◇ ◇ ◇

*Pattern Generation Revisited**

David Antoš, Petr Sojka
Faculty of Informatics, Masaryk University Brno
Email: {xantos|sojka}@informatics.muni.cz

abstract.

The program PatGen, being nearly twenty years old, doesn't suit today's needs:

- ◇ it is nearly impossible to make changes, as the program is highly optimised (like TEX),
- ◇ it is limited to eight-bit encodings,
- ◇ it uses static data structures,
- ◇ reuse of the pattern technique and packed trie data structure for problems other than hyphenation (context dependent ligature handling, spell checking, Thai syllabification, etc) is cumbersome.

Those and other reasons explained further in the paper led us to the decision to reimplement PatGen from scratch in an object-oriented manner (like NTS–New Typesetting System reimplementation of TEX) and to create the PATtern LIBrary PatLib and the (hyphenation) pattern generator based on it.

We argue that this general approach allows the code to be used in many applications in computer typesetting area, in addition to those of pattern recognition, which include various natural language processing, optical character recognition, and others.

keywords: patterns, Unicode, hyphenation, tagging, transformation, Omega, PatGen, PatLib, reimplementation, templates, C++

Introduction

The ultimate goal of mathematics is to eliminate all need for intelligent thought.
— Graham, Knuth, Patashnik [2, page 56]

The ultimate goal of a typesetting engine is to automate as much as possible of what is needed for a given design, allowing the author to concentrate on the content of the text. The author maps her/his thoughts in *linear* writing, a sequence of *symbols*. Symbols (characters, words or even sentences) can be combined

*Presentation of the paper has been made possible with the support of EuroTEX bursary fund. This research has been partially supported by the Grant CEZ:J07/98:143300003.

into *patterns* (of characters, words or sentences). Patterns describe "higher rules" and dependencies between symbols, depending on *context*.

The technique of covering and inhibiting patterns used in the program PatGen [11] is highly effective and powerful. The pattern technique is an effective way to extract information out of large data files and to recognise the structures again. It is used in TEX as an elegant and language-independent solution for high-quality word hyphenation. This effective approach found its place in many other typesetting systems including the commercial ones. We think this method should be studied well, as many other applications are possible, in addition to those in the field of typesetting and natural language processing.

The generation of hyphenation patterns using the PatGen program does not satisfy today's needs. Many generalisations are needed for wider use. The Omega system [6, 12] was introduced. One of it's goals is to make direct typesetting of texts in Unicode possible, hence enabling the hyphenation of languages with more than 256 characters. An example of such a language is Greek, where 345 different combinations of Greek letters with accents, breathings, syllable lengths and the subscript iota are needed [5]. Therefore, Omega needs a generator capable of handling general/universal hyphenation patterns. Those new possibilities and needs in computer typesetting, together with the detailed analysis described below, led us to revise the usage of pattern recognition and to design new software to meet these goals.

The organisation of the paper is as follows. The next section (page 8) defines the patterns, using a standard example of hyphenation. Then an overview is given (page 9) of the process of pattern generation. The following section (page 10) describes one possible use for patterns and is followed by a section (page 11), in which the limitations for exploiting the current version of PatGen are argued.

The second part of this paper starts with a section (page 12) which describes the design of the new software library for pattern handling. Then packed digital trees, the basic data structure used in PatLib, are presented (page 12). Some thoughts about implementing the translation/tagging process using pattern based techniques are summarised in the section on page 15. The final section (page 16) contains a summary and suggestions for future work.

Patterns

> *Middle English* patron *'something serving as a model', from Old French. The change in sense is from the idea of a patron giving an example to be copied. Metathesis in the second syllable occured in the 16th century. By 1700* patron *ceased to be used on things, and the two forms became differentiated in sense.*
> — Origin of word *pattern*: [3]

Patterns are used to recognise "points of interest" in data. A point of interest may be the inter-character position where hyphenation is allowed, or the border between water and forest on a landscape photograph, or something similar. The

pattern is a sub-word of a given word set and the information of the points of interest is written between its symbols.

There are two possible values for this information. One value indicates the point of interest *is* here, the other indicates the point of interest *is not* here. Natural numbers are the typical representation of that knowledge; odd for yes, even for no. So we have *covering* and *inhibiting* patterns. Special symbols are often used, for example a dot for the word boundary.

Below we show a couple of hyphenation patterns, chosen out of the English hyphenating pattern file. For the purpose of the following example, we deal with a small subset of the real set of patterns. Note that the dot represents a word boundary.

```
.li4g    .lig5a    3ture    1ga    2gam
```

Using the patterns goes as follows. All patterns matching any sub-word of the word to be hyphenated are selected. Using the above subset of patterns with the word "ligature" we get:

```
. l i g a t u r e .
. l i4g
. l i g5a
        3t u r e
    1g a
```

The pattern `2gam` matches no sub-word of "ligature". The patterns *compete* and the endresult is the maximum for inter-character positions of all matching patterns, in our example we get:

```
. l0i4g5a3t0u0r0e .
```

According to the above we may hyphenate `lig-a-ture`.

To sum up: with a "clever" set of patterns, we are able to store a mapping from sequences of tokens (words) to an output domain — sequence of boolean values —, in our case positions of hyphenation points. To put it in another way: tokens (characters) emit output, depending on the context.

For a detailed introduction to TEX's hyphenation algorithms see [8, Appendix H]. We now need to know how patterns are generated to understand why things are done this way.

Pattern Generation

> *An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is going on inside, although he may still be able to some extent to predict his pupil's behaviour.*
> — Alan Turing, [16]

Generating a *minimal* set of competing patterns completely covering a given phenomenon is known to be NP-complete. Giving up the minimality requirement, we may get surprisingly good results compressing the *input data* information into a pattern set iteratively. Let us now describe the generating process.

We need a large input data file with marked points of interest. Hyphenating words, we use a large dictionary with allowed hyphenation points. Now we repeat going through the data file in several *levels*. We generate covering patterns in odd levels and inhibiting ones in even levels.

We have a rule how to choose *pattern candidates* at each level. In our case it may be "an at most k characters long substring of the word containing the hyphenation point". We choose pattern candidates and store them into a suitable data structure. Not all candidates are good patterns, so we need a *pattern choosing rule*. Usually we remember the number of times when the candidate helps and spoils finding a correct hyphenation point. We always test new candidates according to all patterns selected so-far. We are interested in the functionality of the whole set. The pattern choosing rule may be a linear function over the number of good/bad word efficiency of the candidate compared to a threshold. This heuristic is used in PatGen, but other heuristics may lead to better (e.g. with respect to space) pattern sets with the same functionality. The candidates marked as good by the previous process are included into the set of patterns. The pattern set still makes mistakes. We continue generating another level, an even level this time, when we create inhibiting patterns. The next level will be covering and so on. A candidate at a certain level is good if it repairs errors made by previous levels.

This is also the way how PatGen works. A PatGen user has no chance to change the candidate and/or pattern choosing rules, which are similar to the ones previously described. Hyphenating patterns for TEX have been created for several dozens of languages [15], usually created from a list with already hyphenated words. There are languages where the patterns were created by hand, either entirely, or in part.

How successful is this technique? The natural language dictionary has several megabytes of data. Out of such a dictionary patterns of tens of kilobytes may be prepared, covering more than 98 % of the hyphenation points with an error rate of less than 0.1 %. Experiments show that four or five levels are enough to reach those parameters. Using various strategies of setting linear threshold parameters we may optimise the patterns to size, covering ratio and/or errors [13]. As not many lists with hyphenated words are publicly available for serious research on pattern generation heuristics, we think that most available patterns are suboptimal. For more information on pattern generation using PatGen have a look at tutorial [4].

Tagging with Patterns

The solution of the hyphenation problem and the techniques involved have been studied extensively [15] and together with long-lasting usage in TEX and other typesetting systems, their advantages have been verified. The application of the techniques of bootstrapping and stratification [13, 14] made them even more attractive. However, to the best of our knowledge, sofar nobody has suggested and used a context dependent task for the resolution of other ambiguities.

We may look at the hyphenation problem as a problem of *tagging* the possible hyphenation positions in finite sequences of characters called words. On a different level of abstraction, the recognition of sentence borders is nothing more than "tagging" the begins and ends of sentences in sequences of words.

Yet another example: in quality typography, it is often necessary to decide, whether a given sequence of characters is to be typeset as a ligature (such as ij, fi, fl) and not as separate characters (ij, fi, fl). This ambiguity has to be resolved by the tagging of appropriate occurences, depending on the context: ligatures are e.g. forbidden on compound word boundaries.

All these tasks (and many others, see page 15) are "isomorphic" — the same techniques developed and used for hyphenation may be used here as well. The key issue in applicability of the techniques for the variety of context-dependent tagging tasks is the understanding and effective implementation of the pattern generation process. The current implementation of PatGen is not up to these possible new uses.

PatGen Limitations

What man wants is simply independent choice,
whatever that independence may cost
and wherever it may lead.
— Fedor Dostoevsky, *Notes from Underground* (1864)

The program PatGen has several serious restrictions. It is a monolithic structured code, which, although very well documented (documented Pascal, WEB), is very difficult to change. PatGen is also "too optimised", necessary to make it possible to run in the core of the PDP-10, so understanding the code is not easy. In this sense PatGen is very similar to TEX itself. The data structures are tightly bound to the stored information: high-level operations are performed on the data structures directly without any levels of abstraction.

The data structures of PatGen are hardwired for eight-bit characters. Modification to handle more characters — full Unicode — is not straightforward. The maximum number of PatGen levels is nine. When generating patterns, you can collect candidates of the same length at the same time only. The data structures are static, running out of memory requires the user to change constants in the source code and recompile the program.

Of course PatGen may be used to generate patterns on other phenomenons besides word hyphenation, but only if you transform the problem into hyphenation. This might be non-trivial and moreover, it's feasible only for problems with small alphabets, less than approximately 240 symbols (PatGen uses some ascii characters for special and output purposes).

PatLib

My library was dukedom large enough.
— Shakespeare, The Tempest (1611), act 1, sc. 2 l. 109

We decided to generalise PatGen and to implement the PATtern LIBrary PatLib for general pattern manipulation. We hope that this will make the techniques easily accessible. A Unicode word hyphenation pattern generator is the testbed application.

For portability and efficiency reasons we chose C^{++} as the implementation language. The C^{++} code is embedded in CWEB to keep the code documented as much as possible. Moreover the code "patterns" called *templates* in C^{++} let us postpone the precise type specification to higher levels of development which turned out to be a big advantage during the step-wise analysis. We do hope that templates increase flexibility of the library.

The PatLib library consists of two levels, the finite language store (which is a finite automaton with output restricted to finite languages, implemented using packed trie) and the pattern manipulator. The language store handles only basic operations over words, such as inserting, deleting, getting iteratively the whole stored language and similar low-level operations. The output of a word is an object in general, so is the input alphabet.

The pattern manipulator handles patterns, it means words with multiple positioned outputs. We also prepared a mechanism to handle numbers of good and bad counts for pattern candidates.

The manipulator and the language store work with objects in general, nevertheless to keep efficiency reasonable we suggest to use numbers as internal representation for the external alphabet. Even if the external alphabet is Unicode, not all Unicode characters are really used in one input data file. So we can collect the set of all used characters and build a bijection between the alphabet and the internal representation by numbers $\{1, \ldots, n\}$, where all the numbers are really used.

We separated the semantics from the representation. We don't have to care what the application symbols are. An application using this library may implement any strategy for the generation of patterns.

Of course we have to pay for more generality and flexibility with performance loss. As an example, the output of a pattern in PatGen is a pointer to a hash table containing pairs $\langle \text{level\_number}, \text{position} \rangle$, we must have an object with a copy constructor. At the time of writing of this article we are unable to give an indication of the performance ratio.

Packed digital tree (trie)

Gentle reader, if you are not interested in programming or data structures, feel free to skip this section. It will do no harm for understanding the rest of the article. The trie data structure we use to store patterns is quite known.

Its practically usable variant — being described only seldom in programming books — is much less known.

A trie is usually presented and described as in [9]: it is an m-ary tree, its nodes are m-ary vectors indexed by a sorted finite alphabet. A node in depth l from the root corresponds to the prefix of length l. Finding a word in a trie starts at the root node. We take the next symbol of the word, let it be k. Then the kth member of the node points to the lower level node, which corresponds to the unread rest of the word. If the word is not in the trie, we get the longest prefix.

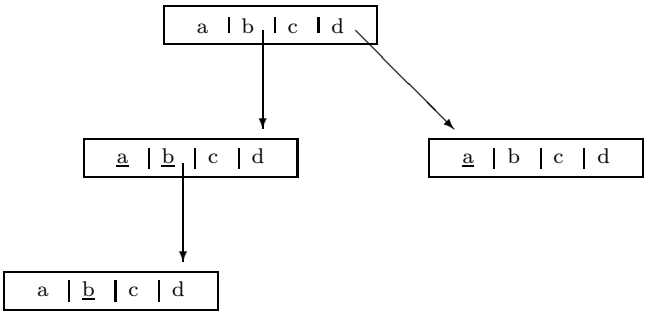

FIGURE 1: TRIE — AN EXAMPLE

Figure 1 shows a trie containing the words *ba*, *bb*, *bbb*, and *da* over the alphabet $\{a, b, c, d\}$. Underlining indicates the end of a word.

It is not difficult to implement this data structure. Nodes may be put into a linear array one by one, pointers going to the start of the next nodes. But this approach wastes memory, especially if the words are long and the nodes sparse. Using dynamic memory does not change this property.

The advantage of a trie is that the time needed for the look-up and inserting of a word is *linear to the length of the word*, this means the time needed does not depend on the amount of words stored.

The need for memory may be reduced (paying with a small amount of time), as shown by Liang in [10]. In practical applications the nodes are sparse, hence we want to store them *mixed into one another* into a linear array. One node uses the fields which are left empty by another node.

When working with this structure, we must have a way to decide which field belongs to a certain node. This may be done with a little trick. To each field we add information about which alphabet symbol is related to the array position. Moreover two nodes must never start at the same position of the array. We must add one bit of information if the position is a *base position* and when inserting, we never pack two nodes at the same base position.

Index	1	2	3	4	5	6	7	8	9
Character		a	b	c	d	a	b	b	a
Pointer			5		8		6		
Base position?	Y				Y	Y		Y	
End of word?						Y	Y	Y	Y

FIGURE 2: PACKED TRIE

In Figure 2 the same language as used previously is stored. The trie starts on position 1, this is the base position. The trie root is treated specially for implementation reasons, it is always stored fully in the array, even if there are no words starting with the appropriate character. Only the pointer is null in that case.

We assert numerical values to the alphabet symbols: $a = 1$, $b = 2$, $c = 3$, $d = 4$. How do we distinguish the fields belonging to a node? Let the node start at base position z. We go through positions $z + a, \ldots, z + d$ and check where the condition "the character on position $z+i$ is i" holds. For the root this is always true. In the root, there is a pointer under character b (on position 3). It points to the base position 5. Moreover the root says we have a word starting with d. Let us go through the positions belonging to base position 5, this means related to the prefix b. They are:

- ◇ position 6, this should be related to a, this holds, the pointer is null, the end-of-word flag is true, hence ba belongs to the language and any other word starting with ba does not.
- ◇ position 7, which is related to b, so the position belongs to the node, the position is end-of-word, therefore bb belongs to the language and there are words starting with bb continuing as said by the node on base position 6.
- ◇ positions 8 and 9 should belong to the characters c and d, this is not the case, these positions do not belong to the current node.

The reader may easily check that the table contains the same language as shown in Figure 1. Sixteen fields are needed to store the language naïvely, we need nine when packing. The ratio is not representative, it depends on language stored.

The trie nodes may be packed using the first-fit algorithm. This means when packing a node, we find the first position where it can be done, where we do not interfere with existing nodes and we do not use the same base position. We can speed up the process using the following heuristics. If the node we want to store is filled less than a threshold, we don't loose time finding an appropriate position but store it at the end of the array. Otherwise we use the first-fit method as described. Our experience shows that array usage much better than 95 % may be obtained without significant loss of speed.

Pattern Translation Processes

If all you have is a hammer, everything looks like a nail.
— popular aphorism

Let us review several tasks related to computer typesetting, in order to see whether they could be implemented as a Pattern Translation Processes (PTP), implemented using PatLib. Most of them are currently being tackled via *external* ΩTPs in Omega [7].

Hyphenation of compound words The plausibility of the approach has been shown for German in [13].

Context-dependent ligatures In addition to the already mentioned ligatures at the compound word boundaries, another example exists:

Fraktur long s versus short s In the Gothic letter-type there are two types of s-es, a long one and the normal one. The actual usage depends on the word morphology. Another typical context-dependent auto-tagging procedure implementable by PTP.

End of sentence recognition To typeset a different width space at the end of a sentence automatically, one has to filter out abbreviations that do not normally appear at the end of a sentence. A hard, but doable task for PTP.

Spell checking Storing a big word-list in a packed digital tree is feasible and gives results comparable to spelling checkers like ispell. For languages with inflection, however, several hierarchical PTP's are needed for better performance. We are afraid that PTP's cannot beat specialised fine-tuned morphological analysers, though.

Thai segmentation There is no explicit word/sentence boundary, punctuation and inflexion in Thai text. This information, implicitly tagged by spaces and punctuation marks in most languages, is missing in standard Thai text transliteration. It is, however, needed, during typesetting for line-breaking. It has yet to be shown that pattern-based technology is at least comparable to the currently used probabilistic trigram model [1].

Arabic letter hamza Typesetting systems for Arabic scripts need to have built-in logic for choosing one of five possible appearances of the letter hamza, depending on context. This process can easily be formulated as a PTP.

Greek accents In [7, page 153] there is an algorithm — full of exceptions and context dependent actions — for the process of adding proper accents in Greek texts. Most parts of it can easily be described as a sequence of pattern-triggered actions and thus be implemented as a PTP.

Similarly, there are many Czech texts written without diacritics from the times when email mailers only supported seven-bit ASCII , which wait to be converted into proper form. Even for this task PTP's could be trained.

We believe that PTP implementation based on PatLib could become common ground for most, if not all, ΩTP's. Hooking and piping various PTP's in Omega may lead to uniform, highly effective (all those mapping are *linear* with respect to the length of the text) document processing. Compared to external ΩTP's, PTP imple-

mentation would win in speed. To some extent, we think that a new version of PatGen based on PatLib will not only be independent of language (for hyphenation), but of application, too.

Summary and Future Work

Write once, use everywhere.
— paraphrase of a well known slogan

We have discussed the motivation for developing a new library for the handling and generation of patterns, and we presented its design and first version. We argue that the pattern-based techniques have a rich future in many application areas and hope for PatLib to be playing a rôle there.

Readers are invited to download the latest version of PatLib and the PatGen reimplementation at `http://www.fi.muni.cz/~xantos/patlib/`.

Acknowledgement
The authors thank a reviewer for detailed language revision.

References

[1] Orchid corpus. Technical Report TR-NECTEC-1997-001, Thai National Electronics and Computer Technology Center, 1999. `http://www.links.nectec.or.th/`.

[2] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, USA, 1989.

[3] Patrick Hanks, editor. *The New Oxford Dictionary of English*. Oxford University Press, Oxford, 1998.

[4] Yannis Haralambous. A Small Tutorial on the Multilingual Features of PATGEN2. in electronic form, available from CTAN as `info/patgen2.tutorial`, January 1994.

[5] Yannis Haralambous and John Plaice. First applications of Ω: Adobe Poetica, Arabic, Greek, Khmer. *TUGboat*, 15(3):344–352, September 1994.

[6] Yannis Haralambous and John Plaice. Methods for Processing Languages with Omega. In *Proceedings of the Second International Symposium on Multilingual Information Processing, Tsukuba, Japan*, 1997. available as `http://genepi.louis-jean.com/omega/tsukuba-methods97.pdf`.

[7] Yannis Haralambous and John Plaice. Traitement automatique des langues et composition sous Omega. *Cahiers Gutenberg*, (39–40):139–166, May 2001.

[8] Donald E. Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[9] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1998.

[10] Franklin M. Liang. *Word Hy-phen-a-tion by Com-put-er*. Ph.D. Thesis, Department of Computer Science, Stanford University, August 1983.

[11] Franklin M. Liang and Peter Breitenlohner. `PAT`tern `GEN`eration program for the TEX82 hyphenator. Electronic documentation of PATGEN program version 2.3 from web2c distribution on CTAN, 1999.

[12] John Plaice and Yannis Haralambous. The latest developments in OMEGA. *TUGboat*, 17(2):181–183, June 1996.

[13] Petr Sojka. Notes on Compound Word Hyphenation in TEX. *TUGboat*, 16(3):290–297, 1995.

[14] Petr Sojka. Hyphenation on Demand. *TUGboat*, 20(3):241–247, 1999.

[15] Petr Sojka and Pavel Ševeček. Hyphenation in TEX — Quo Vadis? *TUGboat*, 16(3):280–289, 1995.

[16] Alan Turing. Computing machinery and intelligence. *Mind*, (59):433–460, 1950.