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 my_name ≡ 'patgen'
define banner ≡ '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(#) ≡ write(output,#)
define print_ln(#) ≡ write_ln(output,#)
define get_input(#) ≡ # ← input_int(std_input)
define get_input_ln(#) ≡
        begin # ← getc(std_input); read_ln(std_input);
        end
define std_input ≡ 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(#) ≡
        begin write_ln(stderr,#); uexit(1);
        end;
define overflow(#) ≡ 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 ← 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´)
```

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 ≥ 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:␣'); 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; 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 ≥ 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 }
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(#) ≡ (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;  { 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 ≠ argc) then
    begin write_ln(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.

```
⟨ 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.

```
⟨ 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.

```
⟨ 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.