 What the patterns are

If you are familiar with Appendix H of The TEXbook and you have experience with generating hyphenating patterns and using them in TEX, feel free to skip to the next section. This section describes what patterns are and how they are used to find hyphen points. If you are completely new to deal with patterns, I recommend you not only reading this guide but also to have a look at the Appendix H.

2.1 The patterns

TEX hyphenates a word first looking in the exception dictionary. If the word is not there, TEX looks for *patterns* for that word. Let's use the example from The TEXbook. Having the word `hyphenation`, TEX first extends it by special markers meaning the beginning and end of the word. Let's use a dot for that marker. So we get

`.hyphenation.`

The extended word has subwords

`. h y p h e n a t i o n .`

of length one,

`.h hy yp ph he en na at ti io on n.`

of length two, and so on.

Each subword is a pattern that defines integer values related to the desirability of hyphens in the positions between its letters. We usually show the values as numbers between letters, for example `0h0e2n0` means that the values of the `hen` subword are `0`, `0`, `2`, and `0`, where `2` is related to the position between `e` and `n` characters.

The interletter values are zero for all subwords except the ones in TEX's pattern dictionary. In this case, only the subwords

```
0h0y3p0h0  0h0e2n0  0h0e0n0a4
0h0e0n5a0t0  1n0a0  0n2a0t0  1t0i0o0  2i0o0  0o2n0
```

happen to be special patterns. TEX computes the maximum intercharacter value that occurs at each subword touching each position. The result of all the maximizations is

`.0h0y3p0h0e2n5a4t2i0o2n0.`

And the most important part: A hyphen is correct if the hyphen value is *odd*. Therefore the break-points found are `hy-phen-ation`.

We also call this type of patterns *competing patterns* as the bigger hyphenating value wins over the smaller one. Viewed in other way, the patterns hold the *context* of the hyphen point that is able to decide whether the point is/is not good to break the word at. To create most efficient patterns we want the context to be as small as possible. Very non-formally we may also say that the patterns we create may also recognize the suitable hyphen points not only in word list they were generated from but also in any word that is broken in similar way. The patterns hold *characteristics* of the breaking-point occurrence.

A similar technique may be also used to create patterns that recognize something else. Quite good results can be obtained when recognizing compound word boundaries (this can be directly done with OPatGen without modifications), typesetting long versus short s in fraktur and so on. Other applications are beyond scope of this manual.

Now the problem stands how to create the pieces of words with the ugly small numbers.

2.2 How patterns are generated

We use an iterative approach to generate the patterns. We need an input data file—dictionary with hyphen points marked with a special symbol. We use a dash for that. The typical list of words in English starts with

```
abil-i-ty
ab-sence
ab-stract
ac-a-dem-ic
ac-cept
ac-cept-able
ac-cept-ed
```

and so on.

We go through the dictionary in several *levels*. In odd levels, we create *covering* patterns, in even levels we create *inhibiting* ones. Let us recall that odd hyphenating values mean that hyphenating is allowed. We also speak about covering and inhibiting levels.

We choose pattern *candidates* at each level. The candidate choosing rule is simple: we take subwords of given length range. For example in the first level we may take candidates of lengths 2 and 3, in the second level candidates of lengths 3 to 5.

The level consists of several *passes*. The pass is the basic unit of the generating process. During the pass the input dictionary is passed just once. The pass means picking candidates of certain length and hyphen position. The passes in the level are ordered from shorter lengths to longer ones and for each length for hyphen positions in "organ-pipe fashion," it means from the middle, then the position left to the middle, right to the middle, and so on to the edges of the word. The candidate is a subword that works well and/or badly on the word. For covering levels working well means covering an allowed hyphen point and the bad counterpart is allowing wrong hyphenation. In inhibiting levels good work is inhibiting an error and bad work is inhibiting a good hyphen already found.

We store the number of cases of good and bad behaviour for each candidate as `good_count` and `bad_count`. We ignore candidates that are superstrings of either good or bad patterns at this level as they simply have no effect on the hyphenation process. The shorter candidate holds the same information as its superstring. This optimization is called *knocking out*.

After the pass is finished, candidates are selected. We use three variables to control this process—`good_wt`, `bad_wt`, and `thresh`. The *pattern choosing rule* goes as follows.

1. If the candidate satisfies

$$\texttt{good\_wt} * \texttt{good\_count} < \texttt{threshold}$$

   then we insert the candidate into patterns marked as bad, it means with odd value higher than the current level. We need it for subsequent passes, it will be removed when the level is finished.

2. If the candidate satisfies

$$\texttt{good\_wt} * \texttt{good\_count} - \texttt{bad\_wt} * \texttt{bad\_count} \geq \texttt{threshold}$$

   then the candidate is good, we insert it into the patterns with the current hyphenation value (the level number).

3. Otherwise, the candidate is thrown away and we set `more_to_come`. It means that there might still be longer patterns extending the current length and hyphen position and we will check them.

After that process various statistics are printed.

At each level, pattern is considered good if it repairs errors made by previous patterns. The `good_wt`, `bad_wt`, and `thresh` variables are local for a level. After the level is finished, the bad patterns that have been added are deleted.

When finishing the work we may have the input dictionary hyphenated by the patterns collected so far. If the number of errors is still to high for us, we may continue making another level correcting the errors of patterns from the previous runs.

Confused? Don't care, breathe deeply and read the step-by-step tutorial, where all the things you need to know are explained slowly with examples.

3 OPatGen tutorial

In this section we study an example of pattern generating process and explain it inside out, covering the features of OPatGen generator.

Convention: We sometimes highlight differences between PatGen and OPatGen in curly braces, saying {PatGen: no Unicode support}. We hope it makes switching the program easy.

3.1 First generating

We need an input data to create patterns. We often call the input data the *dictionary*. The dictionary is a sequence of words, one word on a line. The words must start at the first column, everything after the first space on the line is ignored. The allowed hyphen points are marked with dashes.

Let's have the following dictionary (those are nearly random words over the English alphabet chosen only to show you the things I want to).

```
ab-cd-efgh
cd-cde
cdc-id-cde
de-fgh
```

We have the words in the `dic` file.

3.1.1 Running OPatGen

It is time to run OPatGen. OPatGen takes four parameters, the dictionary file name, the patterns to read in, the output file name and the translate file name. The translate is a topic by itself, so we describe it separately. As the runs of OPatGen may be quite time-consuming, we may read in a set of patterns we created in previous levels, as we will see later. For the start, only the dictionary and output files are important. So we substitute the rest with `/dev/null`. Run the program with me if you can, to see the process alive.

```
opatgen dic /dev/null out /dev/null
```

The screen fills with something like this:

```
This is OPATGEN, version 0.1
... shortened ...
Translate file does not exist or is empty. Defaults used.
left_hyphen_min = 2, right_hyphen_min = 3
```

If we don't say otherwise (using the translate file), OPatGen knows the English alphabet and works in 8-bit ASCII. The translate file may also set the values of `left_hyphen_min` and `right_hyphen_min`. The values specify the number of left and right characters of each word where hyphenation is ignored. The values are language dependent and they mean the minimal number of characters that can be left at the end of a line before a hyphen and the minimal number of characters that can go to the next line after a hyphen. The default values are for PatGen compatibility, I think it makes no sense to use anything else than 1, 1

when *generating* hyphenating patterns. We may ignore borders of a word when *using* patterns, and not generating.

```
26 letters
```

The number of letters is the number of symbols in the alphabet. Each letter may have several representations in the input data, as we'll see later.

```
hyph_start, hyph_finish: 1 1
0 patterns read in
pat_start, pat_finish: 1 2
good weight, bad weight, threshold: 1 1 1
```

Here we set the values. The `hyph_start` and `hyph_finish` mean the range of levels we want to make. After specifying those values patterns from the pattern file are read in. We do not have any as we're just starting, so `/dev/null` was a good choice as we have nothing to read in. The patterns to read in may contain only hyphenating values less than `hyph_start`.

The `pat_start` and `pat_finish` control the range of lengths of the patterns. The values 1 and 2 mean we take candidates of length one and two. The last three variables control the pattern choosing process, the choosing rules have been described above.

```
Generating level 1

Generating a pass with pat_len = 1, pat_dot = 0
```

First the patterns with length one are created, starting with the hyphen (we often say *dot*) position after the zeroth character of the pattern. It sounds quite stupid, nevertheless it is a good way to refer to positions. It simply means the leftmost position of the word, the candidates we deal with look like `1x`, where `x` is a character.

```
0 good 0 bad 6 missed
0 % 0 % 100 %
```

The numbers denote the numbers of cases when the patterns act well, badly, and/or miss finding a hyphen point. The very first pass always misses everything, of course. The percent counts are related to the sum of good and missed, therefore the sum of the line does not have to give 100.

```
Count data structure statistics:
nodes:                          28
patterns:                       5
```

```
trie_max:                          28
current q_max_thresh:              3
```

The statistics of the internal structures, the most interesting thing is the count of the patterns, the rest of values needs to know quite a lot about the internal work of the generator.

```
Collecting candidates
3 good and 1 bad patterns added (more to come)
finding 5 good and 1 bad hyphens
efficiency = 1.25

Pattern data structure statistics:
nodes:                             28
patterns:                          4
trie_max:                          28
current q_max_thresh:              5
number of different outputs:       3
```

Now the candidates are collected. Good and bad ones are added and there were candidates not satisfying the collecting conditions, it means there might still be good patterns longer than current ones. This is indicated by the `(more to come)` text. Numbers of found good and bad hyphens appear and the efficiency is printed.

The efficiency is computed as follows. Let `good_count` be the number of good cases of acting of patterns, `bad_count` the number of erroneous cases, and `good_pat_count` the number of good patterns. Then the efficiency is calculated as

$$\texttt{bad\_eff} = \frac{\texttt{thresh}}{\texttt{good\_wt}}$$

$$\texttt{efficiency} = \frac{\texttt{good\_count}}{\texttt{good\_pat\_count} + \frac{\texttt{bad\_count}}{\texttt{bad\_eff}}}$$

Let's come back to the generating process.

```
Generating a pass with pat_len = 1, pat_dot = 1

5 good 1 bad 1 missed
83 % 16 % 16 %
Count data structure statistics:
nodes:                             28
```

```
patterns:                       3
trie_max:                       28
current q_max_thresh:           3
Collecting candidates
0 good and 2 bad patterns added (more to come)
finding 5 good and 1 bad hyphens
Pattern data structure statistics:
nodes:                          28
patterns:                       4
trie_max:                       28
current q_max_thresh:           5
number of different outputs:    4
```

Now we generate a pass looking for patterns `x1`. It turns out there is nothing good here we can add.

```
Generating a pass with pat_len = 2, pat_dot = 1

5 good 1 bad 1 missed
83 % 16 % 16 %
Count data structure statistics:
nodes:                          29
patterns:                       1
trie_max:                       29
current q_max_thresh:           3
Collecting candidates
0 good and 0 bad patterns added (more to come)
finding 5 good and 1 bad hyphens
Pattern data structure statistics:
nodes:                          28
patterns:                       4
trie_max:                       28
current q_max_thresh:           5
number of different outputs:    4
```

Candidates `x1y` are examined.

```
Generating a pass with pat_len = 2, pat_dot = 0

5 good 1 bad 1 missed
83 % 16 % 16 %
Count data structure statistics:
nodes:                          29
patterns:                       1
```

```
trie_max:                       29
current q_max_thresh:           3
Collecting candidates
0 good and 0 bad patterns added (more to come)
finding 5 good and 1 bad hyphens
Pattern data structure statistics:
nodes:                          28
patterns:                       4
trie_max:                       28
current q_max_thresh:           5
number of different outputs:    4

Generating a pass with pat_len = 2, pat_dot = 2

5 good 1 bad 1 missed
83 % 16 % 16 %
Count data structure statistics:
nodes:                          29
patterns:                       1
trie_max:                       29
current q_max_thresh:           3
Collecting candidates
0 good and 0 bad patterns added (more to come)
finding 5 good and 1 bad hyphens
Pattern data structure statistics:
nodes:                          28
patterns:                       4
trie_max:                       28
current q_max_thresh:           5
number of different outputs:    4
```

And finally the candidates `1xy` and `xy1` are tested. Note that none of them added anything useful. Have a detailed look at the output and check carefully what happens.

```
1 bad patterns deleted
total of 3 patterns at level 1
```

During the first level one bad candidate has been added. It is deleted now, when the level ends.

```
hyphenate word list <y/n>? y
Writing file pattmp.1
```

```
5 good 1 bad 1 missed
83 % 16 % 16 %
```

The final question is if we want to see the work of the new-born patterns on the dictionary file. We want to. So the words of the dictionary are hyphenated with patterns we have and the result is written into `pattmp.n` file, where `n` is the last level number. The patterns we have are written into the output file. The patterns act five times well, make one error, and can't find one of good hyphen points. The percent counts are again related to the sum of good and missed hyphens.

Let's now have a look at the results. The patterns we created are

```
1c
1e
1i
```

and the hyphenated dictionary in the `pattmp.1` goes

```
ab*cd*efgh
cd*cde
cd.c*id*cde
de-fgh
```

The hyphens we find are marked with '`*`', the bad ones (we find and they are wrong) with '`.`', and the ones we miss with '`-`'.

Please have a look at the patterns and the output and try to hyphenate the words using the patterns yourself.

3.1.2 Adding more levels

The patterns are not as good as they might be. They make an error. Let us add the second level, the inhibiting one. The even levels correct errors, the odd ones add hyphenating points. First we copy the `out` file into the `pat`, so as not to have to generate the first level again. Now we start OPatGen with the pattern file name `pat`.

```
opatgen dic pat out /dev/null
```

Now the OPatGen's output will be much more shortened, as I do not like manuals over 500 pages. Let's set the values, we want to generate the second level, and we want to deal with candidates of lengths two and three. Now we slightly prefer good patterns over bad ones, therefore we set the weights to 1, 2, and 1.

```
... shortened ...
hyph_start, hyph_finish:  2 2
3 patterns read in
pat_start, pat_finish: 2 3
good weight, bad weight, threshold: 1 2 1
```

Here we go. We start with patterns of length two and continue with length three, the dot positions are ordered in "organ pipe" fashion for each length.

```
Generating level 2

Generating a pass with pat_len = 2, pat_dot = 1

5 good 1 bad 1 missed
83 % 16 % 16 %
...
Collecting candidates
0 good and 3 bad patterns added (more to come)
finding 5 good and 1 bad hyphens
...

Generating a pass with pat_len = 2, pat_dot = 0

5 good 1 bad 1 missed
83 % 16 % 16 %
...
Collecting candidates
1 good and 3 bad patterns added
finding 6 good and 1 bad hyphens
...

Generating a pass with pat_len = 2, pat_dot = 2

5 good 0 bad 1 missed
83 % 0 % 16 %
...
0 good and 4 bad patterns added
finding 5 good and 0 bad hyphens
...
```

Wow, where is the length three we wanted? The length is silently skipped as there was no more to come, in human words, we know there can't be longer patterns extending the ones we created. So we do not waste time to check them again.

We have taken some bad patterns, we delete them now. We added just one pattern to our set. And we want the word list to be hyphenated.

```
7 bad patterns deleted
total of 1 patterns at level 2
hyphenate word list <y/n>? y
Writing file pattmp.2

5 good 0 bad 1 missed
83 % 0 % 16 %
```

Now the patterns and the hyphenated list are:

```
1c
2ci
1e
1i

ab*cd*efgh
cd*cde
cdc*id*cde
de-fgh
```

What an improvement! We reduced the number of errors from one to zero! Now we only miss one hyphen. We may correct it adding one more level, the third, covering one.

We again copy current outputs to the pattern file and repeat calling OPatGen with the patterns to read in. Now we set the hyphenation level to 3, the length range from 3 to 3 (do you see it's enough?), and the parameters to 1, 10, and 1. This is how we say that we want patterns that do not have to cover many points, nevertheless if they make an error, they are heavily penalized for that.

```
...
hyph_start, hyph_finish: 3 3
4 patterns read in
pat_start, pat_finish: 3 3
good weight, bad weight, threshold: 1 10 1

Generating level 3
...
...
```

```
total of 1 patterns at level 3
hyphenate word list <y/n>? y
Writing file pattmp.3

6 good 0 bad 0 missed
100 % 0 % 0 %
```

Hooray! Complete success! We cover all the hyphen points and make no errors at all, let's have a look at the patterns.

```
.de3
1c
2ci
1e
1i
```

What is the dot now? The dot in the pattern file is a special character meaning the *edge of a word*. Such a pattern matches only the words *starting* with `de`. The dot may also appear at the very end of a pattern.

Final notes: If you have a real dictionary with thousands of words, do not expect the covering of hyphen points to be complete. There will be errors that can be corrected adding more levels or using the exception dictionary. And note that you may generate several levels at a time giving the level range to the first OPatGen's question.

Now try generating patterns with different lengths than I did and with different parameters and check the results carefully.

We sometimes need a word list to be hyphenated *without* pattern generation itself, for example if we want to test the patterns on another word list that they were created. So OPatGen allows a special setting of the level range to achieve the effect. If the `hyph_finish` is smaller than `hyph_start`, the patterns are read in, there is nothing to generate, and OPatGen asks whether to hyphenate the word list.

3.2 Parameters, weights, and relatives

The important question we have not discussed in the previous overview is how to set the generating parameters `good_wt`, `bad_wt`, and `threshold`. There is no simple answer to that. More precisely, the simple answer is "nobody knows." Setting the parameters is the most interesting part of the generating process, it is heavily input data dependent. The problem is more than twenty years old and there is no theoretical framework for that.

Generating of patterns needs some experience and intuition. Now I put only several remarks what you can expect in general. We write the `good_wt`, `bad_wt`,

and `threshold` values as three numbers to be short, so (1, 10, 4) means `good_wt` to be 1, `bad_wt` 10, and `threshold` 4.

Let us start with `bad_wt`. If that value is low (related to the threshold), you allow patterns to make errors. This may be good in first level if you want to cover as much as possible. In higher levels, the setting like (1, very high number like 1000 or so, 1) can be often found, making the patterns to be highly penalized for an error. The `good_wt` is often set to a small number like 1, 2, or 3. For example, setting (1, 2, 20) may be quite nice for first levels, as it takes patterns that are good 20 times with no error, or 22 times with one error and so on. This may be suitable for short patterns, for longer patterns it would miss quite good and error-less patterns if they appear less than 20 times. Another often seen settings are (1, 5, 1), penalizing errors, or (1, 4, 7), preferring patterns covering more points.

Another problem is how to change the pattern length range. For our application patterns can be quite short, 1 to 7 characters for languages like English, a bit more for German, as an example of a language with longer words. Usual setting is 1–3 for first and second level, slowly increasing to 4–6 for the fifth level. Some pattern creators don't like patterns of length 1 and start from 2. In general, the shorter the patterns are, the quicker their usage is.

There is no golden rule. Read some articles summing up the experiences with generating patterns for various languages, there can be found elsewhere.

One small complication can make adjusting the parameters a bit more difficult. The words in our dictionary can be *weighted*. If there is a number at any interletter position in the input data, the position is counted as many times as the number says. It brings the possibility to weight some words in our dictionary more heavily to make their hyphenation more important. For example

```
hy-2phen-a3-ti7on
```

means that the position between y and p will be counted as it appeared twice and the position between a and t as it appeared three times. The position between i and o is counted seven times. Also note it makes no difference if we put the number before or after the hyphenation mark.

The weight may be a natural number, not only a digit. {PATGEN: only one-digit weights are allowed.}

There is a useful exception. If the number appears in the very beginning of the word it means the *global word weight* that is valid until it is changed. After a global weight we represent all positions of all following words as having that weight unless the position itself sets something else. Have a look at the example. Also note the using of global weight 1 to turn back to the defaults.

```
ab-c2d
3qw-erty
u4i-op
1ef-gh
```

will be represented as (we don't put down the default weight 1)

```
ab-c2d
q3w3-e3r3t3y
u4i3-o3p
ef-gh
```

This feature may be useful if you want to prefer correct work of your patterns on a subset of the dictionary over the rest, for example according to the frequency of words in the language.

When hyphenating a word list the weights are copied into the `pattmp.n` file. They are copied in the "minimal" form, the form of the dictionary file doesn't have to be preserved.

3.3 Defining our alphabet

We can generate patterns now. But we used only English alphabet for that, there are many languages using accents and more than the twenty-six symbols. We may use two approaches to handle that problem, the first one is to use the *escape sequences*, the other is Unicode. We may also combine the two things together. Using Unicode obsoletes having escape sequences to represent letters in TEX, nevertheless we provide this feature for PatGen compatibility, even though it complicates the program considerably.

What we need to now is a *translate file*. The translate file controls the alphabet we use. The first line of the translate file is special. It sets the values of `left_hyphen_min` and `right_hyphen_min` variables in the first two and second two columns. If those values are invalid, OPATGEN will ask for them interactively. The remaining three columns of the line, namely the fifth to the seventh, may define replacements for the `.`, `-`, and `*` characters to be used in the word list. The replacement characters may be 7-bit ASCII values. The rest of the line is ignored. The replacements might be useful if you want to use some of that characters to denote an accent.

The rest of the file defines the letters of the alphabet of the language. Note that if the translate file is empty, the defaults are used. If the translate file is not empty, you must put *all the alphabet you use* into it, including the default `a` to `z` symbols (if they appear in your input data, of course). We need it to store the words efficiently.

Each line contains a delimiter in the first column, this is a character not occurring in any representation of the letter on the line. The delimiter is any 7-bit ASCII value. The delimiter is followed by any number of representations of the letter. The representations are separated by the delimiter. The very first representation of the letter in the line is called *primary* or *lowercase*, the other ones

are *secondary* or *uppercase*. The names come from the fact that TEX hyphenates words temporarily converted to their lowercase forms. Any of that forms may be used in input files, but for OPatGen all of them have one internal code. When OPatGen is writing the letter into a file, it uses the lowercase form only. {PATGEN: There must be double delimiter to finish the last escape sequence in the end of the line.}

Anything after double delimiter is a comment, either at the very beginning of the line or anywhere else. Empty lines are ignored.

What the representation of the letter may be depends of OPatGen's mode. If the mode is ASCII (the default we used in our examples), the letter may be an 8-bit ASCII value or escape sequence created out of 8-bit ASCII values. The Unicode mode is specified by the `-u8` switch as the first parameter of the command line. In the Unicode mode the letter representations may be 7-bit ASCII values, UTF-8 multibyte characters, and escape sequences made out of 7-bit ASCII values. We highly recommend using only 7-bit ASCII characters in the escape sequences in any case.

The escape sequence starts with an escape character. If a character is used as escape it may not be used as an ordinary character. The rest is a sequence of letters and characters that are used nowhere else (invalid characters). You may not use digits, escapes, or hyphen characters in the escape sequences. Let us have an example of escape sequences.

```
␣a␣A␣\mya
```

defines `\mya` as equivalent to a and A. The `\` character is an escape character. Having that line in our translate, defining `abb` escape sequence is invalid as the a character is a letter. We may define a `|bb` sequence. The `|` character has not been used before.

If the escape sequence occurs in the input file, it must be followed by a number, a hyphenation character, an escape sequence, end of line, or at least one space. We must be able to recognize its end. {PatGen: no spaces, the escape sequence must not be prefix of another one.} The spaces after the escape sequence are completely ignored, which is similar to TEX's reading input routine. The escape sequence is recognized only if it starts with the same escape character as it was defined in the translate file. For example, having `\` and `|` escape characters, then `|mya` won't be recognized as representation of `a`! Moreover you may define `\mya` and `|mya` to be two different escape sequences. This differs from TEX and I hope I don't have to say I strongly vote against doing this.

Let us have an example of a translate file.

```
␣1␣1
**␣I␣am␣a␣comment.
␣a␣A␣\mya␣\myA
```

```
␣b␣B␣␣I␣am␣a␣comment␣after␣two␣spaces.
#p#P#\varphi
```

The first line sets the left and right minimal hyphenation values to ones. The third line defines the letter `a`. This letter may be written in input data as `A`, `\mya`, or `\myA`. The fourth line defines an ordinary `b` letter. The last line of our example is analogical to the second one, we only demonstrate the usage of non-space delimiter. Note that the `\varphi` is not followed by space, otherwise it would not be recognized in a word like `\varphi-a`! It's a good idea to finish the line with double delimiter to prevent trailing spaces to make hard-to-find errors. We also recommend using a *visible* delimiter. The author once spent several hours debugging the program to finally find out he had double space in his translate. The syntax is efficient but it lets you easily shoot in your leg.

{PatGen: escape sequences were usually followed by the space character in the translate file to make the syntax of input files closer to TeX's one. It made lots of problems that lead to `Bad representation` errors without identifying the line. I consider it quite ugly. The translate file handling in PatGen was added later to make it able to handle features of "8-bit TeX" and is full of beautiful programming tricks. I consider it to be the least readable part of PatGen.}

The input data may now contain for example words like `a\varphi-\myA b` and `aP-\mya b`, they are both equivalent to `ap-ab`. The sequence `a\myAP` is invalid as there is no way to recognize the end of the escape sequence. But `a\myA\varphi` is correct, so is `a\mya-\myA2b`.

OPatGen decides whether the files are in Unicode or ASCII only according to the `-u8` switch. No locale or other system setting is taken into account to be able to handle Unicode on systems that don't support it. As forgetting the `-u8` is a common mistake (at least I forget this very often), the error message (that seems it has nothing to do with this problem at first sight) also reminds this possible problem. The `-u8` switch must be the first parameter of the command line.

OPatGen also tries pretty hard to check the consistency of the translate file. If an error occurs OPatGen informs the user reasonably what happens.

The order of lines in the translate file controls the "alphabetical order" of symbols in the output. The output files will be created in that order except the `pattmp.n`. That file keeps the order of the dictionary.

4 Small but useful tools

4.1 `dic2traskelet`

In order to create a list of all characters occurring in the dictionary file, you can use a tool named `dic2traskelet`. This program can be found in the `tools`

directory. It produces a simple list of characters that appear in the file in simple "binary" order. You can use this as a base to create the translate file, with no risk of forgetting a character.

The `dic2traskelet` program is called using two or three parameters, if the first is `-u8`, it switches into UTF-8 mode. The following two parameters are file names of the dictionary file and of the translate skeleton.

4.2 `opgwrap` and `opglog2rep`

The `opgwrap` utility is an OPatGen wrapper. It takes the file names to deal with and the level parameters and calls OPatGen repeatedly. Each run is logged, therefore you may see exactly what happens. Moreover it always hyphenates the word list. Use `opgwrap --help` to exact explanation and examples.

The wrapper produces logs with names like `log.1`, `log.2`, etc. It is very useful to see the final results of the runs, it means the final covering information. Therefore we have a small tool named `opglog2rep` (for OPatGen log to report), it takes the logs, the starting number, and the name of the output file. Then it fills the output file with the final results of the logged runs, more precisely with several final lines of logs. If the word list haven't been hyphenated in certain run, it just adds some unuseful rubbish.

Both the programs are simple Perl scripts, not very intelligent but may be useful. They were tested on Un*x platforms and the report maker uses the `tail` tool.

5 Invoking OPatGen

- `opatgen --help`

  prints usage help and quits

- `opatgen --version`

  prints version info and quits

- `opatgen [-u8] DICTIONARY PATTERNS OUTPUT TRANSLATE`

  asks for parameters interactively and generates patterns using dictionary, reading patterns before start, writing to output file and all that as translate controls. If `-u8` is set, all the files are in UTF-8 encoding, otherwise 8-bit ASCII.

6 Dealing with bugs

If you find a bug in the OPatGen program or its documentation, please report it to the author and maintainer, `xantos (at) fi.muni.cz`. Describe the data you have problem with and the conditions and parameters when the program fails. Also add the version number, preferably the CVS revision ID, information about your platform and compiler. Volunteers to improve my English are also welcome.

The software is far from perfect. If you have any questions, suggestions, notes, or just anything you want to tell, feel free to contact the author. I'd be really happy to hear of you. Your notes will be taken seriously, this differs from most commercial software.

7 Credits

I would like to thank to

- Petr Sojka, my adviser. He taught me all the basics about pattern generating and helped me very much with analysing the program and its implementation. He always wants more than I am able to do; I am sure this permanent tension made this program better.

- My parents who were walking around silently when I was chewing my pen and hitting the keyboard.

- My friends who didn't ask too often how things go.

- All the people who develop free software, don't wont me to put down their names, it would be looooong.