Appendix

PATtern GENeration program
for the TEX82 hyphenator

(Version 2.4, April 2020)
Introduction. This program takes a list of hyphenated words and generates a set of patterns that can be used by the \TeX{} 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 \TeX{} source.

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

\begin{verbatim}
define my_name ≡ 'patgen'
define banner ≡ 'This is PATGEN, Version 2.4' { printed when the program starts }
\end{verbatim}

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.

\begin{verbatim}
define print(#) ≡ write(output,#)
define print_ln(#) ≡ writeLn(output,#)
define get_input(#) ≡ # ← input_int(std_input)
define get_input_ln(#) ≡
  begin # ← getc(std_input); readLn(std_input);
  end
define std_input ≡ stdin
\end{verbatim}

\begin{verbatim}
program PATGEN(dictionary, patterns, translate, patout);
const { Constants in the outer block 27*}
type { Types in the outer block 12*}
var { Globals in the outer block 4}
  (Define parse_arguments 98*)
procedure initialize; { this procedure gets things started properly }
  var { Local variables for initialization 15}
    begin kpse_set_program_name(argv[0], my_name); parse_arguments; print(banner);
      print_ln(version_string); (Set initial values 14)
    end;
\end{verbatim}

In case of serious problems PATGEN will give up, after issuing an error message about what caused the error.

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

\begin{verbatim}
define error(#) ≡
  begin writeLn(stderr,#); uexit(1);
  end;
define overflow(#) ≡ error(‘PATGEN:capacity exceeded, sorry’)\end{verbatim}
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*)

```
ASCII_code = 0 .. last_ASCII_code; { internal representation of input characters }
text_char = ASCII_code;       { the data type of characters in text files }
text_file = text;
```

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

This code is used in section 3*.
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 T\(\text{E}\)X82 hyphenation patterns (file \texttt{hyphen.tex}).

\begin{verbatim}
27* \{ Constants in the outer block \}
\begin{verbatim}
trie_size = 10000000; \{ space for pattern trie \}
triec_size = 5000000; \{ space for pattern count trie, must be less than \textit{trie}\_size and greater than the number of occurrences of any pattern in the dictionary \}
max_ops = 4080; \{ size of output hash table, should be a multiple of 510 \}
max_val = 10; \{ maximum number of levels+1, also used to denote bad patterns \}
max_dot = 15; \{ maximum pattern length, also maximum length of external representation of a ‘letter’ \}
max_len = 50; \{ maximum word length \}
max_buf_len = 3000; \{ maximum length of input lines, must be at least \textit{max}\_len \}
\end{verbatim}
\end{verbatim}

This code is used in section \textit{3*}. 

\textit{3*}
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.

\[
\text{define close_out}(\#) \equiv \text{xfclose}(\#, \text{`outputfile`}) \quad \text{\{close an output file\}}
\]

\[
\text{define close_in}(\#) \equiv \text{xfclose}(\#, \text{`inputfile`}) \quad \text{\{close an input file\}}
\]

(Globals in the outer block 4) \(\equiv\)

\begin{align*}
\text{dictionary, patterns, translate, patout, pattmp: text\_file; } \\
\text{f\_name: \text{↑char;}} \\
\text{bad\_frac, denom, eff: real;}
\end{align*}

54* The `translate` file may specify the values of `\left\text{hyphenmin}` and `\right\text{hyphenmin}` 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.

\[
\text{procedure read\_translate;}
\]

\[
\text{label done;}
\]

\[
\text{var c: text\_char; n: integer; j: ASCII\_code; bad: boolean; lower: boolean; i: dot\_type;}
\]

\[
\text{s, t: trie\_pointer;}
\]

\[
\text{begin imax \leftarrow edge\_of\_word; f\_name \leftarrow cmdline(4); reset(translate, f\_name);}
\]

\[
\text{if eof(translate) then \{Set up default character translation tables 56\}}
\]

\[
\text{else begin read\_buf(translate); \{Set up hyphenation data 57*\};}
\]

\[
\text{cmax \leftarrow last\_ASCII\_code - 1;}
\]

\[
\text{while \neg eof(translate) do \{Set up representation(s) for a letter 58\};}
\]

\[
\text{end;}
\]

\[
\text{close\_in(translate); print\_ln(`left\_hyphen\_min = 1`, `left\_hyphen\_min = 1`, `right\_hyphen\_min = 1`, `right\_hyphen\_min = 1`, `imax - edge\_of\_word = 1`, `letters`); cmax \leftarrow imax;}
\]

\[
\text{end;}
\]
The first line of the translate file must contain the values of \texttt{\lefthyphenmin} and \texttt{\righthyphenmin} in columns 1–2 and 3–4. In addition columns 5, 6, and 7 may (optionally) contain replacements for the default characters \texttt{`-'}, \texttt{`−'}, and \texttt{`∗'} respectively, representing hyphens in the word list. If the values specified for \texttt{\lefthyphenmin} and \texttt{\righthyphenmin} are invalid (e.g., blank) new values are read from the terminal.

\begin{verbatim}
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: ¦´); input 2ints(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´)
\end{verbatim}

This code is used in section 54*.
67* The recursion in \texttt{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.

\begin{verbatim}
define bad\_eff \equiv (thresh / good\_wt)
procedure collect\_count\_trie;
    begin good\_pat\_count \leftarrow 0; bad\_pat\_count \leftarrow 0; good\_count \leftarrow 0; bad\_count \leftarrow 0; more\_to\_come \leftarrow false;
    traverse\_count\_trie(triec\_root, 1);
    print(good\_pat\_count : 1, \text{"good\_and\_\", bad\_pat\_count : 1, \text{"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, \text{"good\_and\_\", bad\_count : 1, \text{"bad\_hyphens\"});
    if good\_pat\_count > 0 then
        begin print(\"\", \text{"efficiency\_\"}; print\_real(good\_count / (good\_pat\_count + bad\_count / bad\_eff), 1, 2);
            write\_ln(output);
        end
    else print\_ln(\"\\\")
    print\_ln(\"pattern\_trie\_has\", trie\_count : 1, \text{"nodes\_\", \"trie\_max\_\", trie\_max : 1, \text{"op\_count\_\", op\_count : 1, \text{"outputs\"});
end;
\end{verbatim}
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;
f_name ← cmdline(1); reset(dictionary,f_name); (Prepare to read dictionary)

if procesp then
    begin
        init_count_trie;
        println(`processing dictionary with pat_len=X, pat_len: 1, pat_dot=X, pat_dot: 1`);
    end;
if hyphp then
    begin
        strcpy(fnam, `pattmp.`); filnam[7] ← xdig[hyph_level]; rewrite(pattmp,filnam);
        println(`writing pattmp.`);
    end;
⟨Process words until end of file⟩;
close in(dictionary);
println(`good count: 1, bad count: 1, miss count: 1, missed`);
if (good_count + miss_count) > 0 then
    begin
        println(`patterns: 1, trie_count: 1, `);
        println(`triec_max: 1`);
        if hyphp then close out(pattmp);
    end;
§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;
f_name ← cmdline(2); reset(patterns,f_name);
while ¬eof(patterns) do
   begin read_buf(patterns); incr(level_pattern_count);
      ⟨Get pattern and dots and goto found⟩;
   found: ⟨Insert pattern⟩;
   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”));
The main program. This is where \textsc{patgen} actually starts. We initialize the pattern trie, get `hyph_level` and `pat_len` limits from the terminal, and generate patterns.

\begin{verbatim}
begin initialize; init_pattern_trie; read_translate; read_patterns; procesp ← true; hyphp ← false;
repeat print("hyph_start, hyph_finish:"); input_2ints(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; printLn("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 printLn(" ");
  else if hyph_start ≤ max_pat then printLn("Largest hyphenation value, max_pat: 1, in patterns should be less than hyph_start");
  repeat printLn("pat_start, pat_finish:"); input_2ints(n1, n2);
    if (n1 ≥ 1) ∧ (n1 ≤ n2) ∧ (n2 ≤ max_dot) then
      begin pat_start ← n1; pat_finish ← n2;
    end else begin n1 ← 0; printLn("Specify 1<=pat_start<=pat_finish<=' max_dot: 1, '!");
      end;
  until n1 > 0;
repeat printLn("good_weight, bad_weight, threshold:"); input_3ints(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; printLn("Specify good_weight, bad_weight, threshold>=1!");
      end;
  until n1 > 0;
⟨Generate a level 96⟩;
delete_bad_patterns;
printLn("total of patterns at hyph_level: 1");
end;
find_letters(trie_link(trie_root, 1));  { prepare for output }
fname ← cmdline(3); rewrite(patout, fname); output_patterns(trie_root, 1); close_out(patout);
⟨Make final pass to hyphenate word list 97⟩;
end.
\end{verbatim}
98* System-dependent changes. Parse a Unix-style command line.

```plaintext
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
(Define parse_arguments 98*) ≡

procedure parse_arguments;
const n_options = 2; {Pascal won’t count array lengths for us.}
var long_options: array [0 .. n_options] of getopt_struct;
getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options;
begin (Define the option table 99*)
repeat getopt_return_val ← getopt_long_only(argc, argv, '*', long_options, address_of (option_index));
  if getopt_return_val = −1 then
    begin do nothing;
    end
  else if getopt_return_val = '?' then
    begin usage(my_name);
    end
  else if argument_is('help') then
    begin usage_help(PATGEN_HELP, nil);
    end
  else if argument_is('version') then
    begin
      print_version_and_exit(banner, nil, 'Frank_M.Liang_and_Peter_Breitenlohner', nil);
      end;
  until getopt_return_val = −1; {Now optind is the index of first non-option on the command line.}
if (optind + 4 ≠ argc) then
  begin writeLn(stderr, my_name, ': Need exactly four arguments.'); usage(my_name);
  end;
end;
```

This code is used in section 3*.

99* Here are the options we allow. The first is one of the standard GNU options.

```plaintext
(Define the option table 99*) ≡
current_option ← 0; long_options[current_option].name ← 'help';
long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
long_options[current_option].val ← 0; incr(current_option);
```

See also sections 100* and 101*.

This code is used in section 98*.

100* Another of the standard options.

```plaintext
(Define the option table 99*) +≡
long_options[current_option].name ← 'version'; long_options[current_option].has_arg ← 0;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

101* An element with all zeros always ends the list.

```plaintext
(Define the option table 99*) +≡
long_options[current_option].name ← 0; long_options[current_option].has_arg ← 0;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0;
```
102* Index. Pointers to error messages appear here together with the section numbers where each identifier is used.

The following sections were changed by the change file: 1, 3, 10, 12, 27, 51, 54, 57, 67, 88, 90, 94, 98, 99, 100, 101, 102.

- help: 99*
- version: 100*
a: 45, 48, 49, 61, 64, 83.
address_of: 98*
all_freed: 68.
any: 52.
argv: 98*
argument_is: 98*
argv: 3*, 98*
ASCII_code: 2, 12*, 13, 14, 15, 16, 18, 19, 20, 21, 22, 25, 54*, 59, 61.
b: 45, 48, 49, 61, 64.
bad: 4, 14, 15, 28, 54*, 57*, 58, 59.
Bad character: 76, 92.
Bad constants: 14.
Bad edge_of_word: 93.
Bad hyphenation data: 57*
Bad hyphenation value: 92.
Bad representation: 58, 60.
bad_count: 65, 66, 67*, 81, 88*
bad_dot: 84, 85, 86.
bad_eff: 67*
bad_frac: 51*
bad_input: 52, 53, 57*, 58, 60, 76, 92, 93.
bad_pat_count: 65, 66, 67*
bad_set: 4, 65, 94*
big: 1*, 3*, 98*
boolean: 30, 54* 66, 68, 74, 83, 87, 95.
Breitenlochner, Peter: 2.
buf: 52, 53, 57*, 58, 60, 76, 92, 97.
buf_ptr: 52, 53, 58, 60, 76, 92.
c: 19, 34, 38, 44, 48, 54*, 61, 64, 68, 72, 76, 90*
c_int_type: 98*
change_dots: 81, 89.
char: 13, 51*, 52, 87.
character set dependencies: 12*, 18.
chr: 12*, 16, 18, 24.
class_type: 22, 23.
close_in: 51*, 54*, 88*, 90*
close_out: 51*, 88*, 94*
cmax: 25, 38, 48, 54*, 64, 68, 72.
cmdline: 54*, 88*, 90*, 94*.
cmin: 25, 38, 48, 61, 64, 68, 72.
collect_count_trie: 67*, 96.
continue: 8, 83, 86.
current_option: 98*, 99*, 100*, 101*.
d: 39, 72, 90*
Decr: 9.
deer: 9, 39, 70, 71.
delete_bad_patterns: 36, 71, 94*
delete_patterns: 68, 70, 71.
denom: 51*
dictionary: 21, 22, 51*, 52, 73, 75, 76, 88*, 89.
digit: 22, 74, 90*
digit_class: 22, 24, 57*, 76, 92.
do_nothing: 9, 57*, 98*
do_word: 83, 89.
done: 8, 54*, 58, 60, 76, 77, 83, 90*
dot: 29, 32, 41.
dot_len: 84, 85, 89.
dot_max: 83, 84, 85.
dot_min: 83, 84, 85.
dot_type: 4, 29, 39, 40, 41, 54*, 67*, 81, 64, 72, 87, 90*, 95.
dots: 74, 76, 81, 82, 86.
dotw: 74, 76, 81, 82.
dotl: 95, 96.
dpos: 76, 77, 80, 81, 82, 83, 86.
d setw: 20, 24, 25, 54*, 76, 90*, 93.
eff: 51*
eof: 54*, 89, 90*
eoln: 52.
err_buf: 22, 23, 24, 57*, 75, 81, 84, 85.
error: 10*, 14, 53.
escape_class: 22, 59, 60, 76, 92.
exit: 8, 9, 39.
f_name: 51*, 54*, 88*, 90*, 94*.
false: 34, 37, 38, 44, 47, 48, 57*, 58, 67*, 68, 77, 82, 86, 88*, 94*, 96, 97.
filnam: 87, 88*
find_letters: 61, 94*
first_fit: 35, 41, 42, 45, 59.
first_text_char: 12*, 18, 24.
firstc_fit: 45, 49, 50.
flag: 99*, 100*, 101*.
found: 8, 19, 35, 36, 45, 46, 76, 90*, 92.
found_huf: 22, 23, 24, 57*, 75, 81, 84, 85.
fpos: 49, 77, 80, 83.
getASCII: 19, 58.
get_input: 3*.
get_input_in: 3*, 97.
get_letter: 60, 76, 92.
getc: 3*.
getopt: 98*
getopt_long_only: 98*
getopt_return_val: 98*
g getopt_struct: 98*
Largest hyphenation value: 94

last_ASCII_code: 12* 13, 14, 17, 18, 19, 20, 24,
29, 34, 37, 44, 47, 54*, 58, 61.
last_text_char: 12* 18, 24.
left_hyphen_min: 54*, 55, 56, 57* 79.
letter_class: 22, 56, 58, 76, 90*, 92.
level_pattern_count: 66, 67*, 90*, 94*.
Liang, Franklin Mark: 2.
Line too long: 52.
long_options: 98* 99*, 100*, 101*.
loop: 9.
lower: 54* 58, 59.
max_buf_len: 27* 28, 52, 53, 58, 60, 76, 92.
max_dot: 27* 29, 57*, 58, 94* 96.
max_len: 27* 28, 29, 76.
max_ops: 27* 28, 29, 34, 39, 71.
max_pat: 90*, 91*, 92, 94*.
max_val: 27* 28, 29, 65, 69, 71, 80, 92, 94*.
min_packed: 13, 14, 26, 36, 37, 38, 42, 46, 47,
48, 50, 70.
miss_count: 66, 81, 88*.
more_this_level: 95, 96.
my_name: 17, 91*, 98*.
n: 39, 54*, 68,
n_options: 98*.
name: 98*, 99*, 100*, 101*.
new_TRIE_OP: 39, 41.
nil: 9.
no_hyf: 22, 75, 76, 81, 82, 84, 85.
no_more: 74, 77, 80, 86.
not_found: 8, 35, 36, 45, 46.
num_ASCII_codes: 19, 34, 37, 44, 47, 58.
n1: 57*, 94*, 95.
n2: 57*, 94*, 95.
n3: 94*, 95.
odd: 81, 85.
old_OP_count: 71.
old_TRIE_COUNT: 71.
op: 29, 32.
op_count: 33, 34, 39, 67*, 71, 90*.
op_type: 29, 30, 33, 34, 39, 68, 71, 72, 77.
op_word: 29, 30.
ops: 30, 32.
option: 98*.
option_index: 98*.
ord: 16, 18.
output: 3*, 67*.
output_hyphenated_word: 82, 89.
output_patterns: 72, 94*.
overflow: 10* 19, 37, 39, 47, 58, 62, 76.
packed_ASCII_code: 13, 14, 20.
packed_internal_code: 20, 30.
PATGEN changes for C

58 INDEX

parse_arguments: 3*, 98*
pat: 40, 41, 42, 58, 59, 61, 62, 64, 72, 73, 92, 93.
pat_count: 43, 44, 49, 88*
pat_dot: 65, 80, 83, 85, 87, 88, 96.
pat_finish: 4, 94*, 96.
pat_len: 40, 41, 49, 58, 59, 64, 72, 73, 80, 83, 85, 87, 88, 92, 93, 94, 95, 96.
pat_start: 4, 94*, 96.

PATGEN: 3*

PATGEN capacity exceeded ...: 10*

PATGEN_HELP: 98*
patout: 3*, 21, 51*, 73, 94*

patterns: 3*, 21, 22, 51*, 52, 90*
pattmp: 21, 51*, 82, 87, 88*


print_buf: 53, 58, 76.

printLn: 3*, 53, 54*, 57, 67, 71, 88, 90, 94*.

printReal: 67*, 88*

printVersion_and_exit: 98*

procesp: 85, 87, 88*, 89, 94*, 97.

q: 35, 45.
q_back: 32, 45, 48, 49, 50.
q_index: 29, 31, 35, 45.
q_link: 32, 35, 38, 41, 42, 45, 48, 49, 50, 59.
qmax_threshold: 31, 34, 36, 71.

range check violations: 2, 14, 83.

read: 52.

read_buf: 52, 54*, 58, 76, 90*.

readLn: 3*, 52.

read_patterns: 60, 90*, 94*.

print_translate: 54*, 94*.

print_word: 60, 76, 89.

real: 51*.

reset: 54*, 88*, 90*.

return: 9.

rewrites: 88*, 94*.

right_hyphen_min: 54*, 55, 56, 57*, 79.

s: 35, 38, 41, 54*, 68, 72.
si: 13, 14, 34, 35, 42, 44, 45, 50, 62.
so: 13, 14, 38, 41, 48, 49, 59, 60, 61, 63, 64, 68, 72, 77, 83.

space_class: 22, 24, 76, 92.

spso: 49, 50, 77, 80, 83.

std_input: 3*.

stderr: 10*, 98*.

stdin: 3*.

strcpy: 98*.

system_dependencies: 3*, 10*, 11, 12*, 13, 18, 51*, 52.

t: 35, 38, 41, 54*, 68, 71, 72, 76, 77, 90*.

tab_char: 18*.

text: 12*.

text_char: 12*, 15, 16, 19, 23, 52, 54*, 76, 90*.

text_file: 12*, 51*.

thresh: 4, 65, 67*, 94*.

translate: 2, 3*, 6, 12*, 19, 21, 51*, 52, 54*, 57*, 58.

traverse_count_trie: 64, 67*.

trie_back: 26, 32, 34, 35, 36, 37, 38, 42, 70.

trie_base_used: 26, 32, 34, 35, 36, 37, 38, 68.

trie_bmax: 33, 34, 37.

trie_c: 30, 32.

trie_char: 26, 32, 34, 35, 36, 37, 38, 41, 42, 59, 60, 61, 68, 70, 72, 77.

trie_count: 33, 34, 41, 42, 59, 67*, 70, 71, 90*.

trie_i: 30, 32.

trie_link: 26, 32, 34, 35, 36, 37, 38, 41, 42, 59, 60, 61, 68, 70, 72, 77, 94*.

trie_max: 33, 34, 35, 36, 42, 67*, 70, 90*.

trie_outp: 26, 32, 34, 35, 38, 41, 42, 59, 60, 61, 62, 68, 69, 72, 77.

trie_pointer: 29, 30, 31, 33, 35, 38, 41, 54*, 61, 68, 71, 72, 76, 77, 90*.

trie_r: 30, 32.

trie_root: 26, 34, 41, 59, 60, 68, 71, 77, 94*.


trie_taken: 30, 32.

trie_back: 32, 44, 45, 46, 47, 48, 50.

trie_bad: 32, 43, 65, 83.

trie_base_used: 32, 44, 45, 46, 47, 48.

trie_bmax: 43, 44, 47.

trie_c: 30, 32.

trie_char: 32, 44, 45, 46, 47, 48, 49, 50, 62, 63, 64, 83.

trie_count: 43, 44, 49, 50, 88*.

trie_good: 32, 43, 65, 83.

trie_kmax: 43, 44, 47.

trie_l: 30, 32.

trie_link: 32, 44, 45, 46, 47, 48, 49, 50, 62, 63, 64, 83.

trie_max: 43, 44, 45, 46, 50, 62, 83, 88*.

trie_pointer: 29, 30, 43, 45, 48, 49, 61, 64, 72, 82, 83.

trie_r: 30, 32.

trie_root: 44, 49, 62, 63, 67*, 83.


trie_taken: 30, 32.

trie_c: 31, 32, 35, 38.

trie_l: 31, 32, 35, 38.

trie_r: 31, 32, 35, 38.

true: 9, 34, 35, 44, 45, 57*, 58, 59, 65, 68, 76, 77, 80, 86, 87, 94*, 96, 97.
uexit: 10*
unpack: 38, 42, 48.
unpackc: 48, 50.
usage: 98*
usage help: 98*
v: 39, 77.
val: 29, 32, 41, 99*, 100*, 101*
val_type: 4, 29, 39, 41, 74, 77, 87, 91, 95.
version_string: 3*
wlen: 74, 76, 77, 78, 81, 82, 83, 89.
word: 29, 49, 50, 74, 76, 77, 78, 82, 83.
word_index: 29, 49, 74, 77, 78, 81, 82, 83, 84.
word_wt: 74, 76, 82, 88*
write: 37, 63, 73, 82.
write_letter: 63, 72, 73, 82.
write_letter_end: 63.
write ln: 3*, 10*, 67*, 73, 82, 98*
w t_chg: 74, 76, 82, 88*
xchr: 16, 17, 18, 19, 24, 56, 58, 63.
xclass: 22, 23, 24, 56, 57*, 58, 59, 75, 76, 90*, 92.
xclause: 9.
xdig: 23, 24, 73, 82, 88*
xdot: 23.
xxect: 23, 24, 56, 58, 62, 63, 73, 82.
xfclose: 51*
xhyf: 23, 24, 57*, 75, 82.
xxint: 22, 23, 24, 56, 57*, 58, 75, 76, 90*, 92.
xord: 16, 18, 19, 60.
(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*.
(Constats 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.
(Define the option table 99*, 100*, 101*) Used in section 98*.
(Define parse_arguments 98*) Used in section 3*.
(Ensure trie linked up to $b + \text{num\_ASCII\_codes}$ 47) Used in section 46.
(Ensure trie linked up to $s + \text{num\_ASCII\_codes}$ 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 $\text{no\_more}$ arrays, and advance $h$ 80) Used in section 77.
(Types in the outer block 12*, 13, 20, 22, 29) Used in section 3*.