PATtern GENeration program
for the TeX82 hyphenator

(Version 2.4, April 2020)

Editor's Note: The present variant of this C/WEB source file has been modified for use in the TeX Live system.
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.

July 9, 2024 at 19:33

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 $my\_name \equiv$ ´patgen´
define $banner \equiv$ ´This␣is␣PATGEN,␣Version␣2.4´   { printed when the program starts }

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 $print(\#) \equiv write(output, \#)$
define $print\_ln(\#) \equiv write\_ln(output, \#)$
define $get\_input(\#) \equiv \# \leftarrow input\_int(std\_input)$
define $get\_input\_ln(\#) \equiv$
         begin $\# \leftarrow getc(std\_input)$; $read\_ln(std\_input)$;
         end
define $std\_input \equiv stdin$

⟨ Compiler directives 11 ⟩
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;

10*   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.

define $error(\#) \equiv$
         begin $write\_ln(stderr, \#)$; $uexit(1)$;
         end;
define $overflow(\#) \equiv error($´PATGEN␣capacity␣exceeded,␣sorry␣[´, #, ´].´$)$

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*.

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* ⟩ ≡

$trie\_size = 10000000$;   { space for pattern trie }

$triec\_size = 5000000$;   { space for pattern count trie, must be less than $trie\_size$ 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 max_len }

This code is used in section 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.

> **define** *close_out*(#) ≡ *xfclose*(#, ´outputfile´)   { close an output file }
> **define** *close_in*(#) ≡ *xfclose*(#, ´inputfile´)   { close an input file }

⟨ Globals in the outer block 4 ⟩ +≡
dictionary, *patterns*, *translate*, *patout*, *pattmp*: *text_file*;
f_name: ↑*char*;
bad_frac, *denom*, *eff*: *real*;

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*; *f_name* ← *cmdline*(4); *reset*(*translate*, *f_name*);
  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;

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* ⟩ ≡

  $bad \leftarrow false$; $n \leftarrow 0$;

  if $buf[1] = \text{´}_\sqcup\text{´}$ **then** $do\_nothing$

  else if $xclass[buf[1]] = digit\_class$ **then** $n \leftarrow xint[buf[1]]$ **else** $bad \leftarrow true$;

  if $xclass[buf[2]] = digit\_class$ **then** $n \leftarrow 10 * n + xint[buf[2]]$ **else** $bad \leftarrow true$;

  if $(n \geq 1) \wedge (n < max\_dot)$ **then** $left\_hyphen\_min \leftarrow n$ **else** $bad \leftarrow true$;

  $n \leftarrow 0$;

  if $buf[3] = \text{´}_\sqcup\text{´}$ **then** $do\_nothing$

  else if $xclass[buf[3]] = digit\_class$ **then** $n \leftarrow xint[buf[3]]$ **else** $bad \leftarrow true$;

  if $xclass[buf[4]] = digit\_class$ **then** $n \leftarrow 10 * n + xint[buf[4]]$ **else** $bad \leftarrow true$;

  if $(n \geq 1) \wedge (n < max\_dot)$ **then** $right\_hyphen\_min \leftarrow n$ **else** $bad \leftarrow true$;

  if bad **then**

    begin $bad \leftarrow false$;

    repeat $print(\text{´left\_hyphen\_min,}_\sqcup\text{right\_hyphen\_min:}_\sqcup\text{´})$; $input\_2ints(n1, n2)$;

      if $(n1 \geq 1) \wedge (n1 < max\_dot) \wedge (n2 \geq 1) \wedge (n2 < max\_dot)$ **then**

        begin $left\_hyphen\_min \leftarrow n1$; $right\_hyphen\_min \leftarrow n2$;

        end

      else begin $n1 \leftarrow 0$;

        $print\_ln(\text{´Specify}_\sqcup\text{1<=left\_hyphen\_min,right\_hyphen\_min<=´}, max\_dot - 1 : 1, \text{´}_\sqcup\text{!´})$;

        end;

    until $n1 > 0$;

    end;

  for $j \leftarrow err\_hyf$ **to** $found\_hyf$ **do**

    begin if $buf[j + 4] \neq \text{´}_\sqcup\text{´}$ **then** $xhyf[j] \leftarrow buf[j + 4]$;

    if $xclass[xhyf[j]] = invalid\_class$ **then** $xclass[xhyf[j]] \leftarrow hyf\_class$ **else** $bad \leftarrow true$;

    end;

  $xclass[\text{´.´}] \leftarrow hyf\_class$;   { in case the default has been changed }

  if bad **then** $bad\_input(\text{´Bad}_\sqcup\text{hyphenation}_\sqcup\text{data´})$

This code is used in section 54*.

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**
     begin *print*(´,␣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, ´␣nodes,␣´,
      ´trie_max␣=␣´, *trie_max* : 1, ´,␣´, *op_count* : 1, ´␣outputs´);
   end;

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*;
  f_name ← *cmdline*(1); *reset*(*dictionary*, *f_name*); ⟨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 *strcpy*(*filnam*, ´pattmp.␣´); *filnam*[7] ← *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**
    begin *print_real*((100 ∗ (*good_count*/(*good_count* + *miss_count*))), 1, 2); *print*(´␣%,␣´);
    print_real((100 ∗ (*bad_count*/(*good_count* + *miss_count*))), 1, 2); *print*(´␣%,␣´);
    print_real((100 ∗ (*miss_count*/(*good_count* + *miss_count*))), 1, 2); *print_ln*(´␣%´);
    end;
  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;

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* 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;

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:␣´); *input_2ints*(*n1*, *n2*);
  if $(n1 \geq 1) \wedge (n1 < max\_val) \wedge (n2 \geq 1) \wedge (n2 < max\_val)$ **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:␣´); *input_2ints*(*n1*, *n2*);
    if $(n1 \geq 1) \wedge (n1 \leq n2) \wedge (n2 \leq max\_dot)$ **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:␣´); *input_3ints*(*n1*, *n2*, *n3*);
    if $(n1 \geq 1) \wedge (n2 \geq 1) \wedge (n3 \geq 1)$ **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 }
f_name ← *cmdline*(3); *rewrite*(*patout*, *f_name*); *output_patterns*(*trie_root*, 1); *close_out*(*patout*);
⟨ Make final pass to hyphenate word list 97 ⟩;
end.

98* System-dependent changes.   Parse a Unix-style command line.

> **define** $argument\_is(\#) \equiv (strcmp(long\_options[option\_index].name, \#) = 0)$

⟨ 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 $getopt\_return\_val \leftarrow getopt\_long\_only(argc, argv, \text{´´}, long\_options, address\_of(option\_index))$;
    if $getopt\_return\_val = -1$ **then**
      begin $do\_nothing$;
      end
    else if $getopt\_return\_val = \text{´?´}$ **then**
      begin $usage(my\_name)$;
      end
    else if $argument\_is(\text{´help´})$ **then**
      begin $usage\_help(PATGEN\_HELP, \textbf{nil})$;
      end
    else if $argument\_is(\text{´version´})$ **then**
      begin
        $print\_version\_and\_exit(banner, \textbf{nil}, \text{´Frank}_{\sqcup}\text{M.}_{\sqcup}\text{Liang}_{\sqcup}\text{and}_{\sqcup}\text{Peter}_{\sqcup}\text{Breitenlohner´}, \textbf{nil})$;
      end;   { Else it was just a flag; $getopt$ has already done the assignment. }
  until $getopt\_return\_val = -1$;   { Now $optind$ is the index of first non-option on the command line. }
  if $(optind + 4 \neq argc)$ **then**
    begin $write\_ln(stderr, my\_name, \text{´:}_{\sqcup}\text{Need}_{\sqcup}\text{exactly}_{\sqcup}\text{four}_{\sqcup}\text{arguments.´})$; $usage(my\_name)$;
    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.

⟨ Define the option table 99* ⟩ ≡
  $current\_option \leftarrow 0$; $long\_options[current\_option].name \leftarrow \text{´help´}$;
  $long\_options[current\_option].has\_arg \leftarrow 0$; $long\_options[current\_option].flag \leftarrow 0$;
  $long\_options[current\_option].val \leftarrow 0$; $incr(current\_option)$;
See also sections 100* and 101*.

This code is used in section 98*.

100*   Another of the standard options.

⟨ Define the option table 99* ⟩ +≡
  $long\_options[current\_option].name \leftarrow \text{´version´}$; $long\_options[current\_option].has\_arg \leftarrow 0$;
  $long\_options[current\_option].flag \leftarrow 0$; $long\_options[current\_option].val \leftarrow 0$; $incr(current\_option)$;

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

⟨ Define the option table 99* ⟩ +≡
  $long\_options[current\_option].name \leftarrow 0$; $long\_options[current\_option].has\_arg \leftarrow 0$;
  $long\_options[current\_option].flag \leftarrow 0$; $long\_options[current\_option].val \leftarrow 0$;

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.
argc: 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_wt: 4, 65, 94.*
banner: 1,* 3,* 98.*
boolean: 30, 54,* 66, 68, 74, 83, 87, 95.
Breitenlohner, 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, 39, 70, 71.
Decr: 9.

delete_bad_patterns: 36, 71, 94.*
delete_patterns: 68, 70, 71.
denom: 51.*
dictionary: 3,* 21, 22, 51,* 52, 75, 76, 88,* 89.
digit: 22, 74, 90.*
digit_class: 22, 24, 57,* 76, 92.
do_dictionary: 87, 88,* 96, 97.
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,* 55, 61, 64, 72, 87, 90,* 95.
dots: 74, 76, 81, 82, 86.
dotw: 74, 76, 81, 82.
dot1: 95, 96.
dpos: 76, 77, 80, 81, 82, 83, 86.
edge_of_word: 20, 24, 25, 54,* 76, 90,* 93.
eff: 51.*
eof: 54,* 89, 90.*
eoln: 52.
err_hyf: 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_hyf: 22, 23, 24, 57,* 75, 81, 84, 85.
fpos: 49, 77, 80, 83.
get_ASCII: 19, 58.
get_input: 3.*
get_input_ln: 3,* 97.
get_letter: 60, 76, 92.
getc: 3.*
getopt: 98.*
getopt_long_only: 98.*
getopt_return_val: 98.*
getopt_struct: 98.*