PATtern GENeration program for the TeX82 hyphenator

(Version 2.4, April 2020)

1. Introduction. This program takes a list of hyphenated words and generates a set of patterns that can be used by the TEX82 hyphenation algorithm.

The patterns consist of strings of letters and digits, where a digit indicates a 'hyphenation value' for some intercharacter position. For example, the pattern `3t2ion` specifies that if the string `tion` occurs in a word, we should assign a hyphenation value of 3 to the position immediately before the `t`, and a value of 2 to the position between the `t` and the `i`.

To hyphenate a word, we find all patterns that match within the word and determine the hyphenation values for each intercharacter position. If more than one pattern applies to a given position, we take the maximum of the values specified (i.e., the higher value takes priority). If the resulting hyphenation value is odd, this position is a feasible breakpoint; if the value is even or if no value has been specified, we are not allowed to break at this position.

In order to find quickly the patterns that match in a given word and to compute the associated hyphenation values, the patterns generated by this program are compiled by `INITEX` into a compact version of a finite state machine. For further details, see the TEX82 source.

The *banner* string defined here should be changed whenever `PATGEN` gets modified.

define *banner* ≡ ´This␣is␣PATGEN,␣Version␣2.4´ { printed when the program starts }

2. The original version 1 of `PATGEN` was written by Frank M. Liang in 1982; a major revision (version 2) by Peter Breitenlohner in 1991 is mostly related to the new features of '8-bit TEX' (version 3 of TEX82). The differences between versions 1 and 2 fall into several categories (all of Liang's algorithms have been left essentially unchanged): (1) enhancements related to 8-bit TEX, e.g., the introduction of 8-bit *ASCII_code* values and of `\lefthyphenmin` and `\righthyphenmin`; (2) a modification of the input and output procedures which should make language specific modifications of this program unnecessary (information about the external representation of all 'letters' used by a particular language is obtained from the *translate* file); (3) removal of ANSI standard Pascal and range check violations; (4) removal of uninitialized variables; (5) minor modifications in order to simplify system-dependent modifications.

3. This program is written in standard Pascal, except where it is necessary to use extensions. All places where nonstandard constructions are used have been listed in the index under "system dependencies."

The program uses Pascal's standard *input* and *output* files to read from and write to the user's terminal.

```
define print(#) ≡ write(output, #)
define print_ln(#) ≡ write_ln(output, #)
define get_input(#) ≡ read(input, #)
define get_input_ln(#) ≡
        begin if eoln(input) then read_ln(input);
        read(input, #);
        end
define end_of_PATGEN = 9999

⟨ Compiler directives 11 ⟩
program PATGEN(dictionary, patterns, translate, patout);
  label end_of_PATGEN;
  const ⟨ Constants in the outer block 27 ⟩
  type ⟨ Types in the outer block 12 ⟩
  var ⟨ Globals in the outer block 4 ⟩
  procedure initialize; { this procedure gets things started properly }
    var ⟨ Local variables for initialization 15 ⟩
    begin print_ln(banner);
    ⟨ Set initial values 14 ⟩
    end;
```

4. The patterns are generated in a series of sequential passes through the dictionary. In each pass, we collect count statistics for a particular type of pattern, taking into account the effect of patterns chosen in previous passes. At the end of a pass, the counts are examined and new patterns are selected.

Patterns are chosen one level at a time, in order of increasing hyphenation value. In the sample run shown below, the parameters *hyph_start* and *hyph_finish* specify the first and last levels, respectively, to be generated.

Patterns at each level are chosen in order of increasing pattern length (usually starting with length 2). This is controlled by the parameters *pat_start* and *pat_finish* specified at the beginning of each level.

Furthermore patterns of the same length applying to different intercharacter positions are chosen in separate passes through the dictionary. Since patterns of length n may apply to $n+1$ different positions, choosing a set of patterns of lengths 2 through n for a given level requires $(n+1)(n+2)/2-3$ passes through the word list.

At each level, the selection of patterns is controlled by the three parameters *good_wt*, *bad_wt*, and *thresh*. A hyphenating pattern will be selected if $good * good\_wt - bad * bad\_wt \geq thresh$, where *good* and *bad* are the number of times the pattern could and could not be hyphenated, respectively, at a particular point. For inhibiting patterns, *good* is the number of errors inhibited, and *bad* is the number of previously found hyphens inhibited.

⟨ Globals in the outer block 4 ⟩ ≡
pat_start, *pat_finish*: *dot_type*;
hyph_start, *hyph_finish*: *val_type*;
good_wt, *bad_wt*, *thresh*: *integer*;

See also sections 16, 23, 25, 30, 31, 33, 40, 43, 51, 52, 55, 66, 74, 78, 84, 87, 91, and 95.

This code is used in section 3.

5. The proper choice of the parameters to achieve a desired degree of hyphenation is discussed in Chapter 4. Below we show part of a sample run of `PATGEN`, with the user's inputs underlined.

```
ex patgen
DICTIONARY : murray.hyf
PATTERNS : nul:
TRANSLATE : nul:
PATOUT : murray.pat
This is PATGEN, Version 2.0
left_hyphen_min = 2, right_hyphen_min = 3, 26 letters
0 patterns read in
pattern trie has 256 nodes, trie_max = 256, 0 outputs
hyph_start, hyph_finish: 1 1
pat_start, pat_finish: 2 3
good weight, bad weight, threshold: 1 3 3
processing dictionary with pat_len = 2, pat_dot = 1

0 good, 0 bad, 3265 missed
0.00 %, 0.00 %, 100.00 %
338 patterns, 466 nodes in count trie, triec_max = 983
46 good and 152 bad patterns added (more to come)
finding 715 good and 62 bad hyphens, efficiency = 10.72
pattern trie has 326 nodes, trie_max = 509, 2 outputs
processing dictionary with pat_len = 2, pat_dot = 0

   ...

1592 nodes and 39 outputs deleted
total of 220 patterns at hyph_level 1
hyphenate word list? y
writing pattmp.1

2529 good, 243 bad, 736 missed
77.46 %, 7.44 %, 22.54 %
```

6. Note that before beginning a pattern selection run, a file of existing patterns may be read in. In order for pattern selection to work properly, this file should only contain patterns with hyphenation values less than *hyph_start*. Each word in the dictionary is hyphenated according to the existing set of patterns (including those chosen on previous passes of the current run) before pattern statistics are collected.

Also, a hyphenated word list may be written out at the end of a run. This list can be read back in as the 'dictionary' to continue pattern selection from this point. In addition to ordinary hyphens (`"-"`) the new list will contain two additional kinds of "hyphens" between letters, namely hyphens that have been found by previously generated patterns, as well as erroneous hyphens that have been inserted by those patterns. These are represented by the symbols `"*"` and `"."`, respectively. The three characters `"-"`, `"*"`, and `"."` are, in fact, just the default values used to represent the three kinds of hyphens, the *translate* file may specify different characters to be used instead of them.

In addition, a word list can include hyphen weights, both for entire words and for individual hyphen positions. (The syntax for this is explained in the dictionary processing routines.) Thus common words can be weighted more heavily, or, more generally, words can be weighted according to their frequency of occurrence, if such information is available. The use of hyphen weights combined with an appropriate setting of the pattern selection threshold can be used to guarantee hyphenation of certain words or certain hyphen positions within a word.

7. Below we show the first few lines of a typical word list, before and after generating a level of patterns.

```
abil-i-ty        abil*i*ty
ab-sence         ab*sence
ab-stract        ab*stract
ac-a-dem-ic      ac-a-d.em-ic
ac-cept          ac*cept
ac-cept-able     ac*cept-able
ac-cept-ed       ac*cept*ed
   ...              ...
```

8. We augment Pascal's control structures a bit using **goto**'s and the following symbolic labels.

define *exit* = 10 { go here to leave a procedure }
define *continue* = 22 { go here to resume a loop }
define *done* = 30 { go here to exit a loop }
define *found* = 40 { go here when you've found it }
define *not_found* = 41 { go here when you've found something else }

9. Here are some macros for common programming idioms.

define *incr*(#) ≡ # ← # + 1 { increase a variable by unity }
define *decr*(#) ≡ # ← # − 1 { decrease a variable by unity }
define *Incr_Decr_end*(#) ≡ #
define *Incr*(#) ≡ # ← # + *Incr_Decr_end* { we use *Incr*(*a*)(*b*) to increase ... }
define *Decr*(#) ≡ # ← # − *Incr_Decr_end*
{ ... and *Decr*(*a*)(*b*) to decrease variable *a* by *b*; this can be optimized for some compilers }
define *loop* ≡ **while** *true* **do** { repeat over and over until a **goto** happens }
define *do_nothing* ≡ { empty statement }
define *return* ≡ **goto** *exit* { terminate a procedure call }
format *return* ≡ *nil*
format *loop* ≡ *xclause*

10. In case of serious problems `PATGEN` will give up, after issuing an error message about what caused the error. Such errors might be discovered inside of subroutines inside of subroutines, so a `WEB` macro called *jump_out* has been introduced. This macro, which transfers control to the label *end_of_PATGEN* at the end of the program, contains the only non-local **goto** statement in `PATGEN`. Some Pascal compilers do not implement non-local **goto** statements. In such cases the **goto** *end_of_PATGEN* in the definition of *jump_out* should simply be replaced by a call on some system procedure that quietly terminates the program.

An overflow stop occurs if `PATGEN`'s tables aren't large enough.

define *jump_out* ≡ **goto** *end_of_PATGEN* { terminates `PATGEN` }
define *error*(#) ≡
  begin *print_ln*(#); *jump_out*;
  end
define *overflow*(#) ≡ *error*(´PATGEN␣capacity␣exceeded,␣sorry␣[´, #, ´].´)

11. ⟨ Compiler directives 11 ⟩ ≡
@{@&$*C*−, *A*+, *D*−@} { no range check, catch arithmetic overflow, no debug overhead }

This code is used in section 3.

12. The character set. Since different Pascal systems may use different character sets, we use the name *text_char* to stand for the data type of characters appearing in external text files. We also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The definitions below should be adjusted if necessary.

Internally, characters will be represented using the type *ASCII_code*. Note, however, that only some of the standard ASCII characters are assigned a fixed *ASCII_code*; all other characters are assigned an *ASCII_code* dynamically when they are first read from the *translate* file specifying the external representation of the 'letters' used by a particular language. For the sake of generality the standard version of this program allows for 256 different *ASCII_code* values, but 128 of them would probably suffice for all practical purposes.

define *first_text_char* $= 0$ { ordinal number of the smallest element of *text_char* }
define *last_text_char* $= 255$ { ordinal number of the largest element of *text_char* }

define *last_ASCII_code* $= 255$ { the highest allowed *ASCII_code* value }

⟨ Types in the outer block 12 ⟩ ≡
text_char = *char*; { the data type of characters in text files }
ASCII_code = 0 .. *last_ASCII_code*; { internal representation of input characters }
text_file = *text*;

See also sections 13, 20, 22, and 29.

This code is used in section 3.

13. Some Pascals can store only signed eight-bit quantities ($-128\,..\,127$) but not unsigned ones ($0\,..\,255$) in one byte. If storage is tight we must, for such Pascals, either restrict *ASCII_code* to the range $0\,..\,127$ (with some loss of generality) or convert between *ASCII_code* and *packed_ASCII_code* and vice versa by subtracting or adding an offset. (Or we might define *packed_ASCII_code* as *char* and use suitable typecasts for the conversion.) Only the type *packed_ASCII_code* will be used for large arrays and the `WEB` macros *si* and *so* will always be used to convert an *ASCII_code* into a *packed_ASCII_code* and vice versa.

define *min_packed* $= 0$ { change this to '*min_packed* $= -128$' when necessary; and don't forget to change the definitions of *si* and *so* below accordingly }

define *si*(#) ≡ # { converts *ASCII_code* to *packed_ASCII_code* }
define *so*(#) ≡ # { converts *packed_ASCII_code* to *ASCII_code* }

⟨ Types in the outer block 12 ⟩ +≡
packed_ASCII_code = *min_packed* .. *last_ASCII_code* + *min_packed*;

14. We want to make sure that the "constants" defined in this program satisfy all the required relations. Some of them are needed to avoid time-consuming checks while processing the dictionary and/or to prevent range check and array bound violations.

Here we check that the definitions of *ASCII_code* and *packed_ASCII_code* are consistent with those of *si* and *so*.

⟨ Set initial values 14 ⟩ ≡
bad ← 0;
if *last_ASCII_code* < 127 **then** *bad* ← 1;
if $(si(0) \neq min\_packed) \lor (so(min\_packed) \neq 0)$ **then** *bad* ← 2;
⟨ Check the "constant" values for consistency 28 ⟩
if *bad* > 0 **then** *error*(´`Bad␣constants---case␣`´, *bad* : 1);

See also sections 17, 18, and 24.

This code is used in section 3.

15. $\langle$ Local variables for initialization 15 $\rangle \equiv$
bad: *integer*;
i: *text_char*;
j: *ASCII_code*;

This code is used in section 3.

16. We convert between *ASCII_code* and the user's external character set by means of arrays *xord* and *xchr* that are analogous to Pascal's *ord* and *chr* functions.

$\langle$ Globals in the outer block 4 $\rangle$ +$\equiv$
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [*ASCII_code*] **of** *text_char*; { specifies conversion of output characters }

17. The following code initializes the *xchr* array with some of the standard ASCII characters.

$\langle$ Set initial values 14 $\rangle$ +$\equiv$
for $j \leftarrow 0$ **to** *last_ASCII_code* **do** *xchr*[*j*] ← ´␣´;
xchr["."] ← ´.´;
xchr["0"] ← ´0´; *xchr*["1"] ← ´1´; *xchr*["2"] ← ´2´; *xchr*["3"] ← ´3´; *xchr*["4"] ← ´4´;
xchr["5"] ← ´5´; *xchr*["6"] ← ´6´; *xchr*["7"] ← ´7´; *xchr*["8"] ← ´8´; *xchr*["9"] ← ´9´;
xchr["A"] ← ´A´; *xchr*["B"] ← ´B´; *xchr*["C"] ← ´C´; *xchr*["D"] ← ´D´; *xchr*["E"] ← ´E´;
xchr["F"] ← ´F´; *xchr*["G"] ← ´G´; *xchr*["H"] ← ´H´; *xchr*["I"] ← ´I´; *xchr*["J"] ← ´J´;
xchr["K"] ← ´K´; *xchr*["L"] ← ´L´; *xchr*["M"] ← ´M´; *xchr*["N"] ← ´N´; *xchr*["O"] ← ´O´;
xchr["P"] ← ´P´; *xchr*["Q"] ← ´Q´; *xchr*["R"] ← ´R´; *xchr*["S"] ← ´S´; *xchr*["T"] ← ´T´;
xchr["U"] ← ´U´; *xchr*["V"] ← ´V´; *xchr*["W"] ← ´W´; *xchr*["X"] ← ´X´; *xchr*["Y"] ← ´Y´;
xchr["Z"] ← ´Z´;
xchr["a"] ← ´a´; *xchr*["b"] ← ´b´; *xchr*["c"] ← ´c´; *xchr*["d"] ← ´d´; *xchr*["e"] ← ´e´;
xchr["f"] ← ´f´; *xchr*["g"] ← ´g´; *xchr*["h"] ← ´h´; *xchr*["i"] ← ´i´; *xchr*["j"] ← ´j´;
xchr["k"] ← ´k´; *xchr*["l"] ← ´l´; *xchr*["m"] ← ´m´; *xchr*["n"] ← ´n´; *xchr*["o"] ← ´o´;
xchr["p"] ← ´p´; *xchr*["q"] ← ´q´; *xchr*["r"] ← ´r´; *xchr*["s"] ← ´s´; *xchr*["t"] ← ´t´;
xchr["u"] ← ´u´; *xchr*["v"] ← ´v´; *xchr*["w"] ← ´w´; *xchr*["x"] ← ´x´; *xchr*["y"] ← ´y´;
xchr["z"] ← ´z´;

18. The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*.

define *invalid_code* = 0 { *ASCII_code* that should not appear }
define *tab_char* = ´11
{ *ord* of tab character; tab characters seem to be unavoidable with files from UNIX systems }

$\langle$ Set initial values 14 $\rangle$ +$\equiv$
for *i* ← *chr*(*first_text_char*) **to** *chr*(*last_text_char*) **do** *xord*[*i*] ← *invalid_code*;
for $j \leftarrow 0$ **to** *last_ASCII_code* **do** *xord*[*xchr*[*j*]] ← *j*;
xord[´␣´] ← "␣"; *xord*[*chr*(*tab_char*)] ← "␣";

19. So far each invalid *ASCII_code* has been assigned the character `' '` and all invalid characters have been assigned *ASCII_code* = *invalid_code*. The *get_ASCII* function, used only while reading the *translate* file, returns the *ASCII_code* corresponding to a character, assigning a new *ASCII_code* first if necessary.

define *num_ASCII_codes* = *last_ASCII_code* + 1 { number of different *ASCII_code* values }

```
function get_ASCII(c : text_char): ASCII_code;
  label found;
  var i: ASCII_code;
  begin i ← xord[c];
  if i = invalid_code then
    begin while i < last_ASCII_code do
      begin incr(i);
      if (xchr[i] = ' ') ∧ (i ≠ " ") then goto found;
      end;
    overflow(num_ASCII_codes : 1, ' characters');
  found: xord[c] ← i; xchr[i] ← c;
    end;
  get_ASCII ← i;
  end;
```

20. The TEX82 hyphenation algorithm operates on 'hyphenable words' converted temporarily to lower case, i.e., they may consist of up to 255 different 'letters' corresponding to `\lccode`s 1 .. 255. These `\lccode`s could, in principle, be language dependent but this might lead to undesirable results when hyphenating multilingual paragraphs. No more than 245 different letters can occur in hyphenation patterns since the characters `'0'` .. `'9'` and `'.'` play a special rôle when reading patterns. For the purpose of this program each letter is represented internally by a unique *internal_code* ≥ 2 (*internal_code* = 1 is the *edge_of_word* indicator); *internal_code* values 2 .. 127 will probably suffice for all practical purposes, but we allow the range 2 .. *last_ASCII_code* for the sake of generality. Syntactically *internal_code* and *ASCII_code* are the same, we will use one or the other name according to the semantic context.

define *edge_of_word* = 1 { *internal_code* for start and end of a word }

⟨ Types in the outer block 12 ⟩ +≡
internal_code = *ASCII_code*; *packed_internal_code* = *packed_ASCII_code*;

21. Note that an *internal_code* used by this program is in general quite different from the *ASCII_code* (or rather `\lccode`) used by TEX82. This program allows the input of characters (from the *dictionary* and *patterns* file) corresponding to an *internal_code* in either lower or upper case form; the output (to the *patout* and *pattmp* file) will always be in lower case form.

Unfortunately there does not (yet?) exist a standardized and widely accepted 8-bit character set (or a unique one-to-one translation between such sets). On the other hand macro expansion takes place in TEX82 when reading hyphenable words and when reading patterns. Thus the lower and upper case versions of all 'letters' used by a particular language can (and for the sake of portability should) be represented entirely in terms of the standard ASCII character set; either directly as characters or via macros (or active characters) with or without arguments. The macro definitions for such a representation will in general be language dependent.

For the purpose of this program the external representation of the lower and upper case version of a letter (i.e., *internal_code*) consists of a unique sequence of characters (or *ASCII_codes*), the only restriction being that no such sequence must be a subsequence of an other one. Moreover such sequences must not start with `"␣"`, `"."`, `"0"` .. `"9"` or with one of the three characters (`"-"`, `"*"`, and `"."`) representing hyphens in the *dictionary* file; a sequence may, however, end with a mandatory `"␣"` as, e.g., the sequence `"\ss␣"`.

The language dependent values of `\lefthyphenmin` and `\righthyphenmin` as well as the external representation of the lower and upper case letters and their collating sequence are specified in the *translate* file, thus making any language dependent modifications of this program unnecessary. If the *translate* file is empty (or does not exist) the values `\lefthyphenmin=2` and `\righthyphenmin=3` and *internal_code* values 2 .. 27 with the one character external representations `"a"` .. `"z"` and `"A"` .. `"Z"` will be used as defaults.

Incidentally this program can be used to convert a *dictionary* and *patterns* file from one ("upper case") to another ("lower case") external representation of letters.

22. When reading the *dictionary* (and *patterns*) file sequences of characters must be recognized and converted to their corresponding *internal_code*. This conversion is part of `PATGEN`s inner loop and must therefore be done as efficient as possible. Thus we will mostly bypass the conversion from character to *ASCII_code* and convert directly to the corresponding *internal_code* using the *xclass* and *xint* arrays. Six types of characters are distinguished by their *xclass*:

space_class character ´␣´ terminates a pattern or word.

digit_class characters ´0´ . . ´9´ are hyphen values for a pattern or hyphen weights for a word; their *xint* is the corresponding numeric value 0 . . 9.

hyf_class characters (´.´, ´-´, and ´*´) are 'dots' and indicate hyphens in a word; their *xint* is the corresponding numeric value *err_hyf* . . *found_hyf*.

letter_class characters represent a letter; their *xint* is the corresponding *internal_code*.

escape_class characters indicate the start of a multi-character sequence representing a letter.

invalid_class characters should not occur except as part of multi-character sequences.

define *space_class* = 0 { the character ´␣´ }
define *digit_class* = 1 { the characters ´0´ . . ´9´ }
define *hyf_class* = 2 { the 'hyphen' characters (´.´, ´-´, and ´*´) }
define *letter_class* = 3 { characters representing a letter }
define *escape_class* = 4 { characters that start a multi-character sequence representing a letter }
define *invalid_class* = 5 { characters that normally should not occur }

define *no_hyf* = 0 { no hyphen }
define *err_hyf* = 1 { erroneous hyphen }
define *is_hyf* = 2 { hyphen }
define *found_hyf* = 3 { found hyphen }

⟨ Types in the outer block 12 ⟩ +≡
class_type = *space_class* . . *invalid_class*; { class of a character }
digit = 0 . . 9; { a hyphen weight (or word weight) }
hyf_type = *no_hyf* . . *found_hyf*; { type of a hyphen }

23. In addition we will use the *xext*, *xdig*, and *xdot* arrays to convert from the internal representation to the corresponding characters.

⟨ Globals in the outer block 4 ⟩ +≡
xclass: **array** [*text_char*] **of** *class_type*; { specifies the class of a character }
xint: **array** [*text_char*] **of** *internal_code*; { specifies the *internal_code* for a character }
xdig: **array** [0 . . 9] **of** *text_char*; { specifies conversion of output characters }
xext: **array** [*internal_code*] **of** *text_char*; { specifies conversion of output characters }
xhyf: **array** [*err_hyf* . . *found_hyf*] **of** *text_char*; { specifies conversion of output characters }

24. ⟨ Set initial values 14 ⟩ +≡
for *i* ← *chr*(*first_text_char*) **to** *chr*(*last_text_char*) **do**
  begin *xclass*[*i*] ← *invalid_class*; *xint*[*i*] ← 0;
  end;
xclass[´␣´] ← *space_class*;
for *j* ← 0 **to** *last_ASCII_code* **do** *xext*[*j*] ← ´␣´;
xext[*edge_of_word*] ← ´.´;
for *j* ← 0 **to** 9 **do**
  begin *xdig*[*j*] ← *xchr*[*j* + "0"]; *xclass*[*xdig*[*j*]] ← *digit_class*; *xint*[*xdig*[*j*]] ← *j*;
  end;
xhyf[*err_hyf*] ← ´.´; *xhyf*[*is_hyf*] ← ´-´; *xhyf*[*found_hyf*] ← ´*´;
  { default representation for hyphens }

25. We assume that words use only the letters $cmin + 1$ through $cmax$. This allows us to save some time on trie operations that involve searching for packed transitions belonging to a particular state.

define $cmin = edge\_of\_word$

⟨ Globals in the outer block 4 ⟩ +≡
cmax: *internal_code*; { largest *internal_code* or *ASCII_code* }

26. Data structures. The main data structure used in this program is a dynamic packed trie. In fact we use two of them, one for the set of patterns selected so far, and one for the patterns being considered in the current pass.

For a pattern $p_1 \ldots p_k$, the information associated with that pattern is accessed by setting $t_1 \leftarrow \mathit{trie\_root} + p_1$ and then, for $1 < i \le k$, setting $t_i \leftarrow \mathit{trie\_link}(t_{i-1}) + p_i$; the pattern information is then stored in a location addressed by t_k. Since all trie nodes are packed into a single array, in order to distinguish nodes belonging to different trie families, a special field is provided such that $\mathit{trie\_char}(t_i) = \mathit{si}(p_i)$ for all i.

In addition the trie must support dynamic insertions and deletions. This is done by maintaining a doubly linked list of unoccupied cells and repacking trie families as necessary when insertions are made.

Each trie node consists of three fields: the character *trie_char*, and the two link fields *trie_link* and *trie_back*. In addition there is a separate boolean array *trie_base_used*. When a node is unoccupied, $\mathit{trie\_char} = \mathit{min\_packed}$ and the link fields point to the next and previous unoccupied nodes, respectively, in the doubly linked list. When a node is occupied, *trie_link* points to the next trie family, and *trie_back* (renamed *trie_outp*) contains the output associated with this transition. The *trie_base_used* bit indicates that some family has been packed at this base location, and is used to prevent two families from being packed at the same location.

27. The sizes of the pattern tries may have to be adjusted depending on the particular application (i.e., the parameter settings and the size of the dictionary). The sizes below were sufficient to generate the original set of English TEX82 hyphenation patterns (file `hyphen.tex`).

⟨ Constants in the outer block 27 ⟩ ≡
  $\mathit{trie\_size} = 55000$; { space for pattern trie }
  $\mathit{triec\_size} = 26000$; { space for pattern count trie, must be less than *trie_size* and greater than the number of occurrences of any pattern in the dictionary }
  $\mathit{max\_ops} = 4080$; { size of output hash table, should be a multiple of 510 }
  $\mathit{max\_val} = 10$; { maximum number of levels+1, also used to denote bad patterns }
  $\mathit{max\_dot} = 15$; { maximum pattern length, also maximum length of external representation of a 'letter' }
  $\mathit{max\_len} = 50$; { maximum word length }
  $\mathit{max\_buf\_len} = 80$; { maximum length of input lines, must be at least *max_len* }

This code is used in section 3.

28. ⟨ Check the "constant" values for consistency 28 ⟩ ≡
  if $(\mathit{triec\_size} < 4096) \lor (\mathit{trie\_size} < \mathit{triec\_size})$ **then** $\mathit{bad} \leftarrow 3$;
  if $\mathit{max\_ops} > \mathit{trie\_size}$ **then** $\mathit{bad} \leftarrow 4$;
  if $\mathit{max\_val} > 10$ **then** $\mathit{bad} \leftarrow 5$;
  if $\mathit{max\_buf\_len} < \mathit{max\_len}$ **then** $\mathit{bad} \leftarrow 6$;

This code is used in section 14.

29. ⟨ Types in the outer block 12 ⟩ +≡
  $\mathit{q\_index} = 1 \mathinner{..} \mathit{last\_ASCII\_code}$; { number of transitions in a state }
  $\mathit{val\_type} = 0 \mathinner{..} \mathit{max\_val}$; { hyphenation values }
  $\mathit{dot\_type} = 0 \mathinner{..} \mathit{max\_dot}$; { dot positions }
  $\mathit{op\_type} = 0 \mathinner{..} \mathit{max\_ops}$; { index into output hash table }
  $\mathit{word\_index} = 0 \mathinner{..} \mathit{max\_len}$; { index into *word* }
  $\mathit{trie\_pointer} = 0 \mathinner{..} \mathit{trie\_size}$; $\mathit{triec\_pointer} = 0 \mathinner{..} \mathit{triec\_size}$;
  op_word = **packed record** *dot*: *dot_type*;
    val: *val_type*;
    op: *op_type*
    end;

30. Trie is actually stored with its components in separate packed arrays, in order to save space and time (although this depends on the computer's word size and the size of the trie pointers).

⟨ Globals in the outer block 4 ⟩ +≡
trie_c: **packed array** [*trie_pointer*] **of** *packed_internal_code*;
trie_l, *trie_r*: **packed array** [*trie_pointer*] **of** *trie_pointer*;
trie_taken: **packed array** [*trie_pointer*] **of** *boolean*;
triec_c: **packed array** [*triec_pointer*] **of** *packed_internal_code*;
triec_l, *triec_r*: **packed array** [*triec_pointer*] **of** *triec_pointer*;
triec_taken: **packed array** [*triec_pointer*] **of** *boolean*;
ops: **array** [*op_type*] **of** *op_word*; { output hash table }

31. When some trie state is being worked on, an unpacked version of the state is kept in positions 1 .. *qmax* of the global arrays *trieq_c*, *trieq_l*, and *trieq_r*. The character fields need not be in any particular order.

⟨ Globals in the outer block 4 ⟩ +≡
trieq_c: **array** [*q_index*] **of** *internal_code*; { character fields of a single trie state }
trieq_l, *trieq_r*: **array** [*q_index*] **of** *trie_pointer*; { link fields }
qmax: *q_index*; { number of transitions in an unpacked state }
qmax_thresh: *q_index*; { controls density of first-fit packing }

32. Trie fields are accessed using the following macros.

define *trie_char*(#) ≡ *trie_c*[#]
define *trie_link*(#) ≡ *trie_l*[#]
define *trie_back*(#) ≡ *trie_r*[#]
define *trie_outp*(#) ≡ *trie_r*[#]
define *trie_base_used*(#) ≡ *trie_taken*[#]

define *triec_char*(#) ≡ *triec_c*[#]
define *triec_link*(#) ≡ *triec_l*[#]
define *triec_back*(#) ≡ *triec_r*[#]
define *triec_good*(#) ≡ *triec_l*[#]
define *triec_bad*(#) ≡ *triec_r*[#]
define *triec_base_used*(#) ≡ *triec_taken*[#]

define *q_char*(#) ≡ *trieq_c*[#]
define *q_link*(#) ≡ *trieq_l*[#]
define *q_back*(#) ≡ *trieq_r*[#]
define *q_outp*(#) ≡ *trieq_r*[#]

define *hyf_val*(#) ≡ *ops*[#].*val*
define *hyf_dot*(#) ≡ *ops*[#].*dot*
define *hyf_nxt*(#) ≡ *ops*[#].*op*

33. Routines for pattern trie. The pattern trie holds the set of patterns chosen prior to the current pass, including bad or "hopeless" patterns at the current level that occur too few times in the dictionary to be of use. Each transition of the trie includes an output field pointing to the hyphenation information associated with this transition.

⟨ Globals in the outer block 4 ⟩ +≡
trie_max: *trie_pointer*; { maximum occupied trie node }
trie_bmax: *trie_pointer*; { maximum base of trie family }
trie_count: *trie_pointer*; { number of occupied trie nodes, for space usage statistics }
op_count: *op_type*; { number of outputs in hash table }

34. Initially, the dynamic packed trie has just one state, namely the root, with all transitions present (but with null links). This is convenient because the root will never need to be repacked and also we won't have to check that the base is nonnegative when packing other states. Moreover in many cases we need not check for a vanishing link field: if $trie\_link(t) = 0$ then a subsequent test for $trie\_char(trie\_link(t) + c) = si(c)$ will always fail due to $trie\_root = 1$.

define $trie\_root = 1$

```
procedure init_pattern_trie;
  var c: internal_code; h: op_type;
  begin for c ← 0 to last_ASCII_code do
    begin trie_char(trie_root + c) ← si(c); { indicates node occupied; fake for c = 0 }
    trie_link(trie_root + c) ← 0; trie_outp(trie_root + c) ← 0; trie_base_used(trie_root + c) ← false;
    end;
  trie_base_used(trie_root) ← true; trie_bmax ← trie_root; trie_max ← trie_root + last_ASCII_code;
  trie_count ← num_ASCII_codes;
  qmax_thresh ← 5;
  trie_link(0) ← trie_max + 1; trie_back(trie_max + 1) ← 0;
    { trie_link(0) is used as the head of the doubly linked list of unoccupied cells }
  for h ← 1 to max_ops do hyf_val(h) ← 0; { clear output hash table }
  op_count ← 0;
  end;
```

35. The *first_fit* procedure finds a hole in the packed trie into which the state in *trieq_c*, *trieq_l*, and *trieq_r* will fit. This is normally done by going through the linked list of unoccupied cells and testing if the state will fit at each position. However if a state has too many transitions (and is therefore unlikely to fit among existing transitions) we don't bother and instead just pack it immediately to the right of the occupied region (starting at $trie\_max + 1$).

```
function first_fit: trie_pointer;
  label found, not_found;
  var s, t: trie_pointer; q: q_index;
  begin ⟨ Set s to the trie base location at which this state should be packed 36 ⟩;
  for q ← 1 to qmax do { pack it }
    begin t ← s + q_char(q);
    trie_link(trie_back(t)) ← trie_link(t); trie_back(trie_link(t)) ← trie_back(t); { link around filled cell }
    trie_char(t) ← si(q_char(q)); trie_link(t) ← q_link(q); trie_outp(t) ← q_outp(q);
    if t > trie_max then trie_max ← t;
    end;
  trie_base_used(s) ← true; first_fit ← s
  end;
```

36. The threshold for large states is initially 5 transitions. If more than one level of patterns is being generated, the threshold is set to 7 on subsequent levels because the pattern trie will be sparser after bad patterns are deleted (see *delete_bad_patterns*).

```
⟨Set s to the trie base location at which this state should be packed 36⟩ ≡
  if qmax > qmax_thresh then t ← trie_back(trie_max + 1) else t ← 0;
  loop
    begin t ← trie_link(t); s ← t − q_char(1);   { get next unoccupied cell }
    ⟨Ensure trie linked up to s + num_ASCII_codes 37⟩;
    if trie_base_used(s) then goto not_found;
    for q ← qmax downto 2 do   { check if state fits here }
      if trie_char(s + q_char(q)) ≠ min_packed then goto not_found;
    goto found;
  not_found: end;
found:
```

This code is used in section 35.

37. The trie is only initialized (as a doubly linked list of empty cells) as far as necessary. Here we extend the initialization if necessary, and check for overflow.

```
⟨Ensure trie linked up to s + num_ASCII_codes 37⟩ ≡
  if s > trie_size − num_ASCII_codes then overflow(trie_size : 1, ´␣pattern␣trie␣nodes´);
  while trie_bmax < s do
    begin incr(trie_bmax); trie_base_used(trie_bmax) ← false;
    trie_char(trie_bmax + last_ASCII_code) ← min_packed;
    trie_link(trie_bmax + last_ASCII_code) ← trie_bmax + num_ASCII_codes;
    trie_back(trie_bmax + num_ASCII_codes) ← trie_bmax + last_ASCII_code;
    end
```

This code is used in section 36.

38. The *unpack* procedure finds all transitions associated with the state with base s, puts them into the arrays *trieq_c*, *trieq_l*, and *trieq_r*, and sets *qmax* to one more than the number of transitions found. Freed cells are put at the beginning of the free list.

```
procedure unpack(s : trie_pointer);
  var c: internal_code; t: trie_pointer;
  begin qmax ← 1;
  for c ← cmin to cmax do   { search for transitions belonging to this state }
    begin t ← s + c;
    if so(trie_char(t)) = c then   { found one }
      begin q_char(qmax) ← c; q_link(qmax) ← trie_link(t); q_outp(qmax) ← trie_outp(t); incr(qmax);
        { now free trie node }
      trie_back(trie_link(0)) ← t; trie_link(t) ← trie_link(0); trie_link(0) ← t; trie_back(t) ← 0;
      trie_char(t) ← min_packed;
      end;
    end;
  trie_base_used(s) ← false;
  end;
```

39. The function *new_trie_op* returns the 'opcode' for the output consisting of hyphenation value v, hyphen position d, and next output n. The hash function used by *new_trie_op* is based on the idea that 313/510 is an approximation to the golden ratio [cf. *The Art of Computer Programming* **3** (1973), 510–512]; but the choice is comparatively unimportant in this particular application.

```
function new_trie_op(v : val_type; d : dot_type; n : op_type): op_type;
  label exit;
  var h: op_type;
  begin h ← ((n + 313 * d + 361 * v) mod max_ops) + 1;   { trial hash location }
  loop
    begin if hyf_val(h) = 0 then   { empty position found }
      begin incr(op_count);
      if op_count = max_ops then overflow(max_ops : 1, ´␣outputs´);
      hyf_val(h) ← v; hyf_dot(h) ← d; hyf_nxt(h) ← n; new_trie_op ← h; return;
      end;
    if (hyf_val(h) = v) ∧ (hyf_dot(h) = d) ∧ (hyf_nxt(h) = n) then   { already in hash table }
      begin new_trie_op ← h; return;
      end;
    if h > 1 then decr(h) else h ← max_ops;   { try again }
    end;
exit: end;
```

40. ⟨ Globals in the outer block 4 ⟩ +≡

```
pat: array [dot_type] of internal_code;   { current pattern }
pat_len: dot_type;   { pattern length }
```

41. Now that we have provided the necessary routines for manipulating the dynamic packed trie, here is a procedure that inserts a pattern of length *pat_len*, stored in the *pat* array, into the pattern trie. It also adds a new output.

```
procedure insert_pattern(val : val_type; dot : dot_type);
  var i: dot_type; s, t: trie_pointer;
  begin i ← 1; s ← trie_root + pat[i]; t ← trie_link(s);
  while (t > 0) ∧ (i < pat_len) do   { follow existing trie }
    begin incr(i); Incr(t)(pat[i]);
    if so(trie_char(t)) ≠ pat[i] then ⟨ Insert critical transition, possibly repacking 42 ⟩;
    s ← t; t ← trie_link(s);
    end;
  q_link(1) ← 0; q_outp(1) ← 0; qmax ← 1;
  while i < pat_len do   { insert rest of pattern }
    begin incr(i); q_char(1) ← pat[i]; t ← first_fit; trie_link(s) ← t; s ← t + pat[i]; incr(trie_count);
    end;
  trie_outp(s) ← new_trie_op(val, dot, trie_outp(s));
  end;
```

42. We have accessed a transition not in the trie. We insert it, repacking the state if necessary.

⟨ Insert critical transition, possibly repacking 42 ⟩ ≡
begin if $trie\_char(t) = min\_packed$ **then**
  begin { we're lucky, no repacking needed }
  $trie\_link(trie\_back(t)) \gets trie\_link(t)$; $trie\_back(trie\_link(t)) \gets trie\_back(t)$;
  $trie\_char(t) \gets si(pat[i])$; $trie\_link(t) \gets 0$; $trie\_outp(t) \gets 0$;
  if $t > trie\_max$ **then** $trie\_max \gets t$;
  end
else begin { whoops, have to repack }
  $unpack(t - pat[i])$;
  $q\_char(qmax) \gets pat[i]$; $q\_link(qmax) \gets 0$; $q\_outp(qmax) \gets 0$;
  $t \gets first\_fit$; $trie\_link(s) \gets t$; $Incr(t)(pat[i])$;
  end;
$incr(trie\_count)$;
end

This code is used in sections 41 and 59.

43. Routines for pattern count trie. The pattern count trie is used to store the set of patterns considered in the current pass, along with the counts of good and bad instances. The fields of this trie are the same as the pattern trie, except that there is no output field, and leaf nodes are also used to store counts (*triec_good* and *triec_bad*). Except where noted, the following routines are analogous to the pattern trie routines.

⟨ Globals in the outer block 4 ⟩ +≡
triec_max, *triec_bmax*, *triec_count*: *triec_pointer*; { same as for pattern trie }
triec_kmax: *triec_pointer*; { shows growth of trie during pass }
pat_count: *integer*; { number of patterns in count trie }

44. [See *init_pattern_trie*.] The variable *triec_kmax* always contains the size of the count trie rounded up to the next multiple of 4096, and is used to show the growth of the trie during each pass.

define *triec_root* = 1

procedure *init_count_trie*;
  var *c*: *internal_code*;
  begin for *c* ← 0 **to** *last_ASCII_code* **do**
    begin *triec_char*(*triec_root* + *c*) ← *si*(*c*);
    triec_link(*triec_root* + *c*) ← 0; *triec_back*(*triec_root* + *c*) ← 0; *triec_base_used*(*triec_root* + *c*) ← *false*;
    end;
  triec_base_used(*triec_root*) ← *true*; *triec_bmax* ← *triec_root*; *triec_max* ← *triec_root* + *last_ASCII_code*;
  triec_count ← *num_ASCII_codes*; *triec_kmax* ← 4096;
  triec_link(0) ← *triec_max* + 1; *triec_back*(*triec_max* + 1) ← 0;
  pat_count ← 0;
  end;

45. [See *first_fit*.]

function *firstc_fit*: *triec_pointer*;
  label *found*, *not_found*;
  var *a*, *b*: *triec_pointer*; *q*: *q_index*;
  begin ⟨ Set *b* to the count trie base location at which this state should be packed 46 ⟩;
  for *q* ← 1 **to** *qmax* **do** { pack it }
    begin *a* ← *b* + *q_char*(*q*);
    triec_link(*triec_back*(*a*)) ← *triec_link*(*a*); *triec_back*(*triec_link*(*a*)) ← *triec_back*(*a*);
    triec_char(*a*) ← *si*(*q_char*(*q*)); *triec_link*(*a*) ← *q_link*(*q*); *triec_back*(*a*) ← *q_back*(*q*);
    if *a* > *triec_max* **then** *triec_max* ← *a*;
    end;
  triec_base_used(*b*) ← *true*; *firstc_fit* ← *b*
  end;

46. The threshold for attempting a first-fit packing is 3 transitions, which is lower than for the pattern trie because speed is more important here.

⟨ Set b to the count trie base location at which this state should be packed 46 ⟩ ≡

```
if qmax > 3 then a ← triec_back(triec_max + 1) else a ← 0;
loop
  begin a ← triec_link(a); b ← a − q_char(1);
  ⟨ Ensure triec linked up to b + num_ASCII_codes 47 ⟩;
  if triec_base_used(b) then goto not_found;
  for q ← qmax downto 2 do
    if triec_char(b + q_char(q)) ≠ min_packed then goto not_found;
  goto found;
not_found: end;
found:
```

This code is used in section 45.

47. ⟨ Ensure *triec* linked up to $b + num\_ASCII\_codes$ 47 ⟩ ≡

```
if b > triec_kmax − num_ASCII_codes then
  begin if triec_kmax = triec_size then overflow(triec_size : 1, ´␣count␣trie␣nodes´);
  print(triec_kmax div 1024 : 1, ´K␣´);
  if triec_kmax > triec_size − 4096 then triec_kmax ← triec_size
  else Incr(triec_kmax)(4096);
  end;
while triec_bmax < b do
  begin incr(triec_bmax); triec_base_used(triec_bmax) ← false;
  triec_char(triec_bmax + last_ASCII_code) ← min_packed;
  triec_link(triec_bmax + last_ASCII_code) ← triec_bmax + num_ASCII_codes;
  triec_back(triec_bmax + num_ASCII_codes) ← triec_bmax + last_ASCII_code;
  end
```

This code is used in section 46.

48. [See *unpack*.]

```
procedure unpackc(b : triec_pointer);
  var c: internal_code; a: triec_pointer;
  begin qmax ← 1;
  for c ← cmin to cmax do   { search for transitions belonging to this state }
    begin a ← b + c;
    if so(triec_char(a)) = c then   { found one }
      begin q_char(qmax) ← c; q_link(qmax) ← triec_link(a); q_back(qmax) ← triec_back(a);
      incr(qmax);
      triec_back(triec_link(0)) ← a; triec_link(a) ← triec_link(0); triec_link(0) ← a; triec_back(a) ← 0;
      triec_char(a) ← min_packed;
      end;
    end;
  triec_base_used(b) ← false;
  end;
```

49. [See *insert_pattern*.] Patterns being inserted into the count trie are always substrings of the current word, so they are contained in the array *word* with length *pat_len* and finishing position *fpos*.

```
function insertc_pat(fpos : word_index): triec_pointer;
  var spos: word_index; a, b: triec_pointer;
  begin spos ← fpos − pat_len;   { starting position of pattern }
  incr(spos); b ← triec_root + word[spos]; a ← triec_link(b);
  while (a > 0) ∧ (spos < fpos) do   { follow existing trie }
    begin incr(spos); Incr(a)(word[spos]);
    if so(triec_char(a)) ≠ word[spos] then ⟨Insert critical count transition, possibly repacking 50⟩;
    b ← a; a ← triec_link(a);
    end;
  q_link(1) ← 0; q_back(1) ← 0; qmax ← 1;
  while spos < fpos do   { insert rest of pattern }
    begin incr(spos); q_char(1) ← word[spos]; a ← firstc_fit; triec_link(b) ← a; b ← a + word[spos];
    incr(triec_count);
    end;
  insertc_pat ← b; incr(pat_count);
  end;
```

50. ⟨Insert critical count transition, possibly repacking 50⟩ ≡

```
  begin if triec_char(a) = min_packed then   { lucky }
    begin triec_link(triec_back(a)) ← triec_link(a); triec_back(triec_link(a)) ← triec_back(a);
    triec_char(a) ← si(word[spos]); triec_link(a) ← 0; triec_back(a) ← 0;
    if a > triec_max then triec_max ← a;
    end
  else begin   { have to repack }
    unpackc(a − word[spos]);
    q_char(qmax) ← word[spos]; q_link(qmax) ← 0; q_back(qmax) ← 0; a ← firstc_fit; triec_link(b) ← a;
    Incr(a)(word[spos]);
    end;
  incr(triec_count);
  end
```

This code is used in section 49.

51. Input and output. For some Pascal systems output files must be closed before the program terminates; it may also be necessary to close input files. Since standard Pascal does not provide for this, we use WEB macros and will say *close_out*(*f*) resp. *close_in*(*f*); these macros should not produce errors or system messages, even if a file could not be opened successfully.

```
define close_out(#) ≡ close(#)   { close an output file }
define close_in(#) ≡ do_nothing   { close an input file }
```

⟨ Globals in the outer block 4 ⟩ +≡
dictionary, *patterns*, *translate*, *patout*, *pattmp*: *text_file*;

52. When reading a line from one of the input files (*dictionary*, *patterns*, or *translate*) the characters read from that line (padded with blanks if necessary) are to be placed into the *buf* array. Reading lines from the *dictionary* file should be as efficient as possible since this is part of PATGEN's "inner loop". Standard Pascal, unfortunately, does not provide for this; consequently the WEB macro *read_buf* defined below should be optimized if possible. For many Pascal's this can be done with *read_ln*(*f*, *buf*) where *buf* is declared as Pascal string (i.e., as **packed array** [1 .. *any*] **of** *char*), for others a string type with dynamic length can be used.

```
define read_buf(#) ≡   { reads a line from input file # into buf array }
      begin buf_ptr ← 0;
      while ¬eoln(#) do
        begin if (buf_ptr ≥ max_buf_len) then bad_input(´Line␣too␣long´);
        incr(buf_ptr); read(#, buf[buf_ptr]);
        end;
      read_ln(#);
      while buf_ptr < max_buf_len do
        begin incr(buf_ptr); buf[buf_ptr] ← ´␣´;
        end;
      end
```

⟨ Globals in the outer block 4 ⟩ +≡
buf: **array** [1 .. *max_buf_len*] **of** *text_char*;   { array to hold lines of input }
buf_ptr: 0 .. *max_buf_len*;   { index into *buf* }

53. When an error is caused by bad input data we say *bad_input*(#) in order to display the contents of the *buf* array before terminating with an error message.

```
define print_buf ≡   { print contents of buf array }
      begin buf_ptr ← 0;
      repeat incr(buf_ptr); print(buf[buf_ptr]);
      until buf_ptr = max_buf_len;
      print_ln(´␣´);
      end
define bad_input(#) ≡
        begin print_buf; error(#);
        end
```

54. The *translate* file may specify the values of `\lefthyphenmin` and `\righthyphenmin` as well as the external representation and collating sequence of the 'letters' used by the language. In addition replacements may be specified for the characters `'-'`, `'*'`, and `'.'` representing hyphens in the word list. If the *translate* file is empty (or does not exist) default values will be used.

```
procedure read_translate;
  label done;
  var c: text_char; n: integer; j: ASCII_code; bad: boolean; lower: boolean; i: dot_type;
    s, t: trie_pointer;
  begin imax ← edge_of_word; reset(translate);
  if eof(translate) then ⟨Set up default character translation tables 56⟩
  else begin read_buf(translate); ⟨Set up hyphenation data 57⟩;
    cmax ← last_ASCII_code − 1;
    while ¬eof(translate) do ⟨Set up representation(s) for a letter 58⟩;
    end;
  close_in(translate); print_ln('left_hyphen_min␣=␣', left_hyphen_min : 1, ',␣right_hyphen_min␣=␣',
      right_hyphen_min : 1, ',␣', imax − edge_of_word : 1, '␣letters'); cmax ← imax;
  end;
```

55. ⟨Globals in the outer block 4⟩ +≡

```
imax: internal_code;   { largest internal_code assigned so far }
left_hyphen_min, right_hyphen_min: dot_type;
```

56. ⟨Set up default character translation tables 56⟩ ≡

```
  begin left_hyphen_min ← 2; right_hyphen_min ← 3;
  for j ← "A" to "Z" do
    begin incr(imax); c ← xchr[j + "a" − "A"]; xclass[c] ← letter_class; xint[c] ← imax;
    xext[imax] ← c; c ← xchr[j]; xclass[c] ← letter_class; xint[c] ← imax;
    end;
  end
```

This code is used in section 54.

57. The first line of the *translate* file must contain the values of `\lefthyphenmin` and `\righthyphenmin` in columns 1–2 and 3–4. In addition columns 5, 6, and 7 may (optionally) contain replacements for the default characters `´.´`, `´-´`, and `´*´` respectively, representing hyphens in the word list. If the values specified for `\lefthyphenmin` and `\righthyphenmin` are invalid (e.g., blank) new values are read from the terminal.

⟨ Set up hyphenation data 57 ⟩ ≡

```
bad ← false; n ← 0;
if buf[1] = ´␣´ then do_nothing
else if xclass[buf[1]] = digit_class then n ← xint[buf[1]] else bad ← true;
if xclass[buf[2]] = digit_class then n ← 10 ∗ n + xint[buf[2]] else bad ← true;
if (n ≥ 1) ∧ (n < max_dot) then left_hyphen_min ← n else bad ← true;
n ← 0;
if buf[3] = ´␣´ then do_nothing
else if xclass[buf[3]] = digit_class then n ← xint[buf[3]] else bad ← true;
if xclass[buf[4]] = digit_class then n ← 10 ∗ n + xint[buf[4]] else bad ← true;
if (n ≥ 1) ∧ (n < max_dot) then right_hyphen_min ← n else bad ← true;
if bad then
  begin bad ← false;
  repeat print(´left_hyphen_min,␣right_hyphen_min:␣´); get_input(n1, n2);
    if (n1 ≥ 1) ∧ (n1 < max_dot) ∧ (n2 ≥ 1) ∧ (n2 < max_dot) then
      begin left_hyphen_min ← n1; right_hyphen_min ← n2;
      end
    else begin n1 ← 0;
      print_ln(´Specify␣1<=left_hyphen_min,right_hyphen_min<=´, max_dot − 1 : 1, ´␣!´);
      end;
  until n1 > 0;
  end;
for j ← err_hyf to found_hyf do
  begin if buf[j + 4] ≠ ´␣´ then xhyf[j] ← buf[j + 4];
  if xclass[xhyf[j]] = invalid_class then xclass[xhyf[j]] ← hyf_class else bad ← true;
  end;
xclass[´.´] ← hyf_class; { in case the default has been changed }
if bad then bad_input(´Bad␣hyphenation␣data´)
```

This code is used in section 54.

58. Each following line is either a comment or specifies the external representations for one 'letter' used by the language. Comment lines start with two equal characters (e.g., are blank) and are ignored. Other lines contain the external representation of the lower case version and an arbitrary number of 'upper case versions' of a letter preceded and separated by a delimiter and followed by two consecutive delimiters; the delimiter may be any character not occurring in either version.

```
⟨Set up representation(s) for a letter 58⟩ ≡
  begin read_buf(translate); buf_ptr ← 1; lower ← true;
  while ¬bad do   { lower and then upper case version }
    begin pat_len ← 0;
    repeat if buf_ptr < max_buf_len then incr(buf_ptr) else bad ← true;
      if buf[buf_ptr] = buf[1] then
        if pat_len = 0 then goto done
        else begin if lower then
              begin if imax = last_ASCII_code then
                begin print_buf; overflow(num_ASCII_codes : 1, ´␣letters´);
                end;
              incr(imax); xext[imax] ← xchr[pat[pat_len]];
              end;
            c ← xchr[pat[1]];
            if pat_len = 1 then
              begin if xclass[c] ≠ invalid_class then bad ← true;
              xclass[c] ← letter_class; xint[c] ← imax;
              end
            else ⟨Insert a letter into pattern trie 59⟩;
            end
      else if pat_len = max_dot then bad ← true
        else begin incr(pat_len); pat[pat_len] ← get_ASCII(buf[buf_ptr]);
          end;
    until (buf[buf_ptr] = buf[1]) ∨ bad;
    lower ← false;
    end;
done: if bad then bad_input(´Bad␣representation´);
  end
```

This code is used in section 54.

59. When the (lower or upper case) external representation of a letter consists of more than one character and the corresponding *ASCII_code* values have been placed into the *pat* array we store them in the pattern trie. [See *insert_pattern*.] Since this 'external subtrie' starts at *trie_link*(*trie_root*) it does not interfere with normal patterns. The output field of leaf nodes contains the *internal_code* and the link field distinguishes between lower and upper case letters.

```
⟨Insert a letter into pattern trie 59⟩ ≡
  begin if xclass[c] = invalid_class then xclass[c] ← escape_class;
  if xclass[c] ≠ escape_class then bad ← true;
  i ← 0; s ← trie_root; t ← trie_link(s);
  while (t > trie_root) ∧ (i < pat_len) do   { follow existing trie }
    begin incr(i); Incr(t)(pat[i]);
    if so(trie_char(t)) ≠ pat[i] then ⟨Insert critical transition, possibly repacking 42⟩
    else if trie_outp(t) > 0 then bad ← true;
    s ← t; t ← trie_link(s);
    end;
  if t > trie_root then bad ← true;
  q_link(1) ← 0; q_outp(1) ← 0; qmax ← 1;
  while i < pat_len do   { insert rest of pattern }
    begin incr(i); q_char(1) ← pat[i]; t ← first_fit; trie_link(s) ← t; s ← t + pat[i]; incr(trie_count);
    end;
  trie_outp(s) ← imax;
  if ¬lower then trie_link(s) ← trie_root;
  end
```

This code is used in section 58.

60. The *get_letter* WEB macro defined here will be used in *read_word* and *read_patterns* to obtain the *internal_code* corresponding to a letter externally represented by a multi-character sequence (starting with an *escape_class* character).

```
  define get_letter(#) ≡
        begin t ← trie_root;
        loop
          begin t ← trie_link(t) + xord[c];
          if so(trie_char(t)) ≠ xord[c] then bad_input('Bad␣representation');
          if trie_outp(t) ≠ 0 then
            begin # ← trie_outp(t); goto done;
            end;
          if buf_ptr = max_buf_len then c ← '␣'
          else begin incr(buf_ptr); c ← buf[buf_ptr];
            end;
          end;
      done: end
```

61. In order to prepare for the output phase we store all but the last of the *ASCII_codes* of the external representation of each 'lower case letter' in the pattern count trie which is no longer used at that time. The recursive *find_letters* procedure traverses the 'external subtrie'.

```
procedure find_letters(b : trie_pointer; i : dot_type);
        { traverse subtries of family b; i is current depth in trie }
  var c: ASCII_code;   { a local variable that must be saved on recursive calls }
    a: trie_pointer;   { does not need to be saved }
    j: dot_type;   { loop index }
    l: triec_pointer;
  begin if i = 1 then init_count_trie;
  for c ← cmin to last_ASCII_code do   { find transitions belonging to this family }
    begin a ← b + c;
    if so(trie_char(a)) = c then   { found one }
      begin pat[i] ← c;
      if trie_outp(a) = 0 then find_letters(trie_link(a), i + 1)
      else if trie_link(a) = 0 then   { this is a lower case letter }
          ⟨Insert external representation for a letter into count trie 62⟩;
      end;
    end;
  end;
```

62. Starting from $triec\_root + trie\_outp(a)$ we proceed through link fields and store all *ASCII_codes* except the last one in the count trie; the last character has already been stored in the *xext* array.

```
⟨Insert external representation for a letter into count trie 62⟩ ≡
  begin l ← triec_root + trie_outp(a);
  for j ← 1 to i − 1 do
    begin if triec_max = triec_size then overflow(triec_size : 1, ´␣count␣trie␣nodes´);
    incr(triec_max); triec_link(l) ← triec_max; l ← triec_max; triec_char(l) ← si(pat[j]);
    end;
  triec_link(l) ← 0;
  end
```

This code is used in section 61.

63. During the output phase we will say $write\_letter(i)(f)$ and $write(f, xext[i])$ to write the lower case external representation of the letter with internal code i to file f: $xext[i]$ is the last character of the external representation whereas the WEB macro *write_letter* defined here writes all preceding characters (if any).

```
  define write_letter_end(#) ≡
          while l > 0 do
            begin write(#, xchr[so(triec_char(l))]); l ← triec_link(l);
            end
  define write_letter(#) ≡ l ← triec_link(triec_root + #); write_letter_end
```

64. Routines for traversing pattern tries. At the end of a pass, we traverse the count trie using the following recursive procedure, selecting good and bad patterns and inserting them into the pattern trie.

procedure *traverse_count_trie*(*b* : *triec_pointer*; *i* : *dot_type*);
  { traverse subtries of family *b*; *i* is current depth in trie }
 var *c*: *internal_code*; { a local variable that must be saved on recursive calls }
  a: *triec_pointer*; { does not need to be saved }
 begin for $c \leftarrow cmin$ **to** *cmax* **do** { find transitions belonging to this family }
  begin $a \leftarrow b + c$;
  if *so*(*triec_char*(*a*)) = *c* **then** { found one }
   begin *pat*[*i*] ← *c*;
   if *i* < *pat_len* **then** *traverse_count_trie*(*triec_link*(*a*), *i* + 1)
   else ⟨ Decide what to do with this pattern 65 ⟩;
   end;
  end;
 end;

65. When we have come to the end of a pattern, *triec_good*(*a*) and *triec_bad*(*a*) contain the number of times this pattern helps or hinders the cause. We use the counts to determine if this pattern should be selected, or if it is hopeless, or if we can't decide yet. In the latter case, we set *more_to_come* true to indicate that there might still be good patterns extending the current type of patterns.

⟨ Decide what to do with this pattern 65 ⟩ ≡
 if *good_wt* ∗ *triec_good*(*a*) < *thresh* **then** { hopeless pattern }
  begin *insert_pattern*(*max_val*, *pat_dot*); *incr*(*bad_pat_count*)
  end
 else if *good_wt* ∗ *triec_good*(*a*) − *bad_wt* ∗ *triec_bad*(*a*) ≥ *thresh* **then** { good pattern }
   begin *insert_pattern*(*hyph_level*, *pat_dot*); *incr*(*good_pat_count*); *Incr*(*good_count*)(*triec_good*(*a*));
   Incr(*bad_count*)(*triec_bad*(*a*));
   end
  else *more_to_come* ← *true*

This code is used in section 64.

66. Some global variables are used to accumulate statistics about the performance of a pass.

⟨ Globals in the outer block 4 ⟩ +≡
good_pat_count, *bad_pat_count*: *integer*; { number of patterns added at end of pass }
good_count, *bad_count*, *miss_count*: *integer*; { hyphen counts }
level_pattern_count: *integer*; { number of good patterns at level }
more_to_come: *boolean*;

67. The recursion in *traverse_count_trie* is initiated by the following procedure, which also prints some statistics about the patterns chosen. The "efficiency" is an estimate of pattern effectiveness.

```
  define bad_eff ≡ (thresh/good_wt)
procedure collect_count_trie;
  begin good_pat_count ← 0; bad_pat_count ← 0; good_count ← 0; bad_count ← 0; more_to_come ← false;
  traverse_count_trie(triec_root, 1);
  print(good_pat_count : 1, ´␣good␣and␣´, bad_pat_count : 1, ´␣bad␣patterns␣added´);
  Incr(level_pattern_count)(good_pat_count);
  if more_to_come then print_ln(´␣(more␣to␣come)´) else print_ln(´␣´);
  print(´finding␣´, good_count : 1, ´␣good␣and␣´, bad_count : 1, ´␣bad␣hyphens´);
  if good_pat_count > 0 then
    print_ln(´,␣efficiency␣=␣´, good_count/(good_pat_count + bad_count/bad_eff) : 1 : 2)
  else print_ln(´␣´);
  print_ln(´pattern␣trie␣has␣´, trie_count : 1, ´␣nodes,␣´,
      ´trie_max␣=␣´, trie_max : 1, ´,␣´, op_count : 1, ´␣outputs´);
  end;
```

68. At the end of a level, we traverse the pattern trie and delete bad patterns by removing their outputs. If no output remains, the node is also deleted.

```
function delete_patterns(s : trie_pointer): trie_pointer;
      { delete bad patterns in subtrie s, return 0 if entire subtrie freed, otherwise s }
  var c: internal_code; t: trie_pointer; all_freed: boolean;   { must be saved on recursive calls }
    h, n: op_type;   { do not need to be saved }
  begin all_freed ← true;
  for c ← cmin to cmax do   { find transitions belonging to this family }
    begin t ← s + c;
    if so(trie_char(t)) = c then
      begin ⟨Link around bad outputs 69⟩;
      if trie_link(t) > 0 then trie_link(t) ← delete_patterns(trie_link(t));
      if (trie_link(t) > 0) ∨ (trie_outp(t) > 0) ∨ (s = trie_root) then all_freed ← false
      else ⟨Deallocate this node 70⟩;
      end;
    end;
  if all_freed then   { entire state is freed }
    begin trie_base_used(s) ← false; s ← 0;
    end;
  delete_patterns ← s;
  end;
```

69. ⟨Link around bad outputs 69⟩ ≡

```
  begin h ← 0; hyf_nxt(0) ← trie_outp(t); n ← hyf_nxt(0);
  while n > 0 do
    begin if hyf_val(n) = max_val then hyf_nxt(h) ← hyf_nxt(n)
    else h ← n;
    n ← hyf_nxt(h);
    end;
  trie_outp(t) ← hyf_nxt(0);
  end
```

This code is used in section 68.

70. Cells freed by *delete_patterns* are put at the end of the free list.

⟨ Deallocate this node 70 ⟩ ≡
begin *trie_link*(*trie_back*(*trie_max* + 1)) ← *t*; *trie_back*(*t*) ← *trie_back*(*trie_max* + 1);
trie_link(*t*) ← *trie_max* + 1; *trie_back*(*trie_max* + 1) ← *t*; *trie_char*(*t*) ← *min_packed*;
decr(*trie_count*);
end

This code is used in section 68.

71. The recursion in *delete_patterns* is initiated by the following procedure, which also prints statistics about the number of nodes deleted, and zeros bad outputs in the hash table. Note that the hash table may become somewhat disorganized when more levels are added, but this defect isn't serious.

```
procedure delete_bad_patterns;
  var old_op_count: op_type; old_trie_count: trie_pointer; t: trie_pointer; h: op_type;
  begin old_op_count ← op_count; old_trie_count ← trie_count;
  t ← delete_patterns(trie_root);
  for h ← 1 to max_ops do
    if hyf_val(h) = max_val then
      begin hyf_val(h) ← 0; decr(op_count);
      end;
  print_ln(old_trie_count − trie_count : 1, ´␣nodes␣and␣´,
      old_op_count − op_count : 1, ´␣outputs␣deleted´); qmax_thresh ← 7;
      { pattern trie will be sparser because of deleted patterns }
  end;
```

72. After all patterns have been generated, we will traverse the pattern trie and output all patterns. Note that if a pattern appears more than once, only the maximum value at each position will be output.

```
procedure output_patterns(s : trie_pointer; pat_len : dot_type);
      { output patterns in subtrie s; pat_len is current depth in trie }
  var c: internal_code;  { must be saved on recursive calls }
    t: trie_pointer; h: op_type; d: dot_type;
    l: triec_pointer;  { for write_letter }
  begin for c ← cmin to cmax do
    begin t ← s + c;
    if so(trie_char(t)) = c then
      begin pat[pat_len] ← c; h ← trie_outp(t);
      if h > 0 then ⟨ Output this pattern 73 ⟩;
      if trie_link(t) > 0 then output_patterns(trie_link(t), pat_len + 1);
      end;
    end;
  end;
```

73. ⟨Output this pattern 73⟩ ≡

```
begin for d ← 0 to pat_len do hval[d] ← 0;
repeat d ← hyf_dot(h);
  if hval[d] < hyf_val(h) then hval[d] ← hyf_val(h);
  h ← hyf_nxt(h);
until h = 0;
if hval[0] > 0 then write(patout, xdig[hval[0]]);
for d ← 1 to pat_len do
  begin write_letter(pat[d])(patout); write(patout, xext[pat[d]]);
  if hval[d] > 0 then write(patout, xdig[hval[d]]);
  end;
write_ln(patout);
end
```

This code is used in section 72.

74. Dictionary processing routines. The procedures in this section are the "inner loop" of the pattern generation process. To speed the program up, key parts of these routines could be coded in machine language.

⟨ Globals in the outer block 4 ⟩ +≡
word: **array** [*word_index*] **of** *internal_code*; { current word }
dots: **array** [*word_index*] **of** *hyf_type*; { current hyphens }
dotw: **array** [*word_index*] **of** *digit*; { dot weights }
hval: **array** [*word_index*] **of** *val_type*; { hyphenation values }
no_more: **array** [*word_index*] **of** *boolean*; { positions 'knocked out' }
wlen: *word_index*; { length of current word }
word_wt: *digit*; { global word weight }
wt_chg: *boolean*; { indicates *word_wt* has changed }

75. The words in the *dictionary* consist of the 'letters' used by the language. "Dots" between letters can be one of four possibilities: ´-´ indicating a hyphen, ´*´ indicating a found hyphen, ´.´ indicating an error, or nothing; these are represented internally by the four values *is_hyf*, *found_hyf*, *err_hyf*, and *no_hyf* respectively. When reading a word we will, however, convert *err_hyf* into *no_hyf* and *found_hyf* into *is_hyf* thus ignoring whether a hyphen has or has not been found by a previous set of patterns.

⟨ Prepare to read dictionary 75 ⟩ ≡
xclass[´.´] ← *invalid_class*; { in case the default has been changed }
xclass[*xhyf*[*err_hyf*]] ← *hyf_class*; *xint*[*xhyf*[*err_hyf*]] ← *no_hyf*; *xclass*[*xhyf*[*is_hyf*]] ← *hyf_class*;
xint[*xhyf*[*is_hyf*]] ← *is_hyf*; *xclass*[*xhyf*[*found_hyf*]] ← *hyf_class*; *xint*[*xhyf*[*found_hyf*]] ← *is_hyf*;

See also sections 79 and 85.

This code is used in section 88.

76. Furthermore single-digit word weights are allowed. A digit at the beginning of a word indicates a global word weight that is to be applied to all following words (until the next global word weight). A digit at some intercharacter position indicates a weight for that position only.

The *read_word* procedure scans a line of input representing a word, and places the letters into the array *word*, with *word*[1] = *word*[*wlen*] = *edge_of_word*. The dot appearing between *word*[*dpos*] and *word*[*dpos*+1] is placed in *dots*[*dpos*], and the corresponding dot weight in *dotw*[*dpos*].

```
procedure read_word;
  label done, found;
  var c: text_char; t: trie_pointer;
  begin read_buf(dictionary); word[1] ← edge_of_word; wlen ← 1; buf_ptr ← 0;
  repeat incr(buf_ptr); c ← buf[buf_ptr];
    case xclass[c] of
    space_class: goto found;
    digit_class: if wlen = 1 then   { global word weight }
        begin if xint[c] ≠ word_wt then wt_chg ← true;
        word_wt ← xint[c];
        end
      else dotw[wlen] ← xint[c];   { dot weight }
    hyf_class: dots[wlen] ← xint[c];   { record the dot c }
    letter_class:   { record the letter c }
      begin incr(wlen);
      if wlen = max_len then
        begin print_buf; overflow('word␣length=', max_len : 1);
        end;
      word[wlen] ← xint[c]; dots[wlen] ← no_hyf; dotw[wlen] ← word_wt;
      end;
    escape_class:   { record a multi-character sequence starting with c }
      begin incr(wlen);
      if wlen = max_len then
        begin print_buf; overflow('word␣length=', max_len : 1);
        end;
      get_letter(word[wlen]); dots[wlen] ← no_hyf; dotw[wlen] ← word_wt;
      end;
    invalid_class: bad_input('Bad␣character');
    end;
  until buf_ptr = max_buf_len;
found: incr(wlen); word[wlen] ← edge_of_word;
  end;
```

77. Here is a procedure that uses the existing patterns to hyphenate the current word. The hyphenation value applying between the characters *word*[*dpos*] and *word*[*dpos* + 1] is stored in *hval*[*dpos*].

In addition, *no_more*[*dpos*] is set to *true* if this position is "knocked out" by either a good or bad pattern at this level. That is, if the pattern with current length and hyphen position is a superstring of either a good or bad pattern at this level, then we don't need to collect count statistics for the pattern because it can't possibly be chosen in this pass. Thus we don't even need to insert such patterns into the count trie, which saves a good deal of space.

```
procedure hyphenate;
  label done;
  var spos, dpos, fpos: word_index; t: trie_pointer; h: op_type; v: val_type;
  begin for spos ← wlen − hyf_max downto 0 do
    begin no_more[spos] ← false; hval[spos] ← 0; fpos ← spos + 1; t ← trie_root + word[fpos];
    repeat h ← trie_outp(t);
      while h > 0 do ⟨Store output h in the hval and no_more arrays, and advance h 80⟩;
      t ← trie_link(t);
      if t = 0 then goto done;
      incr(fpos); Incr(t)(word[fpos]);
    until so(trie_char(t)) ≠ word[fpos];
  done: end;
  end;
```

78. In order to avoid unnecessary test (and range check violations) the globals *hyf_min*, *hyf_max*, and *hyf_len* are set up such that only positions from *hyf_min* up to *wlen* − *hyf_max* of the *word* array need to be checked, and that words with *wlen* < *hyf_len* need not to be checked at all.

```
⟨Globals in the outer block 4⟩ +≡
hyf_min, hyf_max, hyf_len: word_index;  { limits for legal hyphens }
```

79.
```
⟨Prepare to read dictionary 75⟩ +≡
  hyf_min ← left_hyphen_min + 1; hyf_max ← right_hyphen_min + 1; hyf_len ← hyf_min + hyf_max;
```

80.
```
⟨Store output h in the hval and no_more arrays, and advance h 80⟩ ≡
  begin dpos ← spos + hyf_dot(h); v ← hyf_val(h);
  if (v < max_val) ∧ (hval[dpos] < v) then hval[dpos] ← v;
  if (v ≥ hyph_level) then  { check if position knocked out }
    if ((fpos − pat_len) ≤ (dpos − pat_dot)) ∧ ((dpos − pat_dot) ≤ spos) then no_more[dpos] ← true;
  h ← hyf_nxt(h);
  end
```

This code is used in section 77.

81. The *change_dots* procedure updates the *dots* array representing the printing values of the hyphens. Initially, hyphens (and correctly found hyphens) in the word list are represented by *is_hyf* whereas non-hyphen positions (and erroneous hyphens) are represented by *no_hyf*. Here these values are increased by one for each hyphen found by the current patterns, thus changing *no_hyf* into *err_hyf* and *is_hyf* into *found_hyf*. The routine also collects statistics about the number of good, bad, and missed hyphens.

```
define incr_wt(#) ≡ Incr(#)(dotw[dpos])

procedure change_dots;
  var dpos: word_index;
  begin for dpos ← wlen − hyf_max downto hyf_min do
    begin if odd(hval[dpos]) then incr(dots[dpos]);
    if dots[dpos] = found_hyf then incr_wt(good_count)
    else if dots[dpos] = err_hyf then incr_wt(bad_count)
      else if dots[dpos] = is_hyf then incr_wt(miss_count);
    end;
  end;
```

82. The following procedure outputs the word as hyphenated by the current patterns, including any word weights. Hyphens inhibited by the values of `\lefthyphenmin` and `\righthyphenmin` are output as well.

```
procedure output_hyphenated_word;
  var dpos: word_index;
    l: triec_pointer;  { for write_letter }
  begin if wt_chg then  { output global word weight }
    begin write(pattmp, xdig[word_wt]); wt_chg ← false
    end;
  for dpos ← 2 to wlen − 2 do
    begin write_letter(word[dpos])(pattmp); write(pattmp, xext[word[dpos]]);
    if dots[dpos] ≠ no_hyf then write(pattmp, xhyf[dots[dpos]]);
    if dotw[dpos] ≠ word_wt then write(pattmp, xdig[dotw[dpos]]);
    end;
  write_letter(word[wlen − 1])(pattmp); write_ln(pattmp, xext[word[wlen − 1]]);
  end;
```

83. For each dot position in the current word, the *do_word* routine first checks to see if we need to consider it. It might be knocked out or a dot we don't care about. That is, when considering hyphenating patterns, for example, we don't need to count hyphens already found. If a relevant dot is found, we increment the count in the count trie for the corresponding pattern, inserting it first if necessary. At this point of the program range check violations may occur if these counts are incremented beyond *triec_max*; it would, however, be too expensive to prevent this.

```
procedure do_word;
  label continue, done;
  var spos, dpos, fpos: word_index; a: triec_pointer; goodp: boolean;
  begin for dpos ← wlen − dot_max downto dot_min do
    begin spos ← dpos − pat_dot; fpos ← spos + pat_len;
    ⟨Check this dot position and goto continue if don't care 86⟩;
    incr(spos); a ← triec_root + word[spos];
    while spos < fpos do
      begin   { follow existing count trie }
      incr(spos); a ← triec_link(a) + word[spos];
      if so(triec_char(a)) ≠ word[spos] then
        begin   { insert new count pattern }
        a ← insertc_pat(fpos); goto done;
        end;
      end;
  done: if goodp then incr_wt(triec_good(a)) else incr_wt(triec_bad(a));
  continue: end;
  end;
```

84. The globals *good_dot* and *bad_dot* will be set to *is_hyf* and *no_hyf*, or *err_hyf* and *found_hyf*, depending on whether the current level is odd or even, respectively. The globals *dot_min*, *dot_max*, and *dot_len* are analogous to *hyf_min*, *hyf_max*, and *hyf_len* defined earlier.

```
⟨Globals in the outer block 4⟩ +≡
good_dot, bad_dot: hyf_type;   { good and bad hyphens at current level }
dot_min, dot_max, dot_len: word_index;   { limits for legal dots }
```

85.
```
⟨Prepare to read dictionary 75⟩ +≡
  if procesp then
    begin dot_min ← pat_dot; dot_max ← pat_len − pat_dot;
    if dot_min < hyf_min then dot_min ← hyf_min;
    if dot_max < hyf_max then dot_max ← hyf_max;
    dot_len ← dot_min + dot_max;
    if odd(hyph_level) then
      begin good_dot ← is_hyf; bad_dot ← no_hyf;
      end
    else begin good_dot ← err_hyf; bad_dot ← found_hyf;
      end;
    end;
```

86. If the dot position *dpos* is out of bounds, knocked out, or a “don’t care”, we skip this position. Otherwise we set the flag *goodp* indicating whether this is a good or bad dot.

⟨ Check this dot position and **goto** *continue* if don’t care 86 ⟩ ≡

```
if no_more[dpos] then goto continue;
if dots[dpos] = good_dot then goodp ← true
else if dots[dpos] = bad_dot then goodp ← false
  else goto continue;
```

This code is used in section 83.

87. If *hyphp* is set to *true*, *do_dictionary* will write out a copy of the dictionary as hyphenated by the current set of patterns. If *procesp* is set to *true*, *do_dictionary* will collect pattern statistics for patterns with length *pat_len* and hyphen position *pat_dot*, at level *hyph_level*.

⟨ Globals in the outer block 4 ⟩ +≡

```
procesp, hyphp: boolean;
pat_dot: dot_type;   { hyphen position, measured from beginning of pattern }
hyph_level: val_type;   { hyphenation level }
filnam: packed array [1 .. 8] of char;   { for pattmp }
```

88. The following procedure makes a pass through the word list, and also prints out statistics about number of hyphens found and storage used by the count trie.

```
procedure do_dictionary;
  begin good_count ← 0; bad_count ← 0; miss_count ← 0; word_wt ← 1; wt_chg ← false;
  reset(dictionary);
  ⟨ Prepare to read dictionary 75 ⟩
  if procesp then
    begin init_count_trie;
    print_ln(´processing␣dictionary␣with␣pat_len␣=␣´, pat_len : 1, ´,␣pat_dot␣=␣´, pat_dot : 1);
    end;
  if hyphp then
    begin filnam ← ´pattmp.␣´; filnam[8] ← xdig[hyph_level]; rewrite(pattmp, filnam);
    print_ln(´writing␣pattmp.´, xdig[hyph_level]);
    end;
  ⟨ Process words until end of file 89 ⟩;
  close_in(dictionary);
  print_ln(´␣´); print_ln(good_count : 1, ´␣good,␣´, bad_count : 1, ´␣bad,␣´, miss_count : 1, ´␣missed´);
  if (good_count + miss_count) > 0 then print_ln((100 ∗ good_count/(good_count + miss_count)) : 1 : 2,
        ´␣%,␣´, (100 ∗ bad_count/(good_count + miss_count)) : 1 : 2, ´␣%,␣´,
        (100 ∗ miss_count/(good_count + miss_count)) : 1 : 2, ´␣%´);
  if procesp then print_ln(pat_count : 1, ´␣patterns,␣´, triec_count : 1, ´␣nodes␣in␣count␣trie,␣´,
        ´triec_max␣=␣´, triec_max : 1);
  if hyphp then close_out(pattmp);
  end;
```

89. ⟨ Process words until end of file 89 ⟩ ≡

```
while ¬eof(dictionary) do
  begin read_word;
  if wlen ≥ hyf_len then   { short words are never hyphenated }
    begin hyphenate; change_dots;
    end;
  if hyphp then
    if wlen > 2 then output_hyphenated_word;   { empty words are ignored }
  if procesp then
    if wlen ≥ dot_len then do_word;
  end
```

This code is used in section 88.

90. Reading patterns. Before beginning a run, we can read in a file of existing patterns. This is useful for extending a previous pattern selection run to get some more levels. (Since these runs are quite time-consuming, it is convenient to choose patterns one level at a time, pausing to look at the results of the previous level, and possibly amending the dictionary.)

```
procedure read_patterns;
  label done, found;
  var c: text_char; d: digit; i: dot_type; t: trie_pointer;
  begin xclass['.'] ← letter_class; xint['.'] ← edge_of_word; level_pattern_count ← 0; max_pat ← 0;
  reset(patterns);
  while ¬eof(patterns) do
    begin read_buf(patterns); incr(level_pattern_count);
    ⟨Get pattern and dots and goto found 92⟩;
  found: ⟨Insert pattern 93⟩;
    end;
  close_in(patterns); print_ln(level_pattern_count : 1, '␣patterns␣read␣in');
  print_ln('pattern␣trie␣has␣', trie_count : 1, '␣nodes,␣',
      'trie_max␣=␣', trie_max : 1, ',␣', op_count : 1, '␣outputs');
  end;
```

91. The global variable *max_pat* keeps track of the largest hyphenation value found in any pattern.

```
⟨Globals in the outer block 4⟩ +≡
max_pat: val_type;
```

92. When a new pattern has been input into *buf*, we extract the letters of the pattern, and insert the hyphenation values (digits) into the *hval* array.

```
⟨Get pattern and dots and goto found 92⟩ ≡
  pat_len ← 0; buf_ptr ← 0; hval[0] ← 0;
  repeat incr(buf_ptr); c ← buf[buf_ptr];
    case xclass[c] of
    space_class: goto found;
    digit_class: begin d ← xint[c];
      if d ≥ max_val then bad_input('Bad␣hyphenation␣value');
      if d > max_pat then max_pat ← d;
      hval[pat_len] ← d;
      end;
    letter_class: begin incr(pat_len); hval[pat_len] ← 0; pat[pat_len] ← xint[c];
      end;
    escape_class:   { record a multi-character sequence starting with c }
      begin incr(pat_len); hval[pat_len] ← 0; get_letter(pat[pat_len]);
      end;
    hyf_class, invalid_class: bad_input('Bad␣character');
    end;
  until buf_ptr = max_buf_len
```

This code is used in section 90.

93. Then we insert the pattern for each non-vanishing hyphenation value. In addition we check that *edge_of_word* (i.e., `´.´`) occurs only as first or last character; otherwise we would have to perform a time consuming test for the end of a word in the *hyphenate* procedure.

⟨ Insert pattern 93 ⟩ ≡

```
if pat_len > 0 then  { avoid spurious patterns }
  for i ← 0 to pat_len do
    begin if hval[i] ≠ 0 then insert_pattern(hval[i], i);
    if i > 1 then
      if i < pat_len then
        if pat[i] = edge_of_word then bad_input(´Bad␣edge_of_word´);
    end
```

This code is used in section 90.

94. The main program. This is where PATGEN actually starts. We initialize the pattern trie, get *hyph_level* and *pat_len* limits from the terminal, and generate patterns.

```
begin initialize; init_pattern_trie; read_translate; read_patterns; procesp ← true; hyphp ← false;
repeat print(´hyph_start,␣hyph_finish:␣´); get_input(n1, n2);
  if (n1 ≥ 1) ∧ (n1 < max_val) ∧ (n2 ≥ 1) ∧ (n2 < max_val) then
    begin hyph_start ← n1; hyph_finish ← n2;
    end
  else begin n1 ← 0; print_ln(´Specify␣1<=hyph_start,hyph_finish<=´, max_val − 1 : 1, ´␣!´);
    end;
until n1 > 0;
hyph_level ← max_pat; { in case hyph_finish < hyph_start }
for i ← hyph_start to hyph_finish do
  begin hyph_level ← i; level_pattern_count ← 0;
  if hyph_level > hyph_start then print_ln(´␣´)
  else if hyph_start ≤ max_pat then print_ln(´Largest␣hyphenation␣value␣´, max_pat : 1,
        ´␣in␣patterns␣should␣be␣less␣than␣hyph_start´);
  repeat print(´pat_start,␣pat_finish:␣´); get_input(n1, n2);
    if (n1 ≥ 1) ∧ (n1 ≤ n2) ∧ (n2 ≤ max_dot) then
      begin pat_start ← n1; pat_finish ← n2;
      end
    else begin n1 ← 0; print_ln(´Specify␣1<=pat_start<=pat_finish<=´, max_dot : 1, ´␣!´);
      end;
  until n1 > 0;
  repeat print(´good␣weight,␣bad␣weight,␣threshold:␣´); get_input(n1, n2, n3);
    if (n1 ≥ 1) ∧ (n2 ≥ 1) ∧ (n3 ≥ 1) then
      begin good_wt ← n1; bad_wt ← n2; thresh ← n3;
      end
    else begin n1 ← 0; print_ln(´Specify␣good␣weight,␣bad␣weight,␣threshold>=1␣!´);
      end;
  until n1 > 0;
  ⟨Generate a level 96⟩;
  delete_bad_patterns;
  print_ln(´total␣of␣´, level_pattern_count : 1, ´␣patterns␣at␣hyph_level␣´, hyph_level : 1);
  end;
find_letters(trie_link(trie_root), 1); { prepare for output }
rewrite(patout); output_patterns(trie_root, 1); close_out(patout);
⟨Make final pass to hyphenate word list 97⟩;
end_of_PATGEN: end.
```

95. The patterns of a given length (at a given level) are chosen with dot positions ordered in an "organ-pipe" fashion. For example, for *pat_len* = 4 we choose patterns for different dot positions in the order 2, 1, 3, 0, 4. The variable *dot1* controls this iteration in a clever manner.

```
⟨Globals in the outer block 4⟩ +≡
n1, n2, n3: integer; { accumulators }
i: val_type; { loop index: hyphenation level }
j: dot_type; { loop index: pattern length }
k: dot_type; { loop index: hyphen position }
dot1: dot_type;
more_this_level: array [dot_type] of boolean;
```

96. The array *more_this_level* remembers which positions are permanently "knocked out". That is, if there aren't any possible good patterns remaining at a certain dot position, we don't need to consider longer patterns at this level containing that position.

⟨ Generate a level 96 ⟩ ≡
for $k \leftarrow 0$ **to** *max_dot* **do** *more_this_level*[k] ← *true*;
for $j \leftarrow$ *pat_start* **to** *pat_finish* **do**
  begin *pat_len* ← j; *pat_dot* ← *pat_len* **div** 2; *dot1* ← *pat_dot* ∗ 2;
  repeat *pat_dot* ← *dot1* − *pat_dot*; *dot1* ← *pat_len* ∗ 2 − *dot1* − 1;
    if *more_this_level*[*pat_dot*] **then**
      begin *do_dictionary*; *collect_count_trie*; *more_this_level*[*pat_dot*] ← *more_to_come*;
      end;
  until *pat_dot* = *pat_len*;
  for $k \leftarrow$ *max_dot* **downto** 1 **do**
    if ¬*more_this_level*[$k - 1$] **then** *more_this_level*[k] ← *false*;
  end

This code is used in section 94.

97. When all patterns have been found, the user has a chance to see what they do. The resulting `pattmp` file can be used as the new 'dictionary' if we want to continue pattern generation from this point.

⟨ Make final pass to hyphenate word list 97 ⟩ ≡
procesp ← *false*; *hyphp* ← *true*;
print(´`hyphenate␣word␣list?␣`´); *get_input_ln*(*buf*[1]);
if (*buf*[1] = ´`Y`´) ∨ (*buf*[1] = ´`y`´) **then** *do_dictionary*

This code is used in section 94.

98. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make PATGEN work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

99. Index. Pointers to error messages appear here together with the section numbers where each identifier is used.

⟨ Check the "constant" values for consistency 28 ⟩ Used in section 14.
⟨ Check this dot position and **goto** *continue* if don't care 86 ⟩ Used in section 83.
⟨ Compiler directives 11 ⟩ Used in section 3.
⟨ Constants in the outer block 27 ⟩ Used in section 3.
⟨ Deallocate this node 70 ⟩ Used in section 68.
⟨ Decide what to do with this pattern 65 ⟩ Used in section 64.
⟨ Ensure *triec* linked up to $b + num\_ASCII\_codes$ 47 ⟩ Used in section 46.
⟨ Ensure *trie* linked up to $s + num\_ASCII\_codes$ 37 ⟩ Used in section 36.
⟨ Generate a level 96 ⟩ Used in section 94.
⟨ Get pattern and dots and **goto** *found* 92 ⟩ Used in section 90.
⟨ Globals in the outer block 4, 16, 23, 25, 30, 31, 33, 40, 43, 51, 52, 55, 66, 74, 78, 84, 87, 91, 95 ⟩ Used in section 3.
⟨ Insert a letter into pattern trie 59 ⟩ Used in section 58.
⟨ Insert critical count transition, possibly repacking 50 ⟩ Used in section 49.
⟨ Insert critical transition, possibly repacking 42 ⟩ Used in sections 41 and 59.
⟨ Insert external representation for a letter into count trie 62 ⟩ Used in section 61.
⟨ Insert pattern 93 ⟩ Used in section 90.
⟨ Link around bad outputs 69 ⟩ Used in section 68.
⟨ Local variables for initialization 15 ⟩ Used in section 3.
⟨ Make final pass to hyphenate word list 97 ⟩ Used in section 94.
⟨ Output this pattern 73 ⟩ Used in section 72.
⟨ Prepare to read dictionary 75, 79, 85 ⟩ Used in section 88.
⟨ Process words until end of file 89 ⟩ Used in section 88.
⟨ Set initial values 14, 17, 18, 24 ⟩ Used in section 3.
⟨ Set up default character translation tables 56 ⟩ Used in section 54.
⟨ Set up hyphenation data 57 ⟩ Used in section 54.
⟨ Set up representation(s) for a letter 58 ⟩ Used in section 54.
⟨ Set b to the count trie base location at which this state should be packed 46 ⟩ Used in section 45.
⟨ Set s to the trie base location at which this state should be packed 36 ⟩ Used in section 35.
⟨ Store output h in the *hval* and *no_more* arrays, and advance h 80 ⟩ Used in section 77.
⟨ Types in the outer block 12, 13, 20, 22, 29 ⟩ Used in section 3.