1. Introduction. This is `utfpatgen` – reimplementation of the classic `patgen` program for pattern generation. With `utfpatgen`, we intend to overcome several limitations of the original, such as the number of hyphenation levels possible, inability to use some reserved characters in dictionary, and most importantly, we enable native usage of the UTF-8 encoding in dictionaries that are no longer limited by the fixed number of lowercase characters permitted by `patgen`.

We provide `utfpatgen` open-source and free of charge under MIT license. Please note that there is no warranty and despite our greatest effort, the program may contain bugs.

2. Terminology. Before diving into the implementation part of the program, it is useful to mention several terms occurring frequently throughout the text. It may come useful especially to those who are not thoroughly familiar with `patgen`. The definitions here are mostly informal and intended to ease reader's understanding of the topic.

Hyphenation is a process of splitting words so that they fit better to the paragraphs when typeset. It follows the linguistic rules set for the language in which the text is written.

Pattern is a sequence of characters containing hyphenation information tied to some position (**dot position**) on the edges of the pattern or in between the characters. Whether this information tells that the position should or should not be hyphenated, we distinguish between **hyphenating** and **inhibiting patterns**.

Hyphen (in context of `patgen` and `utfpatgen`) is a mark between two characters holding the information whether that position should be split during hyphenation and whether this split was detected by current patterns. There are therefore 4 types of hyphens: **NO_HYF** (not in the data, not marked), **MISS_HYF** (in the data, not marked), **BAD_HYF** (not in the data, marked), and **GOOD_HYF** (in the data, marked). Strictly speaking, there is a hyphen between each pair of neighboring characters in the data, but NO_HYF marking is omitted implicitly.

Supporting occurrences of a pattern in the data are those that are in favor of the hyphenation information it holds. For hyphenating patterns, these are the cases where the word has a MISS_HYF on the dot position pointed by the pattern. For inhibiting patterns, dot position with a BAD_HYF is a supporting one. **Contradicting occurrences** are those that would break a correct hyphen, thus NO_HYF for hyphenating, and GOOD_HYF for inhibiting patterns. Note that an occurrence of a pattern can be neither supporting nor contradicting.

Hyphenation level marks the strength of a hyphenation. It is represented by a non-negative integer and creates a hierarchy of patterns. Pattern with higher level always takes precedence over any pattern with lower level. At odd levels, the patterns are hyphenating, at even levels inhibiting.

Trie is a data structure for storing n-ary trees. It can implemented using an associated array with highly effective insertion, deletion and lookup. Furthermore, we can condense it via *packing* to save space.

3. Algorithm overview. In general, we tried to adhere as tightly as possible to the original ideas of `patgen`. Therefore, you may find the names and functions similar to those in the `patgen` technical report. We have nevertheless decided to rewrite several parts of the algorithm in more "modern" way to improve its readability and testability. The main points to mention are:

- the algorithm is implemented in CWEB (C being the language of the program), not WEB (with Pascal),
- instead of statically defining the sizes of structures (tries, buffers, etc.), these are allocated and reallocated dynamically, allowing for greater flexibility and possibly space savings,
- global variables are localized as much as possible,
- `goto` statements are eliminated.

You may also find a few unit tests appended to the code. These are by no means exhaustive, but feel free to run them and add your own.

We decided to present `utfpatgen` in top-down fashion – starting with the full overview and moving to details in later sections. Same goes for the structures which have their own dedicated sections.

4. Dependencies. All external libraries used in `utfpatgen` come from the standard C package, so we hope it to be widely portable without greater trouble. All in all, we use fixed-size types from `<stdint.h>` and `<stdbool.h>`, IO support from `<stdio.h>`, string manipulation methods from `<string.h>`, and memory management provided by `<stdlib.h>`.

⟨ Library includes 4 ⟩ ≡
```
#include "utfpatgen.h"
#include <string.h>
```
This code is used in section 5.

5. Main method. The general flow of the program is rather simple: initialize structures, get the parameters, generate patterns, and optionally hyphenate the dictionary. For sure, it gets more complicated the deeper we dive.

```
⟨ Library includes 4 ⟩
#ifndef TEST
  int main(int argc, char *argv[])
  {
    for (int i = 0; i < argc; i++) {
      if (strcmp(argv[i], "--help") ≡ 0) {
        print_help();
        return EXIT_SUCCESS;
      }
      else if (strcmp(argv[i], "--version") ≡ 0) {
        print_version();
        return EXIT_SUCCESS;
      }
    }
    print_version();
    ⟨ Initialization sequence 6 ⟩;
    ⟨ Level range specification 7 ⟩;
    ⟨ Pattern generation 8 ⟩;
    ⟨ Final pass 11 ⟩;
    destroy_params(params);
    destroy_tr_table(tt);
    destroy_pattern_trie(pt);
    return EXIT_SUCCESS;
  }
#endif
```

6. Initialization sequence. First, the input parameters provided to the program call are read, validated and processed. Unless asking for help (`--help`) or version printout (`--version`), `utfpatgen` takes exactly 4 inputs, representing 4 files:

- **Dictionary file**: contains set of hyphenated words.
- **Patterns file**: stores patterns generated in previous runs.
- **Output file**: where the patterns will be stored after the run.
- **Translate file**: contains the mapping of characters from the dictionary and dictionary-specific parameters.

The required formats of these files are the same as for the `patgen` program and are discussed in dedicated sections.

⟨ Initialization sequence 6 ⟩ ≡

```
struct params *params = init_params( );
if (params ≡ Λ) {
  return EXIT_FAILURE;
}
if (¬parse_input(argv, argc, params)) {
  destroy_params(params);
  return EXIT_FAILURE;
}
struct translate_table *tt = init_tr_table(256, 128);
if (tt ≡ Λ) {
  destroy_params(params);
  return EXIT_FAILURE;
}
if (¬read_translate(params, tt)) {
  destroy_params(params);
  destroy_tr_table(tt);
  return EXIT_FAILURE;
}
struct pattern_trie *pt = init_pattern_trie(256, 128);
if (pt ≡ Λ) {
  destroy_params(params);
  destroy_tr_table(tt);
  return EXIT_FAILURE;
}
struct pass_stats ps;
if (¬read_patterns(params, pt, tt, &ps)) {
  destroy_params(params);
  destroy_tr_table(tt);
  destroy_pattern_trie(pt);
  return EXIT_FAILURE;
}
```

This code is used in section 5.

7. Level range specification. Besides the command line parameters, the program prompts for the hyperparameters of the algorithm. The `hyph_start` and `hyph_finish` specify the range of hyphenation levels covered during the run. The program can generate patterns up to level 255.

⟨ Level range specification 7 ⟩ ≡

```
printf("hyph_start␣(lowest␣-),␣hyph_finish␣(highest␣hyphenation␣level):␣");
size_t hyph_start, hyph_finish;
int result;
while (true) {
  result = scanf("%zu␣%zu", &hyph_start, &hyph_finish);
  if (result ≡ 2 ∧ hyph_start ≥ 1 ∧ hyph_start ≤ 255 ∧ hyph_finish ≥ 1 ∧ hyph_finish ≤ 254) {
    break;
  }
  else {
    printf("Error:␣Specify␣1␣<=␣hyph_start,␣hyph_finish␣<=␣255!␣Insert␣again:␣");
    while (getchar( ) ≠ '\n') ;
  }
}
params¬hyph_start = (uint8_t)hyph_start;
if (hyph_start > hyph_finish) {
  params¬hyph_finish = ps.max_level;
  printf("Warning:␣hyph_start␣>␣hyph_finish,␣using␣hyph_finish␣=␣%u\n", ps.max_level);
}
else {
  params¬hyph_finish = (uint8_t)hyph_finish;
}
```

This code is used in section 5.

8. Pattern generation loop. The algorithm runs for each level within the specified range, going from `hyph_start` up. The program prompts for level-specific hyperparameters, generates and prunes the patterns.

```
⟨ Pattern generation 8 ⟩ ≡
  size_t pat_start, pat_finish, good_wt, bad_wt, thresh;
  for (size_t i = params→hyph_start; i ≤ params→hyph_finish; i++) {
    params→hyph_level = i;
    ps.level_pattern_cnt = 0;
    if (params→hyph_level > params→hyph_start) {
      printf("\n");
    }
    if (params→hyph_start ≤ ps.max_level) {
      printf("Warning: Largest hyphenation value %u in patterns should be\
          less than hyph_start\n", ps.max_level);
    }
    ⟨ Level hyperparameters input 9 ⟩;
    ⟨ Level generation 10 ⟩;
    if (¬delete_bad_patterns(pt)) {
      destroy_params(params);
      destroy_tr_table(tt);
      destroy_pattern_trie(pt);
      return EXIT_FAILURE;
    }
    printf("total of %zu patterns at hyph_level %u\n", ps.level_pattern_cnt, params→hyph_level);
  }
```

This code is used in section 5.

9. Level hyperparameters input. The program again prompts for hyperparameter input. Firstly, it asks for `pat_start` and `pat_finish` that define the length range of patterns for respective level. The maximum length of a pattern in `utfpatgen` is set to 255. Subsequently, the user is prompted to insert the three weights `good_wt`, `bad_wt` and `thresh`. These define the acceptance criteria for candidate patterns – in order to accept the pattern during the ongoing iteration, its number of *good* (supporting) and *bad* (contradicting) occurrences must make the following inequality to hold: $good{*}good_wt - bad{*}bad_wt \geq thresh$ The maximum value for the weights and threshold in `utfpatgen` is set to 255.

```
⟨ Level hyperparameters input 9 ⟩ ≡
  printf("pat_start␣(shortest␣-),␣pat_finish␣(longest␣pattern␣explored):␣");
  while (true) {
    result = scanf("%zu␣%zu", &pat_start, &pat_finish);
    if (result ≡ 2 ∧ pat_start ≥ 1 ∧ pat_finish ≥ 1 ∧ pat_start ≤ pat_finish ∧ pat_start ≤ 255 ∧ pat_finish ≤ 255)
      {
      break;
    }
    else {
      printf("Error:␣Specify␣1␣<=␣pat_start␣<=␣pat_finish␣<=␣255!␣Insert␣again:␣");
      while (getchar( ) ≠ '\n') ;
    }
  }
  params→pat_start = (uint8_t)pat_start;
  params→pat_finish = (uint8_t)pat_finish;
  printf("good_wt␣(good␣-),␣bad_wt␣(bad␣pattern␣weight),␣threshold:␣");
  while (true) {
    result = scanf("%zu␣%zu␣%zu", &good_wt, &bad_wt, &thresh);
    if (result ≡ 3 ∧ good_wt ≥ 1 ∧ bad_wt ≥ 1 ∧ thresh ≥ 1 ∧ good_wt ≤ 255 ∧ bad_wt ≤ 255 ∧ thresh ≤ 255) {
      break;
    }
    else {
      printf("Error:␣Specify␣1␣<=␣good_wt,␣bad_wt,␣threshold␣<=␣255!␣Insert␣again:␣");
      while (getchar( ) ≠ '\n') ;
    }
  }
  params→good_wt = (uint8_t)good_wt;
  params→bad_wt = (uint8_t)bad_wt;
  params→thresh = (uint8_t)thresh;
```

This code is used in section 8.

10. Level generation. The single pass of `utfpatgen` at given hyphenation level comprises iterating through pattern lengths (`pat_len` parameter, in ascending order) and dot positions (`pat_dot`, from the middle toward the edges) and processing the dictionary. The algorithm collects supporting and contradicting occurrences and eventually adds new patterns to the set. It can happen that an iteration is skipped if it is known in advance that it will not yield any new patterns.

```
⟨ Level generation 10 ⟩ ≡
  uint8_t aux_dot;
  bool more_this_level[256];
  for (size_t i = 0; i < 256; i++) {
    more_this_level[i] = true;
  }
  for (size_t j = params→pat_start; j ≤ params→pat_finish; j++) {
    params→pat_len = j;
    params→pat_dot = params→pat_len/2;
    aux_dot = params→pat_dot * 2;
    while (params→pat_dot ≠ params→pat_len) {
      params→pat_dot = aux_dot − params→pat_dot;
      aux_dot = params→pat_len * 2 − aux_dot − 1;
      if (more_this_level[params→pat_dot]) {
        if (¬process_dictionary(params, tt, pt, &ps)) {
          destroy_params(params);
          destroy_tr_table(tt);
          destroy_pattern_trie(pt);
          return EXIT_FAILURE;
        }
        more_this_level[params→pat_dot] = ps.more_to_come;
      }
    }
    for (size_t i = 255; i > 0; i−−) {
      if (¬more_this_level[i − 1]) {
        more_this_level[i] = false;
      }
    }
  }
```

This code is used in section 8.

11. Final pass. If the user wishes, the dictionary is traversed one last time and hyphenated according to found patterns. The output is stored in file 'pattmp.X'.

```
⟨ Final pass 11 ⟩ ≡
  if (¬output_patterns(pt, tt, params→output_file)) {
    destroy_params(params);
    destroy_tr_table(tt);
    destroy_pattern_trie(pt);
    return EXIT_FAILURE;
  }
  char c;
  bool output = false;
  printf("hyphenate␣word␣list?␣(y/n):␣");
  if (scanf("␣%c", &c) < 1) {
    destroy_params(params);
    destroy_tr_table(tt);
    destroy_pattern_trie(pt);
    return EXIT_FAILURE;
  }
  if (c ≡ 'y' ∨ c ≡ 'Y') {
    output = true;
  }
  if (¬hyphenate_dictionary(params, tt, pt, output, &ps)) {
    destroy_params(params);
    destroy_tr_table(tt);
    destroy_pattern_trie(pt);
    return EXIT_FAILURE;
  }
```

This code is used in section 5.

12. Implementation details. In this section we focus on the building blocks, each subsection devoted to one particular aspect of the algorithm.

13. IO procedures. Methods in this subsection stand on the interface between `utfpatgen` and the user, together with the `main` method. `parse_inputs` attempts to open the 4 files provided as inputs into streams and save them for later use. `read_line` is a utility to simplify reading from these streams that stores the information read into provided buffer. The `print_help` and `print_version` methods print out the desired information if the user does not wish to proceed to pattern generation.

14. parse_input. Attempts to open the 4 files provided as inputs into streams and save them for later use. Returns a boolean indicating whether all files have been opened successfully.

```
bool parse_input(char *argv[], int argc, struct params *params)
{
  if (argc ≠ 5) {
    fprintf(stderr, "utfpatgen␣need␣exactly␣4␣arguments.\nTry␣‘utfpatgen␣--help‘\
        ␣for␣more␣information.\n");
    return false;
  }
  FILE *dictionary_file = fopen(argv[1], "rb");
  if (dictionary_file ≡ Λ) {
    fprintf(stderr, "Could␣not␣open␣dictionary␣file␣’%s’.\n", argv[1]);
    return false;
  }
  FILE *pattern_file = fopen(argv[2], "rb");
  if (pattern_file ≡ Λ) {
    fprintf(stderr, "Could␣not␣open␣pattern␣file␣’%s’.\n", argv[2]);
    fclose(dictionary_file);
    return false;
  }
  FILE *output_file = fopen(argv[3], "wb");
  if (output_file ≡ Λ) {
    fprintf(stderr, "Could␣not␣open␣output␣file␣’%s’.\n", argv[3]);
    fclose(dictionary_file);
    fclose(pattern_file);
    return false;
  }
  FILE *translate_file = fopen(argv[4], "rb");
  if (translate_file ≡ Λ) {
    fprintf(stderr, "Could␣not␣open␣translate␣file␣’%s’.\n", argv[4]);
    fclose(dictionary_file);
    fclose(pattern_file);
    fclose(output_file);
    return false;
  }
  params→dictionary_file = dictionary_file;
  params→pattern_file = pattern_file;
  params→output_file = output_file;
  params→translate_file = translate_file;
  return true;
}
```

15. read_line. Reads a line from given stream into a string buffer. If end of file is reached, it sets `eof` flag of the buffer. Returns a boolean indicating whether the line has been read successfully.

```
bool read_line(FILE *stream, struct string_buffer *buf)
{
  reset_buffer(buf);
  char c;
  while ((c = fgetc(stream)) ≠ EOF) {
    if (buf→size ≥ buf→capacity − 1) {
      if (resize_buffer(buf, 2 * buf→capacity) ≡ Λ) {
        return false;
      }
    }
    if (c ≡ '\n') {
      if ((buf→size > 0) ∧ (buf→data[buf→size − 1] ≡ '\r')) {      /* Windows /r/n end of line */
        buf→size −= 1;
      }
      break;
    }
    buf→data[buf→size] = c;
    buf→size ++;
  }
  buf→data[buf→size] = '\0';
  if (c ≡ EOF) {
    buf→eof = true;
  }
  return true;
}
```

16. print_help. Prints out a help message to the standard output.

```
void print_help( )
{
  printf("Usage: utfpatgen [OPTION]... DICTIONARY PATTERNS OUTPUT TRANSLATE\n");
  printf("\tGenerate the OUTPUT hyphenation file for use with TeX\n");
  printf("\tfrom the DICTIONARY, PATTERNS, and TRANSLATE files.\n");
  printf("\n--help         print this help and exit\n");
  printf("--version      output version information and exit\n");
  printf("\nutfpatgen home page: https://ctan.org/pkg/utfpatgen\n");
}
```

17. print_version. Prints out the version number of `utfpatgen` to the standard output.

```
void print_version( )
{
  printf("This is utfpatgen version %s\n", UTFPATGEN_VERSION);
  printf("\tCopyright 2026 Ondej Metelka\n");
  printf("\tLicense MIT: https://mit-license.org/\n");
  printf("\nThis is free software: you are free to change and redistribute it.\n");
  printf("\tThere is NO WARRANTY, to the extent permitted by law.\n");
}
```

18. Translate file processing. Once the translate file has been successfully opened and its stream pointer stored, it is read line by line into translate table. The purpose of the translate file is to list all the characters that occur in the dictionary, together with their uppercase variants. Later during the pattern generation, all of these variants are treated as though they were the same character. Besides, the first line of the file may redefine the language-specific parameters `lefthyphenmin`, `righthyphenmin`, and the symbols used for marking hyphens in the dictionary. The format of the translate file follows the same pattern as required by `patgen`:

- first line (optional): 'LLRR BMG', where 'LL' is the value of `lefthyphenmin`, 'RR' the value of `righthyphenmin`, 'B' the symbol for `BAD_HYF`, 'M' the symbol for `MISS_HYF`, and 'G' the symbol for `GOOD_HYF`. Should any of these parameters stay blank, the default is used: `lefthyphenmin`= 2, `righthyphenmin`= 3, `BAD_HYF` '.', `MISS_HYF` '-', `GOOD_HYF` '*'.
- consequent lines: '$_X_Y_1_...Y_n__$', where 'X' is a lowercase letter, 'Y_k' an arbitrary (even 0) number of upper-case variants of 'X', and '$_$' a delimiter, usually space.

Note that this format allows two-digit values of `lefthyphenmin` and `righthyphenmin` at most and one-byte characters for hyphen symbols. Practically, this is not a problem.

The translate file must be provided as an input to `utfpatgen`, but may be left empty. In that case, the default values for parameters ASCII character mapping are used.

19. read_translate. Reads the contents of the translate file and parses its lines into a translate table. It includes parsing the optional header line and storing the parameters obtained. If the file is empty, the method fetches a default ASCII mapping. Returns a boolean indicating whether the file has been read successfully.

```
bool read_translate(struct params *params, struct translate_table *tt)
{
  rewind(params→translate_file);
  struct string_buffer *buf = init_buffer(64);
  if (buf ≡ Λ) {
    return false;
  }
  struct trie *helper_trie = init_trie(256);
  if (helper_trie ≡ Λ) {
    destroy_buffer(buf);
    return false;
  }
  if (¬read_line(params→translate_file, buf)) {
    destroy_trie(helper_trie);
    destroy_buffer(buf);
    return false;
  }
  if (buf→eof) {
    bool default_mapping = default_ascii_mapping(tt, helper_trie);
    destroy_trie(helper_trie);
    destroy_buffer(buf);
    return default_mapping;
  }
  size_t out_index, letter_index;
  letter_index = tt→letter_count + 1;
  if (letter_index ≥ tt→letter_capacity) {
    size_t new_capacity = tt→letter_capacity * 2;
    size_t *new_array = realloc(tt→index_to_alphabet, new_capacity * sizeof(size_t));
    if (new_array ≡ Λ) {
      return false;
    }
    tt→index_to_alphabet = new_array;
    tt→letter_capacity = new_capacity;
  }
  tt→index_to_alphabet[letter_index] = tt→alphabet→size;
  if (¬insert_pattern(tt→mapping, (const char[]){EDGE_OF_WORD, '\0'}, &out_index,
        helper_trie) ∨ ¬append_string(tt→alphabet, (const char[]){EDGE_OF_WORD, '\0'})) {
    return false;
  }
  tt→mapping→aux[out_index] = letter_index;
  tt→letter_count++;
  bool first_line = true;
  while (¬buf→eof) {
    if (first_line ∧ parse_header(buf, params)) {      /* header parsed successfully */
    }
    else if (¬parse_letters(buf, tt, helper_trie)) {
```

```
        destroy_trie(helper_trie);
        destroy_buffer(buf);
        return false;
      }
      first_line = false;
      reset_buffer(buf);
      if (¬read_line(params→translate_file, buf)) {
        destroy_trie(helper_trie);
        destroy_buffer(buf);
        return false;
      }
    }
    destroy_trie(helper_trie);
    destroy_buffer(buf);
    printf("left_hyphen_min␣=␣%u,␣right_hyphen_min␣=␣%u,␣%zu␣letters\n",
        params→left_hyphen_min, params→right_hyphen_min, tt→letter_count);
    return true;
  }
```

20. is_integer. Returns a boolean indicating whether the given character is an ASCII numeric literal.

```
  bool is_integer(char c)
  {
    return (c ≥ '0' ∧ c ≤ '9');
  }
```

21. is_space. Returns a boolean indicating whether the given character is an ASCII whitespace (0x20).

```
  bool is_space(char c)
  {
    return (c ≡ '␣');
  }
```

22. parse_two_digit. Attempts to parse 2-character sequence from a buffer into 2-digit number. Returns a boolean indicating whether the sequence is representing a number and parsing was finished successfully.

```
bool parse_two_digit(struct string_buffer *buf, size_t pos, int8_t * out)
{
  if (pos + 1 ≥ buf→size) {
    return false;
  }
  char c1 = buf→data[pos];
  char c2 = buf→data[pos + 1];
  if (is_space(c1) ∧ is_space(c2)) {
    return true;
  }
  if (is_space(c1)) {
    c1 = '0';
  }
  if (is_space(c2)) {
    c2 = '0';
  }
  if (¬is_integer(c1) ∨ ¬is_integer(c2)) {
    return false;
  }
  *out = (c1 - '0') * 10 + (c2 - '0');
  return true;
}
```

23. parse_header. Attempts to parse the text stored in a buffer as though it was the header of a translate file. If successful, the values of parameters are stored in `params`. Return value indicates whether the parsing succeeded.

```
bool parse_header(struct string_buffer *buf, struct params *params)
{
  int8_t val = -1;
  if (¬parse_two_digit(buf, 0, &val)) {
    return false;
  }
  else if (val ≠ -1) {
    params→left_hyphen_min = val;
  }
  val = -1;
  if (¬parse_two_digit(buf, 2, &val)) {
    return false;
  }
  else if (val ≠ -1) {
    params→right_hyphen_min = val;
  }
  if (buf→size ≥ 6 ∧ ¬is_space(buf→data[5])) {
    params→bad_hyphen = buf→data[5];
  }
  if (buf→size ≥ 7 ∧ ¬is_space(buf→data[6])) {
    params→missed_hyphen = buf→data[6];
  }
  if (buf→size ≥ 8 ∧ ¬is_space(buf→data[7])) {
    params→good_hyphen = buf→data[7];
  }
  return true;
}
```

24. parse_letters. Attempts to parse the text stored in a buffer as though it was a non-header line of a translate file. If successful, the respective letter and its uppercase variants are stored in the translation table. If any of the letters was already in the table, the method fails. Return values indicates the success of parsing.

```
bool parse_letters(struct string_buffer *buf, struct translate_table *tt, struct trie *helper_trie)
{
  if (buf→size ≡ 0) {
    fprintf(stderr, "Empty␣line␣in␣translate␣file\n");
    return false;
  }
  char separator = buf→data[0];
  if (buf→size > 1 ∧ buf→data[1] ≡ separator) {      /* a comment */
    return true;
  }
  size_t letter_index = tt→letter_count + 1;
  size_t out_index;
  struct string_buffer *letter = init_buffer(4);
  if (letter ≡ Λ) {
    return false;
  }
  bool lower = true;
  if (¬append_char(buf, separator)) {
    destroy_buffer(letter);
    return false;
  }
  for (size_t i = 1; i < buf→size; i++) {
    char c = buf→data[i];
    if (c ≡ separator) {
      if (letter→size ≡ 0) {
        break;
      }
      if (¬append_char(letter, '\0') ∨ ¬insert_pattern(tt→mapping, letter→data, &out_index, helper_trie))
        {
        destroy_buffer(letter);
        return false;
      }
      tt→mapping→aux[out_index] = letter_index;
      if (lower) {
        if (letter_index ≥ tt→letter_capacity) {
          size_t new_capacity = tt→letter_capacity * 2;
          size_t *new_array = realloc(tt→index_to_alphabet, new_capacity * sizeof(size_t));
          if (new_array ≡ Λ) {
            destroy_buffer(letter);
            return false;
          }
          tt→index_to_alphabet = new_array;
          tt→letter_capacity = new_capacity;
        }
        tt→index_to_alphabet[letter_index] = tt→alphabet→size;
        if (¬append_string(tt→alphabet, letter→data)) {
          destroy_buffer(letter);
```

```
            return false;
          }
          tt¬letter_count++;
          lower = false;
        }
        reset_buffer(letter);
      }
      else {
        if (¬append_char(letter, c)) {
          destroy_buffer(letter);
          return false;
        }
      }
    }
    destroy_buffer(letter);
    return true;
  }
```

25. default_ascii_mapping. Fetches default ASCII character mapping into the translate table. Return value indicate the success of fetching.

```
bool default_ascii_mapping(struct translate_table *tt, struct trie *helper_trie)
{
  size_t out_index;
  size_t letter_index;
  char upper;
  for (char c = 'a'; c ≤ 'z'; c++) {
    letter_index = tt→letter_count + 1;
    upper = c − ('a' − 'A');
    if (letter_index ≥ tt→letter_capacity) {
      size_t new_capacity = tt→letter_capacity * 2;
      size_t *new_array = realloc(tt→index_to_alphabet, new_capacity * sizeof(size_t));
      if (new_array ≡ Λ) {
        return false;
      }
      tt→index_to_alphabet = new_array;
      tt→letter_capacity = new_capacity;
    }
    tt→index_to_alphabet[letter_index] = tt→alphabet→size;
    if (¬insert_pattern(tt→mapping, (const char[]){c, '\0'}, &out_index,
          helper_trie) ∨ ¬append_string(tt→alphabet, (const char[]){c, '\0'})) {
      return false;
    }
    tt→mapping→aux[out_index] = letter_index;
    if (¬insert_pattern(tt→mapping, (const char[]){upper, '\0'}, &out_index, helper_trie)) {
      return false;
    }
    tt→mapping→aux[out_index] = letter_index;
    tt→letter_count++;
  }
  letter_index = tt→letter_count + 1;
  if (letter_index ≥ tt→letter_capacity) {
    size_t new_capacity = tt→letter_capacity * 2;
    size_t *new_array = realloc(tt→index_to_alphabet, new_capacity * sizeof(size_t));
    if (new_array ≡ Λ) {
      return false;
    }
    tt→index_to_alphabet = new_array;
    tt→letter_capacity = new_capacity;
  }
  tt→index_to_alphabet[letter_index] = tt→alphabet→size;
  if (¬insert_pattern(tt→mapping, (const char[]){EDGE_OF_WORD, '\0'}, &out_index,
        helper_trie) ∨ ¬append_string(tt→alphabet, (const char[]){EDGE_OF_WORD, '\0'})) {
    return false;
  }
  tt→mapping→aux[out_index] = letter_index;
  tt→letter_count++;
  return true;
}
```

26. Pattern file processing. The user can provide `utfpatgen` with initial set of patterns to work with. Similarly to the translate file, the pattern file must be given as input, but may be left empty. Each line of the file represents one pattern, with following format required:

$< hyph.level >< character >< hyph.level > \ldots < hyph.level >$

Zero levels are omitted implicitly. A special character ('.' in `patgen`, `EDGE_OF_WORD` '0xff' in `utfpatgen`) denotes edges of the word (it can be only used as the first or last character of a pattern). The original `patgen`program that supports only levels up to 9 represents them simply as ASCII numeric literals '0' to '9'. On the other hand, `utfpatgen` expects each byte representing a level preceded with a special `HYPHEN_FLAG` byte of hexadecimal value '0xfe', so the range is extended up to 253. The '0xfe' and '0xff' bytes are not used by any UTF-8 character by design.

27. read_patterns. Iterates over the pattern file and reads its entries into a pattern trie. Return value indicates whether the whole file has been read and parsed successfully.

```
bool read_patterns(struct params *params, struct pattern_trie *pt, struct translate_table *tt, struct
        pass_stats *ps)
{
  ps→level_pattern_cnt = 0;
  ps→max_level = 0;
  struct string_buffer *buf = init_buffer(16);
  if (buf ≡ Λ) {
    return false;
  }
  buf→eof = false;
  struct pattern *pat = init_pattern(16);
  if (pat ≡ Λ) {
    destroy_buffer(buf);
    return false;
  }
  struct trie *helper_trie = init_trie(256);
  if (helper_trie ≡ Λ) {
    destroy_pattern(pat);
    destroy_buffer(buf);
    return false;
  }
  while (¬buf→eof) {
    if (¬read_line(params→pattern_file, buf) ∨ ¬parse_pattern(buf, pat, tt) ∨ ¬insert_new_pattern(pat, pt,
          ps, helper_trie)) {
      destroy_trie(helper_trie);
      destroy_pattern(pat);
      destroy_buffer(buf);
      return false;
    }
  }
  printf("%zu␣patterns␣read␣in\n", ps→level_pattern_cnt);
  printf("pattern␣trie␣has␣%zu␣nodes,␣trie_max␣=␣%zu,␣%zu␣outputs\n", pt→t→occupied,
      pt→t→node_max, pt→ops→count);
  destroy_trie(helper_trie);
  destroy_pattern(pat);
  destroy_buffer(buf);
  return true;
}
```

28. parse_pattern. Processes the pattern text from buffer, translates it to lowercase and inserts it into a pattern structure. Return value indicates the success of translation and parsing.

```
bool parse_pattern(struct string_buffer *buf, struct pattern *out_pattern, struct translate_table *tt)
{
  reset_pattern(out_pattern);
  char c;
  bool next_hyphen = false;
  struct string_buffer *letter = init_buffer(4);
  if (letter ≡ Λ) {
    return false;
  }
  for (size_t i = 0; i < buf→size; i++) {
    c = buf→data[i];
    if (c ≡ EDGE_OF_WORD ∧ i ≠ 0 ∧ i ≠ buf→size − 1) {
      fprintf(stderr, "Edge␣of␣word␣found␣inside␣a␣pattern.\n");
      destroy_buffer(letter);
      return false;
    }
    if (next_hyphen) {
      if (¬set_hyphen(out_pattern, out_pattern→size, (uint8_t)c)) {
        destroy_buffer(letter);
        return false;
      }
      next_hyphen = false;
      continue;
    }
    if (is_utf_start_byte(c) ∧ letter→size > 0) {
      if (¬append_char(letter, '\0')) {
        destroy_buffer(letter);
        return false;
      }
      size_t letter_index = get_letter_index(tt, letter→data);
      if (letter_index ≡ 0) {
        fprintf(stderr, "Unknown␣letter␣%s␣found␣in␣a␣pattern.\n", letter→data);
        destroy_buffer(letter);
        return false;
      }
      if (¬convert_index_to_pattern(letter_index, out_pattern)) {
        destroy_buffer(letter);
        return false;
      }
      reset_buffer(letter);
    }
    if (c ≡ HYPHEN_FLAG) {
      next_hyphen = true;
    }
    else if (¬append_char(letter, c)) {
      destroy_buffer(letter);
      return false;
    }
  }
```

```
    if (next_hyphen) {
      if (¬set_hyphen(out_pattern, out_pattern→size, (uint8_t)c)) {
        destroy_buffer(letter);
        return false;
      }
    }
    else if (letter→size > 0) {
      if (¬append_char(letter, '\0')) {
        destroy_buffer(letter);
        return false;
      }
      size_t letter_index = get_letter_index(tt, letter→data);
      if (letter_index ≡ 0) {
        fprintf(stderr, "Unknown␣letter␣%s␣found␣in␣a␣pattern.\n", letter→data);
        destroy_buffer(letter);
        return false;
      }
      if (¬convert_index_to_pattern(letter_index, out_pattern)) {
        destroy_buffer(letter);
        return false;
      }
    }
    destroy_buffer(letter);
    return true;
  }
```

29. insert_new_pattern. Inserts the pattern into pattern trie and collects statistics along the way. Return value indicates the success of insertion.

```
bool insert_new_pattern(struct pattern *pat, struct pattern_trie *pt, struct pass_stats *ps, struct trie
        *helper_trie)
{
  size_t hyphenation_value, node;
  size_t current_len = 0;
  if (¬insert_pattern(pt→t, pat→text, &node, helper_trie)) {
    return false;
  }
  for (size_t i = 0; i < pat→size; i++) {
    hyphenation_value = get_hyphen(pat, i);
    if (hyphenation_value > 0) {
      ps→level_pattern_cnt++;
      if (¬set_output(pt, node, hyphenation_value, current_len)) {
        return false;
      }
      if (hyphenation_value > ps→max_level) {
        ps→max_level = hyphenation_value;
      }
    }
    if ((uint8_t)pat→text[i] ≠ #ff) {
      current_len++;
    }
  }
  hyphenation_value = get_hyphen(pat, pat→size);
  if (hyphenation_value > 0) {
    ps→level_pattern_cnt++;
    if (¬set_output(pt, node, hyphenation_value, current_len)) {
      return false;
    }
  }
  return true;
}
```

30. Dictionary file processing. The dictionary is the main source of data for pattern generation. The file indeed has to be provided as input, and though no error is raised when it is empty, such case does not make much sense. Each line in the file represents single hyphenated word, with hyphens marked using `GOOD_HYF`, `MISS_HYF`, and `BAD_HYF`. Furthermore, both the whole word and separate hyphens can be weighted by preceding them with `HYPHEN_FLAG` and a value. Similarly to the pattern file, the possible range of weights is increased to 253. If the hyphen weight is omitted, word weight is used. If the word weight is omitted, `utfpatgen` uses the default value 1. The lines of the dictionary file thus look like this:

$$< wordweight >< character >< hyphenweight >< hyphen >< character > \ldots < character >$$

Since some patterns may be tied to the edges of the word, special byte symbol `EDGE_OF_WORD` was introduced that marks the edges. Hexadecimal value of the symbol is '0xff' that is not used by the UTF-8 encoding.

31. process_dictionary. Reads and parses the dictionary, and generates new patterns afterwards. Statistics of the pass are printed out at the end of the method. Return value indicates the success of reading, parsing, and pattern generation.

```
bool process_dictionary(struct params *params, struct translate_table *tt, struct pattern_trie
        *pt, struct pass_stats *ps)
{
  ps→good_cnt = 0;
  ps→bad_cnt = 0;
  ps→miss_cnt = 0;
  params→word_weight = 1;
  if (params→hyph_level % 2 ≡ 1) {
    params→good_dot = MISS_HYF;
    params→bad_dot = NO_HYF;
  }
  else {
    params→good_dot = BAD_HYF;
    params→bad_dot = GOOD_HYF;
  }
  struct count_trie *ct = init_count_trie(256, 256);
  if (ct ≡ Λ) {
    return false;
  }
  printf("processing␣dictionary␣with␣pat_len␣=␣%u,␣pat_dot␣=␣%u\n", params→pat_len,
      params→pat_dot);
  if (¬process_all_words(params, tt, pt, ps, ct)) {
    destroy_count_trie(ct);
    return false;
  }
  printf("\n%zu␣good,␣%zu␣bad,␣%zu␣missed\n", ps→good_cnt, ps→bad_cnt, ps→miss_cnt);
  if (ps→good_cnt + ps→miss_cnt > 0) {
    printf("%.2f␣%%,␣%.2f␣%%,␣%.2f␣%%\n",
        (100 * (float) ps→good_cnt/(float)(ps→good_cnt + ps→miss_cnt)),
        (100 * (float) ps→bad_cnt/(float)(ps→good_cnt + ps→miss_cnt)),
        (100 * (float) ps→miss_cnt/(float)(ps→good_cnt + ps→miss_cnt)));
  }
  printf("%zu␣patterns,␣%zu␣nodes␣in␣count␣trie,␣triec_max␣=␣%zu\n", ct→t→pattern_count,
      ct→t→occupied, ct→t→node_max);
  if (¬collect_count_trie(ct, pt, params, ps)) {
    destroy_count_trie(ct);
    return false;
  }
  destroy_count_trie(ct);
  return true;
}
```

32. process_all_words. Iterates over the lines of the dictionary files and parses them to the count trie. Return value indicates whether the whole file has been processed successfully.

```
bool process_all_words(struct params *params, struct translate_table *tt, struct pattern_trie
        *pt, struct pass_stats *ps, struct count_trie *ct)
{
  rewind(params→dictionary_file);
  uint8_t dot_min = params→pat_dot;
  uint8_t dot_max = params→pat_len − params→pat_dot;
  if (dot_min < params→left_hyphen_min + 1) {
    dot_min = params→left_hyphen_min + 1;
  }
  if (dot_max < params→right_hyphen_min + 1) {
    dot_max = params→right_hyphen_min + 1;
  }
  size_t dot_len = dot_min + dot_max;
  struct string_buffer *buf = init_buffer(64);
  if (buf ≡ Λ) {
    return false;
  }
  buf→eof = false;
  struct word *word = init_word(16);
  if (word ≡ Λ) {
    destroy_buffer(buf);
    return false;
  }
  struct trie *helper_trie = init_trie(256);
  if (helper_trie ≡ Λ) {
    destroy_word(word);
    destroy_buffer(buf);
    return false;
  }
  while (¬buf→eof) {
    if (¬read_line(params→dictionary_file, buf) ∨ ¬parse_word(buf, tt, params,
          word) ∨ ¬hyphenate_word(word, pt, params)) {
      destroy_trie(helper_trie);
      destroy_buffer(buf);
      destroy_word(word);
      return false;
    }
    count_dots(word, params, ps);
    if (word→length ≥ dot_len) {
      if (¬process_word(word, ct, params, helper_trie)) {
        destroy_trie(helper_trie);
        destroy_buffer(buf);
        destroy_word(word);
        return false;
      }
    }
  }
  destroy_trie(helper_trie);
```

```
    destroy_buffer(buf);
    destroy_word(word);
    return true;
  }
```

33. parse_word. Parses the text from buffer to `word` structure. The text is translated to lowercase and hyphens are processed. Return value indicates successful parsing.

```
bool parse_word(struct string_buffer *buf, struct translate_table *tt, struct params *params, struct
        word *out_word)
{
  reset_word(out_word);
  char edge_of_word[] = {EDGE_OF_WORD, '\0'};
  struct string_buffer *letter = init_buffer(4);
  if (letter ≡ Λ) {
    return false;
  }
  uint8_t weight = params→word_weight;
  enum hyphen_type hyf = NO_HYF;
  size_t letter_index = get_letter_index(tt, edge_of_word);
  if (¬convert_index(letter_index, out_word)) {
    destroy_buffer(letter);
    return false;
  }
  char c;
  bool has_weight = false;
  for (size_t i = 0; i < buf→size; i++) {
    c = buf→data[i];
    if (has_weight) {
      weight = (uint8_t)c;
      if (i ≡ 1) {
        params→word_weight = weight;
      }
      has_weight = false;
      continue;
    }
    else if (c ≡ HYPHEN_FLAG) {
      has_weight = true;
      continue;
    }
    else if (c ≡ params→good_hyphen) {
      hyf = GOOD_HYF;
      continue;
    }
    else if (c ≡ params→missed_hyphen) {
      hyf = MISS_HYF;
      continue;
    }
    else if (c ≡ params→bad_hyphen) {
      hyf = BAD_HYF;
      continue;
    }
    else if (is_utf_start_byte(c) ∧ letter→size > 0) {
      if (¬append_char(letter, '\0')) {
        destroy_buffer(letter);
        return false;
```

```
        }
        letter_index = get_letter_index(tt, letter→data);
        if (letter_index ≡ 0) {
          fprintf(stderr, "Character␣'%s'␣in␣word␣'%s'␣not␣known\n", letter→data, buf→data);
          destroy_buffer(letter);
          return false;
        }
        if (¬convert_index(letter_index, out_word)) {
          destroy_buffer(letter);
          return false;
        }
        if (¬set_true_hyphen(out_word, out_word→size − 1, 4 ∗ weight + hyf)) {
          destroy_buffer(letter);
          return false;
        }
        hyf = NO_HYF;
        reset_buffer(letter);
      }
      weight = params→word_weight;
      if (¬append_char(letter, c)) {
        destroy_buffer(letter);
        return false;
      }
    }
    if (letter→size > 0) {
      if (¬append_char(letter, '\0')) {
        destroy_buffer(letter);
        return false;
      }
      letter_index = get_letter_index(tt, letter→data);
      if (letter_index ≡ 0) {
        fprintf(stderr, "Character␣'%s'␣in␣word␣'%s'␣not␣known\n", letter→data, buf→data);
        destroy_buffer(letter);
        return false;
      }
      if (¬convert_index(letter_index, out_word)) {
        destroy_buffer(letter);
        return false;
      }
    }
    letter_index = get_letter_index(tt, edge_of_word);
    if (¬convert_index(letter_index, out_word) ∨ ¬set_true_hyphen(out_word, 0, 0)) {
      destroy_buffer(letter);
      return false;
    }
    destroy_buffer(letter);
    return true;
  }
```

34. count_dots. Collects statistics about the hyphens in parsed word.

```
void count_dots(struct word *word, struct params *params, struct pass_stats *ps)
{
  if (word→length < (uint8_t)(params→right_hyphen_min + 1)) {
    return;
  }
  size_t current_index = 0;
  size_t current_pos = 0;
  bool odd_level;
  size_t dot_index, hyphenation_value, weight;
  enum hyphen_type hyf;
  for (size_t dot_pos = params→left_hyphen_min + 1; dot_pos <
      (uint8_t)(word→length − params→right_hyphen_min); dot_pos++) {
    while (current_pos < dot_pos) {
      if ((uint8_t)word→translated[current_index] ≠ #ff) {
        current_pos++;
      }
      current_index++;
    }
    dot_index = current_index − 1;
    odd_level = (get_found_hyphen(word, dot_index) % 2 ≡ 1);
    hyphenation_value = get_true_hyphen(word, dot_index);
    weight = hyphenation_value/4;
    hyf = hyphenation_value % 4;
    if (hyphenation_value ≡ 0) {
      fprintf(stderr, "Code␣I␣hoped␣unreachable␣was␣reached\n");
      continue;
    }
    else if (hyf % 2 ≡ 0) {
      if (odd_level) {
        ps→bad_cnt += weight;
      }
    }
    else {
      if (odd_level) {
        ps→good_cnt += weight;
      }
      else {
        ps→miss_cnt += weight;
      }
    }
  }
}
```

35. process_word. Generates all candidate patterns from the parsed word and inserts them to count trie. Return value indicates the success of generating and insertion.

```
bool process_word(struct word *word, struct count_trie *ct, struct params *params, struct trie
        *helper_trie)
{
  uint8_t dot_min = params→pat_dot;
  uint8_t dot_max = params→pat_len − params→pat_dot;
  if (dot_min < params→left_hyphen_min + 1) {
    dot_min = params→left_hyphen_min + 1;
  }
  if (dot_max < params→right_hyphen_min + 1) {
    dot_max = params→right_hyphen_min + 1;
  }
  size_t start_pos, end_pos, start_index, dot_index, end_index, node, weight, cnt_index;
  size_t current_pos = 0;
  size_t current_index = 0;
  bool good_pattern;
  enum hyphen_type hyf;
  for (size_t dot_pos = dot_min; dot_pos + dot_max ≤ word→length; dot_pos++) {
    while (current_pos < dot_pos) {
      if ((uint8_t)word→translated[current_index] ≠ #ff) {
        current_pos++;
      }
      current_index++;
    }
    dot_index = current_index − 1;
    if (get_no_more(word, dot_index)) {
      continue;
    }
    hyf = get_true_hyphen(word, dot_index) % 4;
    if (get_found_hyphen(word, dot_index) % 2 ≡ 1) {
      hyf += 2;
    }
    if (hyf ≡ params→good_dot) {
      good_pattern = true;
    }
    else if (hyf ≡ params→bad_dot) {
      good_pattern = false;
    }
    else {
      continue;
    }
    start_pos = dot_pos;
    start_index = current_index;
    while (start_pos + params→pat_dot ≥ dot_pos) {
      if (start_index ≡ 0 ∨ (uint8_t)word→translated[start_index − 1] ≠ #ff) {
        start_pos−−;
      }
      start_index−−;
    }
    start_pos++;
```

```
    start_index ++;
    end_pos = dot_pos;
    end_index = current_index;
    while (end_index < word→size ∧ end_pos < start_pos + params→pat_len) {
      if ((uint8_t)word→translated[end_index] ≠ #ff) {
        end_pos ++;
      }
      end_index ++;
    }
    if (¬insert_substring(ct→t, word→translated, end_index, end_index − start_index, &node, helper_trie)) {
      return false;
    }
    if (ct→cnts→size ≥ ct→cnts→capacity) {
      size_t new_capacity = 2 * ct→cnts→capacity;
      if (resize_pattern_counts(ct→cnts, new_capacity) ≡ Λ) {
        return false;
      }
    }
    cnt_index = ct→t→aux[node];
    if (cnt_index ≡ 0) {
      cnt_index = ct→cnts→size;
      ct→t→aux[node] = cnt_index;
      ct→cnts→size ++;
    }
    weight = get_true_hyphen(word, dot_index)/4;
    if (weight ≡ 0) weight = params→word_weight;
    if (good_pattern) {
      ct→cnts→good[cnt_index] += weight;
    }
    else {
      ct→cnts→bad[cnt_index] += weight;
    }
  }
  return true;
}
```

36. Pattern collection. After the dictionary has been processed, the count trie contains the number of good (supporting) and bad (contradicting) occurrences of each candidate pattern found. There are 3 types of patterns based on these counts and the `good_wt`, `bad_wt`, and `thresh` parameters:

- **good patterns** for which inequality $good * good_wt - bad * bad_wt \geq thresh$ holds. Good patterns are inserted into the pattern trie with corresponding hyphenation level.
- **bad patterns**: for which inequality $good * good_wt < thresh$ holds. Bad patterns are inserted into the pattern trie with a special `BAD_OP_VALUE` level of value 255. Neither them nor their superstrings can become good patterns and are deleted from the pattern trie in the end of the hyphenation level iteration.
- **undecided patterns**: for which none of the inequalities above holds. Some of their superstrings may become good or bad patterns later, so need for further investigation is raised by setting the `more_to_come` flag.

37. collect_count_trie. Generates new patterns from the count trie and prints out statistics. Return value indicates whether the generation ended successfully.

```
bool collect_count_trie(struct count_trie *ct, struct pattern_trie *pt, struct params *params, struct
        pass_stats *ps)
{
  double bad_eff = (double) params→thresh/(double) params→good_wt;
  ps→good_pat_cnt = 0;
  ps→bad_pat_cnt = 0;
  ps→good_cnt = 0;
  ps→bad_cnt = 0;
  ps→more_to_come = false;
  if (¬traverse_count_trie(ct, pt, params, ps)) {
    return false;
  }
  printf("%zu␣good␣and␣%zu␣bad␣patterns␣added", ps→good_pat_cnt, ps→bad_pat_cnt);
  ps→level_pattern_cnt += ps→good_pat_cnt;
  if (ps→more_to_come) {
    printf("␣(more␣to␣come)\n");
  }
  else {
    printf("\n");
  }
  printf("finding␣%zu␣good␣and␣%zu␣bad␣hyphens", ps→good_cnt, ps→bad_cnt);
  if (ps→good_pat_cnt > 0) {
    printf(",␣efficiency␣=␣%.2lf\n",
        (double) ps→good_cnt/(ps→good_pat_cnt + ((double) ps→bad_cnt/bad_eff)));
  }
  else {
    printf("\n");
  }
  printf("pattern␣trie␣has␣%zu␣nodes,␣trie_max␣=␣%zu,␣%zu␣outputs\n", pt→t→occupied,
      pt→t→node_max, pt→ops→count);
  return true;
}
```

38. traverse_count_trie. Searches through the count trie and inserts accepted patterns to the pattern trie. Statistics are collected along the way. Return value indicates whether the whole trie was explored.

```
bool traverse_count_trie(struct count_trie *ct, struct pattern_trie *pt, struct params *params, struct
        pass_stats *ps)
{
  size_t root = 1;
  size_t current_len = 0;
  uint8_t c;
  struct string_buffer *pattern = init_buffer(4 * params→pat_len);
  if (pattern ≡ Λ) {
    return false;
  }
  struct stack *s_base = init_stack(4 * params→pat_len);
  if (s_base ≡ Λ) {
    destroy_buffer(pattern);
    return false;
  }
  struct trie *helper_trie = init_trie(256);
  if (helper_trie ≡ Λ) {
    destroy_buffer(pattern);
    destroy_stack(s_base);
    return false;
  }
  if (¬append_char(pattern, '\0') ∨ ¬put_on_stack(s_base, root)) {
    destroy_trie(helper_trie);
    destroy_buffer(pattern);
    destroy_stack(s_base);
    return false;
  }
  size_t node, op_index, good, bad, cnt_index;
  while (s_base→top > 0) {
    root = get_top_value(s_base);
    pattern→data[pattern→size − 1] += 1;
    c = (uint8_t)pattern→data[pattern→size − 1];
    if (c ≡ 0) {      /* overflow */
      pattern→size −−;
      s_base→top −−;
      if (pattern→size < 1 ∨ (uint8_t)pattern→data[pattern→size − 1] ≠ #ff) {
        current_len −−;
      }
      continue;
    }
    node = root + c;
    if ((uint8_t)ct→t→nodes[node] ≠ c) {
      continue;
    }
    if ((uint8_t)c ≠ #ff) {
      current_len ++;
    }
    if (current_len ≡ params→pat_len) {
```

```
    cnt_index = ct→t→aux[node];
    good = ct→cnts→good[cnt_index];
    bad = ct→cnts→bad[cnt_index];
    if (good > 0 ∨ bad > 0) {
      if (params→good_wt * good < params→thresh) {
        if (¬insert_substring(pt→t, pattern→data, pattern→size, pattern→size, &op_index,
              helper_trie) ∨ ¬set_output(pt, op_index, BAD_OP_VALUE, params→pat_dot)) {
          destroy_trie(helper_trie);
          destroy_buffer(pattern);
          destroy_stack(s_base);
          return false;
        }
        ps→bad_pat_cnt++;
      }
      else if (params→good_wt * good ≥ params→thresh + params→bad_wt * bad) {
        if (¬insert_substring(pt→t, pattern→data, pattern→size, pattern→size, &op_index,
              helper_trie) ∨ ¬set_output(pt, op_index, params→hyph_level, params→pat_dot)) {
          destroy_trie(helper_trie);
          destroy_buffer(pattern);
          destroy_stack(s_base);
          return false;
        }
        ps→good_pat_cnt++;
        ps→good_cnt += good;
        ps→bad_cnt += bad;
      }
      else {
        ps→more_to_come = true;
      }
    }
    if ((uint8_t)c ≠ #ff) {
      current_len--;
    }
    continue;
  }
  root = ct→t→links[node];
  if (root ≡ 0) {
    if ((uint8_t)c ≠ #ff) {
      current_len--;
    }
    continue;
  }
  if (¬append_char(pattern, '\0') ∨ ¬put_on_stack(s_base, root)) {
    destroy_trie(helper_trie);
    destroy_buffer(pattern);
    destroy_stack(s_base);
    return false;
  }
}
destroy_trie(helper_trie);
destroy_buffer(pattern);
destroy_stack(s_base);
```

```
    return true;
  }
```

39. Pattern pruning. Once all the required pattern lengths and dot positions were explored for given hyphenation level, the pattern trie is pruned of the patterns that were marked as bad.

40. delete_bad_patterns. Removes bad patterns from the pattern trie and prints out statistics. Return value indicates success of the operation.

```
bool delete_bad_patterns(struct pattern_trie *pt)
{
  size_t old_op_cnt = pt→ops→count;
  size_t old_trie_cnt = pt→t→occupied;
  if (¬delete_patterns(pt)) {
    return false;
  }
  for (size_t h = 1; h ≤ pt→ops→capacity; h++) {
    if (pt→ops→data[h].value ≡ BAD_OP_VALUE) {
      pt→ops→data[h].value = EMPTY_OP_VALUE;
      pt→ops→count−−;
      pt→ops→data[h].next_op_index = pt→ops→data[0].next_op_index;
      pt→ops→data[0].next_op_index = h;
    }
  }
  printf("%zu␣nodes␣and␣%zu␣outputs␣deleted\n", old_trie_cnt − pt→t→occupied,
      old_op_cnt − pt→ops→count);
  return true;
}
```

41. delete_patterns. Searches through the pattern trie and removes outputs pointing to the bad patterns. Nodes that are not used after the deletion are deleted as well. Return value indicates whether the whole trie was explored.

```
bool delete_patterns(struct pattern_trie *pt)
{
  size_t root = 1;
  struct stack *s_base = init_stack(16);
  if (s_base ≡ Λ) {
    return false;
  }
  struct stack *s_offset = init_stack(16);
  if (s_offset ≡ Λ) {
    destroy_stack(s_base);
    return false;
  }
  struct stack *s_freed = init_stack(16);
  if (s_freed ≡ Λ) {
    destroy_stack(s_base);
    destroy_stack(s_offset);
    return false;
  }
  if (¬put_on_stack(s_base, root) ∨ ¬put_on_stack(s_offset, 0) ∨ ¬put_on_stack(s_freed, (size_t) true)) {
    destroy_stack(s_base);
    destroy_stack(s_offset);
    destroy_stack(s_freed);
    return false;
  }
  size_t node;
  uint8_t c;
  while (s_base→top > 0) {
    root = get_top_value(s_base);
    set_top_value(s_offset, (uint8_t)get_top_value(s_offset) + 1);
    c = (uint8_t)get_top_value(s_offset);
    if (c ≡ 0) {
      bool child_freed = (get_top_value(s_freed) ≡ (size_t) true);
      if (child_freed) {
        if (¬set_base_used(pt→t, root, false)) {
          destroy_stack(s_base);
          destroy_stack(s_offset);
          destroy_stack(s_freed);
          return false;
        }
      }
      s_offset→top −−;
      s_base→top −−;
      s_freed→top −−;
      if (s_base→top > 0) {
        size_t parent_root = get_top_value(s_base);
        uint8_t parent_c = (uint8_t)get_top_value(s_offset);
        size_t parent_node = parent_root + parent_c;
```

```
        if (child_freed) {
          pt→t→links[parent_node] = 0;
          if (pt→t→aux[parent_node] ≡ 0 ∧ parent_root ≠ 1) {
            deallocate_node(pt→t, parent_node);
          }
          else {
            set_top_value(s_freed, (size_t) false);
          }
        }
        else {
          set_top_value(s_freed, (size_t) false);
        }
      }
      continue;
    }
    node = root + c;
    if ((uint8_t)pt→t→nodes[node] ≠ c) {
      continue;
    }
    if (¬link_around_bad_outputs(pt, node)) {
      destroy_stack(s_base);
      destroy_stack(s_offset);
      destroy_stack(s_freed);
      return false;
    }
    if (pt→t→aux[node] > 0 ∨ root ≡ 1) {
      set_top_value(s_freed, (size_t) false);
    }
    else {
      if (pt→t→links[node] ≡ 0) {
        deallocate_node(pt→t, node);
        continue;
      }
    }
    root = pt→t→links[node];
    if (root ≡ 0) {
      continue;
    }
    if (¬put_on_stack(s_base, root) ∨ ¬put_on_stack(s_offset, 0) ∨ ¬put_on_stack(s_freed, (size_t) true)) {
      destroy_stack(s_base);
      destroy_stack(s_offset);
      destroy_stack(s_freed);
      return false;
    }
  }
  destroy_stack(s_base);
  destroy_stack(s_offset);
  destroy_stack(s_freed);
  return true;
}
```

42. deallocate_node. Removes unused node from a trie and links it to the beginning of free space chain.

```
void deallocate_node(struct trie *t, size_t t_index)
{
  size_t old_head = t→links[0];
  t→nodes[t_index] = 0;
  t→links[0] = t_index;
  t→aux[t_index] = 0;
  t→links[t_index] = old_head;
  t→aux[old_head] = t_index;
  t→occupied−−;
}
```

43. link_around_bad_outputs. Fixes the output linking to not contain the outputs that will be deleted. Returns true upon success.

```
bool link_around_bad_outputs(struct pattern_trie *pt, size_t t_index)
{
  size_t lookup_index = pt→t→aux[t_index];
  if (lookup_index ≡ 0) {
    return true;
  }
  size_t op_index = pt→ops→lookup[lookup_index];
  size_t free_list_head = pt→ops→data[0].next_op_index;
  size_t h = 0;
  pt→ops→data[0].next_op_index = op_index;
  size_t n = pt→ops→data[0].next_op_index;
  while (n > 0) {
    if (pt→ops→data[n].value ≡ BAD_OP_VALUE) {
      pt→ops→data[h].next_op_index = pt→ops→data[n].next_op_index;
    }
    else {
      h = n;
    }
    n = pt→ops→data[h].next_op_index;
  }
  if (h ≡ 0) {
    if (pt→ops→lookup[lookup_index] > 0) {
      pt→ops→lookup[lookup_index] = 0;
      pt→ops→lookup_cnt−−;
    }
    pt→t→aux[t_index] = 0;
  }
  else {
    pt→ops→lookup[lookup_index] = pt→ops→data[0].next_op_index;
  }
  pt→ops→data[0].next_op_index = free_list_head;
  return true;
}
```

44. Pattern output. At the end of the run, `utfpatgen` prints out the final set of patterns to the output file. The format is the same as the required format of the input pattern file – each value of a hyphenation level is preceded by the `HYPHEN_FLAG` symbol, the '.' character marking an edge of word.

45. output_patterns. Traverses through the pattern trie and writes all the patterns present there to the output file. Returns true if the whole trie has been explored and patterns successfully written.

```
bool output_patterns(struct pattern_trie *pt, struct translate_table *tt, FILE *output_file)
{
  size_t root = 1;
  uint8_t c;
  struct string_buffer *pattern = init_buffer(16);
  if (pattern ≡ Λ) {
    return false;
  }
  struct stack *s_base = init_stack(16);
  if (s_base ≡ Λ) {
    destroy_buffer(pattern);
    return false;
  }
  if (¬append_char(pattern, '\0') ∨ ¬put_on_stack(s_base, root)) {
    destroy_buffer(pattern);
    destroy_stack(s_base);
    return false;
  }
  size_t node;
  while (s_base→top > 0) {
    root = get_top_value(s_base);
    pattern→data[pattern→size − 1] += 1;
    c = (uint8_t)pattern→data[pattern→size − 1];
    if (c ≡ 0) {
      pattern→data[pattern→size − 1] = '\0';
      pattern→size −−;
      s_base→top −−;
      continue;
    }
    node = root + c;
    if ((uint8_t)pt→t→nodes[node] ≠ c) {
      continue;
    }
    if (pt→t→aux[node] > 0) {
      output_pattern(pattern, tt, pt→ops, pt→t→aux[node], output_file);
    }
    root = pt→t→links[node];
    if (root ≡ 0) {
      continue;
    }
    if (¬append_char(pattern, '\0') ∨ ¬put_on_stack(s_base, root)) {
      destroy_buffer(pattern);
      destroy_stack(s_base);
      return false;
    }
  }
  destroy_buffer(pattern);
  destroy_stack(s_base);
```

```
  return true;
}
```

46. output_pattern. Writes single pattern to the output file.

```
void output_pattern(struct string_buffer *pattern, struct translate_table *tt, struct outputs *ops, size_t
        op_index, FILE *output_file) {
  if (op_index ≡ 0) {
    return;
  }
  size_t pattern_position = 0;
  size_t level, letter_index;
  char *word_index = pattern→data; size_t edge_of_word_idx = get_letter_index (tt, (char[2])
        {EDGE_OF_WORD, '\0'} ) ;
    while (word_index < pattern→data + pattern→size) {
      level = get_highest_level(ops, op_index, pattern_position);
      if (level > 0) {
        fputc('\xfe', output_file);
        fputc((uint8_t)level, output_file);
      }
      letter_index = convert_byte_sequence(&word_index);
      if (letter_index ≡ edge_of_word_idx) {
        fputc('.', output_file);
      }
      else {
        fprintf (output_file, "%s", tt→alphabet→data + tt→index_to_alphabet[letter_index]);
      }
      pattern_position++;
    }
    level = get_highest_level(ops, op_index, pattern_position);
    if (level > 0) {
      fputc('\xfe', output_file);
      fputc((uint8_t)level, output_file);
    }
    fputc('\n', output_file); }
```

47. get_highest_level. Selects the hyphenation level to be used in given dot position. Return the level value.

```
size_t get_highest_level(struct outputs *ops, size_t start_index, size_t position)
{
  size_t highest = 0;
  size_t op_index = ops→lookup[start_index];
  struct output op;
  while (op_index > 0) {
    op = ops→data[op_index];
    if (op.position ≡ position ∧ op.value ≠ BAD_OP_VALUE ∧ op.value > highest) {
      highest = op.value;
    }
    op_index = op.next_op_index;
  }
  return highest;
}
```

48. Hyphenation. Hyphenation appears on several places in the algorithm. Already in the dictionary processing, the words are hyphenated with the current set of patterns. The other case occurs at the absolute end of `utfpatgen` when the user wishes to see the hyphenated dictionary – the final set of patterns is used to hyphenate the dictionary entries into 'pattmp.X' file. The 'X' marks the last hyphenation level explored.

49. hyphenate_word. Hyphenates the given word according to the actual set of patterns from pattern trie. Found hyphens are stored in word's `found_hyphens` field. Returns true if hyphenation ended without errors.

```
bool hyphenate_word(struct word *word, struct pattern_trie *pt, struct params *params)
{
  size_t current_index = 0;
  size_t current_pos = 0;
  size_t node, base, start_index, dot_index, end_index, op_index, dot_pos, end_pos;
  struct output op;
  if (word→length < (uint8_t)(params→right_hyphen_min + 1)) {
    return true;
  }
  size_t start_pos = 0;
  for (size_t i = 0; i < word→length − params→right_hyphen_min; i++) {
    while (current_pos < start_pos) {
      if ((uint8_t)word→translated[current_index] ≠ #ff) {
        current_pos++;
      }
      current_index++;
    }
    start_index = current_index;
    end_index = current_index;
    end_pos = current_pos;
    node = 1 + (uint8_t)word→translated[start_index];
    while (pt→t→nodes[node] ≡ word→translated[end_index]) {
      if ((uint8_t)word→translated[end_index] ≠ #ff) {
        end_pos++;
      }
      end_index++;
      op_index = pt→ops→lookup[pt→t→aux[node]];
      while (op_index > 0) {
        op = pt→ops→data[op_index];
        dot_pos = start_pos;
        dot_index = start_index;
        while (dot_pos < start_pos + op.position) {
          if ((uint8_t)word→translated[dot_index] ≠ #ff) {
            dot_pos++;
          }
          dot_index++;
        }
        if (dot_index > 0) {
          dot_index−−;
          if (op.value < BAD_OP_VALUE ∧ get_found_hyphen(word, dot_index) < op.value) {
            if (¬set_found_hyphen(word, dot_index, op.value)) {
              return false;
            }
          }
          if (op.value ≥ params→hyph_level) {
            if ((end_pos + params→pat_dot ≤ dot_pos + params→pat_len) ∧ (dot_pos ≤
                start_pos + params→pat_dot)) {
              if (¬set_no_more(word, dot_index, true)) {
                return false;
```

```
                }
              }
            }
          }
          op_index = op.next_op_index;
        }
        base = pt¬t¬links[node];
        if (base ≡ 0) {
          break;
        }
        node = base + (uint8_t)word¬translated[end_index];
      }
      start_pos++;
    }
    return true;
  }
```

50. hyphenate_dictionary. Hyphenates dictionary entries and writes them to newly created `pattmp` file if required. Return value indicates whether the hyphenation and file printout finished successfully.

```
bool hyphenate_dictionary(struct params *params, struct translate_table *tt, struct pattern_trie
        *pt, bool output, struct pass_stats *ps)
{
  ps→good_cnt = 0;
  ps→bad_cnt = 0;
  ps→miss_cnt = 0;
  params→word_weight = 1;
  FILE *pattmp = Λ;
  if (output) {
    char *filename = malloc(11 * sizeof(char));
    if (filename ≡ Λ) {
      return false;
    }
    sprintf(filename, "pattmp.%u", params→hyph_level);
    pattmp = fopen(filename, "w");
    if (pattmp ≡ Λ) {
      free(filename);
      return false;
    }
    printf("writing␣%s\n", filename);
    free(filename);
  }
  if (¬hyphenate_all_words(params, tt, pt, pattmp, ps)) {
    if (pattmp ≠ Λ) {
      fclose(pattmp);
    }
    return false;
  }
  printf("\n%zu␣good,␣%zu␣bad,␣%zu␣missed\n", ps→good_cnt, ps→bad_cnt, ps→miss_cnt);
  if (ps→good_cnt + ps→miss_cnt > 0) {
    printf("%.2f␣%%,␣%.2f␣%%,␣%.2f␣%%\n",
        (100 * (float) ps→good_cnt/(float)(ps→good_cnt + ps→miss_cnt)),
        (100 * (float) ps→bad_cnt/(float)(ps→good_cnt + ps→miss_cnt)),
        (100 * (float) ps→miss_cnt/(float)(ps→good_cnt + ps→miss_cnt)));
  }
  if (pattmp ≠ Λ) {
    fclose(pattmp);
  }
  return true;
}
```

51. hyphenate_all_words. Iterates over the words in the dictionary, parses, hyphenates, and writes them to the `pattmp` file. Returns true if no error occurs.

```
bool hyphenate_all_words(struct params *params, struct translate_table *tt, struct pattern_trie
        *pt, FILE *pattmp, struct pass_stats *ps)
{
  rewind(params→dictionary_file);
  struct string_buffer *buf = init_buffer(64);
  if (buf ≡ Λ) {
    return false;
  }
  struct word *word = init_word(16);
  if (word ≡ Λ) {
    destroy_buffer(buf);
    return false;
  }
  buf→eof = false;
  while (¬buf→eof) {
    if (¬read_line(params→dictionary_file, buf) ∨ ¬parse_word(buf, tt, params,
          word) ∨ ¬hyphenate_word(word, pt, params)) {
      destroy_buffer(buf);
      destroy_word(word);
      return false;
    }
    count_dots(word, params, ps);
    if (pattmp ≠ Λ ∧ word→length > 2) {
      output_hyphenated_word(pattmp, word, tt, params);
    }
  }
  destroy_buffer(buf);
  destroy_word(word);
  return true;
}
```

52. output_hyphenated_word. Writes the parsed and hyphenated word to the `pattmp` file.

```
void output_hyphenated_word(FILE *pattmp, struct word *word, struct translate_table *tt, struct
        params *params) {
  if (params→word_weight > 1) {
    fprintf(pattmp, "%d", params→word_weight);
  }
  size_t weight, letter_index, dot_pos = 0; size_t edge_of_word_idx = get_letter_index (tt, (char[2])
          {EDGE_OF_WORD, '\0'} ) ;
    char *word_index = word→translated;
    bool has_hyphen, found_hyphen;
    while (word_index < word→translated + word→size) {
      found_hyphen = (get_found_hyphen(word, word_index − word→translated) % 2 ≡ 1);
      weight = get_true_hyphen(word, word_index − word→translated);
      has_hyphen = (weight % 2 ≡ 1);
      letter_index = convert_byte_sequence(&word_index);
      if (letter_index ≡ edge_of_word_idx) {
        continue;
      }
      fprintf(pattmp, "%s", tt→alphabet→data + tt→index_to_alphabet[letter_index]);
      dot_pos++;
      if (weight ≡ 0 ∨ dot_pos < params→left_hyphen_min ∨ dot_pos ≥
            word→length − params→right_hyphen_min − 1) {
        continue;
      }
      weight /= 4;
      if (weight ≠ params→word_weight) {
        fprintf(pattmp, "%zu", weight);
      }
      if (found_hyphen ∧ has_hyphen) {
        fputc((char) params→good_hyphen, pattmp);
      }
      else if (found_hyphen ∧ ¬has_hyphen) {
        fputc((char) params→bad_hyphen, pattmp);
      }
      else if (¬found_hyphen ∧ has_hyphen) {
        fputc((char) params→missed_hyphen, pattmp);
      }
    }
    fputc('\n', pattmp); }
```

53. UTF-8 specifics. Following methods greatly simplify dealing with UTF-8 encoding. We took advantage of the design that allows to easily determine whether a byte is the first one in UTF-8 character – its two highest bits are not '10'. The first byte also encodes the number of bytes that form the UTF-8 character. If the highest bit is '0', the character comprises a single byte and its meaning is the same as it would be in ASCII encoding. If the highest bit is '1', the number of '1' bits on the highest positions equals the number of bytes of the character (with the exception of '10' which is not allowed in the leading byte), e.g., the byte of binary value '1110XXXX' is a beginning of 3-byte UTF-8 character. Although bytes '0xfe' and '0xff' are not formally allowed in UTF-8, we treat them as one-byte characters with special meaning (`HYPHEN_FLAG`, `EDGE_OF_WORD` respectively).

54. is_utf_start_byte. Returns true if the given byte is the start byte of a UTF-8 character or a `utfpatgen` special symbol.

```
inline bool is_utf_start_byte(uint8_t byte)
{
  return byte < #80 ∨ byte > #bf;
}
```

55. Structures. This section focuses on the data structures we used in `utfpatgen` and their components. Here you can find information about the basic structures such as tries, buffers, and stacks, as well as composite structures like the pattern and count trie, translate table, or word.

We tried to name the methods for basic manipulation with the structures consistently across the whole implementation. Thanks to that, you will find several methods with common prefixes: `init_XX` performs initial allocation of resources for structure *XX*, `resize_XX` reallocates them if the capacity threshold is reached, `reset_XX` reverts the structure to initial state, and `destroy_XX` frees all the resources to avoid memory leaks. Furthermore, we use `get_YY` and `set_YY` methods to read and write into the fields of given structure instead of direct access. This allows us to perform additional checks on the input values.

56. Trie. The packed trie structure forms the backbone of both `patgen` and `utfpatgen` algorithms. The implementation is very similar in both cases, a set of arrays representing a n-ary tree, condensed for better space effectiveness. Single node of a trie comprises a *value*, *link* and *aux* pointers, and *base* indicator. Every admissible index of the arrays is either occupied by a node, or it is an empty space (its value is 0). Index 0 is left empty and serves as the head of empty space chain, index 1 is the root node of the trie itself.

If we want to traverse the trie, that means finding the node corresponding to given sequence of values $x_1 \dots x_n$ (if such node exists), we start in the root node r_0. Our next destination is node $n_1 = r_0 + x_1$. Then we check whether $value(n_1) = x_1$ and if the equation holds, we set $r_1 = link(n_1)$ as the new root. If the equation does not hold or $r_1 = 0$, we can end the search and conclude that the sequence is not present in the trie. Otherwise, we repeat the steps for $n_2 \dots n_n$ and return the n_n node as desired result.

The internal representation of the trie consists a structure with following fields:

- **capacity**: maximal node index that can be inserted,
- **occupied**: number of nodes currently in the trie,
- **node_max**: the highest occupied node index,
- **base_max**: the highest base index,
- **pattern_count**: number of patterns (sequences) in the trie,
- **nodes**: array of values,
- **links**: array of links, pointing either to the base of occupied node's subtree, or to the next empty index in case of an empty space,
- **aux**: array of links, pointing either to occupied node's output, or to the previous empty index in case of an empty space,
- **taken**: bit array storing the information about whether indices are used as bases.

```
struct trie *init_trie(size_t capacity)
{
  struct trie *t = malloc(sizeof(struct trie));
  if (t ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  t→capacity = capacity;
  t→nodes = calloc(capacity, sizeof(char));
  t→links = calloc(capacity, sizeof(size_t));
  t→aux = calloc(capacity, sizeof(size_t));
  t→taken = calloc((capacity/8) + 1, sizeof(char));      /* bit array */
  if (t→nodes ≡ Λ ∨ t→links ≡ Λ ∨ t→aux ≡ Λ ∨ t→taken ≡ Λ ∨ ¬set_base_used(t, 1, true)) {
    fputs("Allocation␣error\n", stderr);
    free(t→nodes);
    free(t→links);
    free(t→aux);
    free(t→taken);
    free(t);
    return Λ;
  }
  t→node_max = 0;
  t→base_max = 0;
  t→occupied = 0;
  t→pattern_count = 0;
  relink_trie(t);
  return t;
}
```

57. put_first_level. Inserts the set of one-byte sequences '0x01' to '0xff' into the trie. It can be called during trie initialization, since these nodes will never be moved elsewhere. Return value indicates whether all the insertions succeeded.

```
bool put_first_level(struct trie *t)
{
  size_t root = 1;
  size_t n_bytes = 255;
  if (t→capacity < n_bytes + 2) {
    size_t new_capacity = (((n_bytes + 2)/t→capacity) + 1) * t→capacity;
    if (resize_trie(t, new_capacity) ≡ Λ) {
      return false;
    }
  }
  for (size_t i = 1; i ≤ n_bytes; i++) {
    t→nodes[root + i] = (uint8_t)i;
    t→links[root + i] = 0;
    t→aux[root + i] = 0;
  }
  t→node_max = root + n_bytes;
  t→base_max = root;
  t→occupied = n_bytes;
  if (¬set_base_used(t, root, true)) {
    return false;
  }
  set_links(t, 0, t→node_max + 1);
  return true;
}
struct trie *resize_trie(struct trie *t, size_t new_capacity)
{
  void *new_nodes = realloc(t→nodes, new_capacity * sizeof(char));
  if (new_nodes ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  t→nodes = new_nodes;
  size_t *new_links = realloc(t→links, new_capacity * sizeof(size_t));
  if (new_links ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  t→links = new_links;
  size_t *new_aux = realloc(t→aux, new_capacity * sizeof(size_t));
  if (new_aux ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  t→aux = new_aux;
  size_t old_taken_bytes = (t→capacity/8) + 1;
  size_t new_taken_bytes = (new_capacity/8) + 1;
  char *new_taken = realloc(t→taken, new_taken_bytes * sizeof(char));
```

```
  if (new_taken ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  t→taken = new_taken;
  if (new_taken_bytes > old_taken_bytes) {
    memset(t→taken + old_taken_bytes, 0, (new_taken_bytes − old_taken_bytes));
  }
  memset(t→nodes + t→capacity, 0, (new_capacity − t→capacity) ∗ sizeof(char));
  t→capacity = new_capacity;
  relink_trie(t);
  return t;
}
void destroy_trie(struct trie ∗t)
{
  free(t→nodes);
  free(t→links);
  free(t→aux);
  free(t→taken);
  free(t);
}
```

58. relink_trie. Connects all empty spaces in the trie.

```
void relink_trie(struct trie ∗t)
{
  size_t last_free = 0;
  for (size_t node = 2; node < t→capacity; node++) {
    if (¬is_node_occupied(t, node)) {
      set_links(t, last_free, node);
      last_free = node;
    }
  }
  set_links(t, last_free, 0);
}
```

59. copy_node. Replicates a node between two tries. Return value indicates the success of the operation.

```
bool copy_node(struct trie *from, size_t from_index, struct trie *to, size_t to_index)
{
  if (to_index ≥ to→capacity) {
    size_t new_capacity = ((to_index/to→capacity) + 1) * to→capacity;
    if (resize_trie(to, new_capacity) ≡ Λ) {
      return false;
    }
  }
  bool from_free = (from→nodes[from_index] ≡ 0);
  bool to_free = (to→nodes[to_index] ≡ 0);
  to→nodes[to_index] = from→nodes[from_index];
  to→links[to_index] = from→links[from_index];
  to→aux[to_index] = from→aux[from_index];
  if (¬from_free ∧ to_free) {
    to→occupied++;
  }
  else if (from_free ∧ ¬to_free) {
    to→occupied--;
  }
  return true;
}
bool get_base_used(struct trie *t, size_t index)
{
  if (index ≥ t→capacity) {
    return false;
  }
  size_t byte_index = index/8;
  size_t bit_index = index % 8;
  return (t→taken[byte_index] & (1 ≪ bit_index)) ≠ 0;
}
bool set_base_used(struct trie *t, size_t index, bool used)
{
  if (index ≥ t→capacity) {
    if (resize_trie(t, index + 1) ≡ Λ) {
      return false;
    }
  }
  size_t byte_index = index/8;
  size_t bit_index = index % 8;
  if (used) {
    t→taken[byte_index] |= (1 ≪ bit_index);
  }
  else {
    t→taken[byte_index] &= ∼(1 ≪ bit_index);
  }
  return true;
}
```

60. set_links. Sets the relevant (`link` and `aux`) pointers to link two empty spaces next to each other.

```
void set_links(struct trie *t, size_t from, size_t to)
{
  t→links[from] = to;
  t→aux[to] = from;
}
```

61. is_node_occupied. Returns true if the given trie index is occupied by a node.

```
bool is_node_occupied(struct trie *t, size_t index)
{
  return t→nodes[index] ≠ 0;
}
```

62. insert_pattern. Attempts to put the given pattern into the trie, possibly creating nodes along the way. If the insertion is successful, true is returned and the resulting index is stored in `out_op_index`.

```
bool insert_pattern(struct trie *t, const char *pattern, size_t *out_op_index, struct trie *helper_trie)
{
  size_t length = strlen(pattern);
  return insert_substring(t, pattern, length, length, out_op_index, helper_trie);
}
```

63. insert_substring. Attempts to put substring $[end - length, end)$ of the given pattern into the trie, possibly creating nodes along the way. If the insertion is successful, true is returned and the resulting index is stored in `out_op_index`.

```
bool insert_substring(struct trie *t, const char *pattern, size_t end, size_t length, size_t
        *out_op_index, struct trie *helper_trie)
{
  size_t index = end - length;
  size_t base = 1;
  size_t node = base + (uint8_t)pattern[index];
  size_t fit;
  size_t node_prev = 0;
  bool new_pattern = false;
  while (index < end ∧ base > 0) {
    node = base + (uint8_t)pattern[index];
    if (node ≥ t→capacity) {
      size_t new_capacity = ((node/t→capacity) + 1) * t→capacity;
      if (resize_trie(t, new_capacity) ≡ Λ) {
        return false;
      }
    }
    if (t→nodes[node] ≠ pattern[index]) {
      new_pattern = true;
      if (t→nodes[node] ≡ 0) {
        t→nodes[node] = pattern[index];
        set_links(t, t→aux[node], t→links[node]);
        t→aux[node] = 0;
        t→links[node] = 0;
        t→occupied++;
        if (node > t→node_max) {
          t→node_max = node;
        }
      }
      else {
        if (¬repack(t, helper_trie, &node_prev, &node, pattern[index])) {
          return false;
        }
      }
    }
    index++;
    node_prev = node;
    base = t→links[node];
  }
  helper_trie→links[1] = 0;
  helper_trie→aux[1] = 0;
  helper_trie→node_max = 1;
  while (index < end) {
    helper_trie→nodes[1] = pattern[index];
    if (¬first_fit(t, helper_trie, &fit)) {
      return false;
    }
    t→links[node] = fit;
```

```
    base = fit;
    node = base + (uint8_t)pattern[index];
    index++;
    new_pattern = true;
  }
  *out_op_index = node;
  if (new_pattern) {
    t→pattern_count++;
  }
  return true;
}
```

64. repack. Attempts to move a subtree of the trie, so that it fits new node with given value. If the repacking finishes successfully, the index for new node is stored in `base` and true is returned.

```
bool repack(struct trie *t, struct trie *q, size_t *node, size_t *base, char value)
{
  if (¬unpack(t, *base − (uint8_t)value, q)) {
    return false;
  }
  if (q→node_max ≥ q→capacity) {
    size_t new_capacity = ((q→node_max/q→capacity) + 1) * q→capacity;
    if (resize_trie(q, new_capacity) ≡ Λ) {
      return false;
    }
  }
  q→nodes[q→node_max] = value;
  q→links[q→node_max] = 0;
  q→aux[q→node_max] = 0;
  size_t fit;
  if (¬first_fit(t, q, &fit)) {
    return false;
  }
  *base = fit;
  t→links[*node] = *base;
  *base += (uint8_t)value;
  return true;
}
```

65. unpack. Moves all nodes with the given base to auxiliary trie.

```
bool unpack(struct trie *from, size_t base, struct trie *to)
{
  to→node_max = 1;
  for (size_t i = 1; i < 256; i++) {
    size_t from_index = base + i;
    if ((uint8_t)from→nodes[from_index] ≡ i) {
      if (¬copy_node(from, from_index, to, to→node_max)) {
        return false;
      }
      deallocate_node(from, from_index);
      to→node_max++;
    }
  }
  if (¬set_base_used(from, base, false)) {
    return false;
  }
  return true;
}
```

66. first_fit. Finds the base of trie `t` that fits the nodes from auxiliary trie `q` and copies them into their new indices. Return true if the search and copying finishes successfully.

```
bool first_fit(struct trie *t, struct trie *q, size_t *out_base)
{
  size_t base;
  if (¬find_base_for_first_fit(t, q, &base)) {
    return false;
  }
  for (size_t q_index = 1; q_index ≤ q→node_max; q_index++) {
    size_t t_index = base + (uint8_t)q→nodes[q_index];
    set_links(t, t→aux[t_index], t→links[t_index]);
    if (¬copy_node(q, q_index, t, t_index)) {
      return false;
    }
    if (t_index > t→node_max) {
      t→node_max = t_index;
    }
  }
  if (¬set_base_used(t, base, true)) {
    return false;
  }
  *out_base = base;
  return true;
}
```

67. find_base_for_first_fit. Searches through the trie `t` to find base index that fits the nodes from the `q` trie. If the search is successful, resulting index is stored in `out_base` and true is returned.

```
bool find_base_for_first_fit(struct trie *t, struct trie *q, size_t *out_base)
{
  size_t t_index;
  uint8_t offset;
  if (q→node_max > 5 ∧ t→capacity > t→node_max + 1) {
    t_index = t→node_max + 1;
  }
  else {
    t_index = t→links[0];
  }
  for (size_t i = 0; i ≤ t→capacity; i++) {
    if (t_index ≡ 0) {
      t_index = t→capacity + 1;      /* free for sure after resize */
      size_t new_capacity = 2 * t→capacity;
      if (¬resize_trie(t, new_capacity)) {
        return false;
      }
    }
    offset = (uint8_t)q→nodes[1];
    if (t_index ≤ offset) {
      t_index = t→links[t_index];
      continue;
    }
    *out_base = t_index − offset;
    size_t max_target_index = *(out_base) + 255;
    if (max_target_index ≥ t→capacity) {
      size_t new_capacity = ((max_target_index/t→capacity) + 1) * t→capacity;
      if (resize_trie(t, new_capacity) ≡ Λ) {
        return false;
      }
    }
    if (get_base_used(t, *out_base)) {
      t_index = t→links[t_index];
      continue;
    }
    bool conflict = false;
    for (size_t q_index = q→node_max; q_index ≥ 2; q_index−−) {
      if (is_node_occupied(t, *out_base + (uint8_t)q→nodes[q_index])) {
        conflict = true;
        break;
      }
    }
    if (¬conflict) {
      return true;
    }
    t_index = t→links[t_index];
  }
  printf("Loop detected!\n");
```

```
  return false;
}
```

68. traverse_trie. Returns the index of the node corresponding to the given pattern, or 0 if such node does not exist.

```
size_t traverse_trie(struct trie *t, const char *pattern)
{
  size_t index = 1;
  size_t node = (uint8_t)pattern[0] + 1;
  size_t base = t->links[node];
  while (index < strlen(pattern) ∧ base > 0) {
    base += (uint8_t)pattern[index];
    if (t->nodes[base] ≠ pattern[index]) {
      return 0;
    }
    node = base;
    base = t->links[node];
    index++;
  }
  if (index < strlen(pattern)) {
    return 0;
  }
  return node;
}
```

69. Outputs. The hyphenation information associated to a pattern is stored as a triplet in `output` structure:

- **value**: the hyphenation level,
- **position**: the dot position,
- **next_op_index**: the pointer to next output of the same pattern.

For space-saving purposes, several patterns may share a single output. This means that, for example, for all patterns having single hyphen of level 1 at position 3 we store only one such output to which all the patterns point. It is thus necessary to access the outputs as effectively as possible. `utfpatgen` uses a "hash table" with indirect addressing – simple function $(next_op_index + 313 * position + 361 * value)\%table_capacity) + 1$ computes the index of an output in the `lookup` array that contains pointers to the real outputs. The whole structure has these fields:

- **capacity**: the capacity of the array of outputs,
- **count**: the number of outputs currently in the array,
- **data**: the array of outputs,
- **lookup_cap**: the capacity of the lookup table,
- **lookup_cnt**: the number of entries currently in the lookup,
- **lookup**: the array of lookup entries.

The sizes of data and lookup are independent to some extent, so the corresponding arrays are resized independently. Since we do not want to fill the lookup fully at any time due to efficiency reasons, we trigger its resize already at 75 % capacity. Lookup index 0 does not point to any output and the program interprets it as "no output present".

```
struct outputs *init_outputs(size_t capacity)
{
  struct outputs *ops = malloc(sizeof(struct outputs));
  if (ops ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  ops→capacity = capacity;
  ops→count = 0;
  ops→data = calloc(capacity + 1, sizeof(struct output));
  if (ops→data ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    free(ops);
    return Λ;
  }
  ops→data[0].next_op_index = 1;
  ops→lookup_cap = 2 * capacity;
  ops→lookup_cnt = 0;
  ops→lookup = calloc(2 * capacity + 1, sizeof(size_t));
  if (ops→lookup ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    free(ops→data);
    free(ops);
    return Λ;
  }
  return ops;
}
struct outputs *resize_outputs(struct outputs *ops, size_t capacity)
{
  struct output *new_data = realloc(ops→data, (capacity + 1) * sizeof(struct output));
```

```
  if (new_data ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  ops→data = new_data;
  size_t diff = capacity − ops→capacity;
  memset(ops→data + ops→capacity + 1, 0, diff ∗ sizeof(struct output));
  ops→capacity = capacity;
  return ops;
}
void destroy_outputs(struct outputs ∗ops)
{
  free(ops→data);
  free(ops→lookup);
  free(ops);
}
bool resize_lookup(struct outputs ∗ops, size_t new_cap, struct trie ∗t)
{
  size_t ∗new_lookup = calloc(new_cap + 1, sizeof(size_t));
  size_t ∗old_lookup = ops→lookup;
  if (new_lookup ≡ Λ) {
    return false;
  }
  ops→lookup = new_lookup;
  ops→lookup_cap = new_cap;
  size_t old_hash, new_hash, op_index;
  struct output op;
  for (size_t node = 0; node < t→capacity; node++) {
    if (¬is_node_occupied(t, node) ∨ t→aux[node] ≡ 0) {
      continue;
    }
    old_hash = t→aux[node];
    op_index = old_lookup[old_hash];
    op = ops→data[op_index];
    new_hash = hash_trie_output(ops, op.value, op.position, op.next_op_index);
    ops→lookup[new_hash] = op_index;
    t→aux[node] = new_hash;
  }
  free(old_lookup);
  return true;
}
```

70. hash_trie_output. Computes the value of the hash function for given output and finds a free index in the lookup that will store it. If successful, returns the hash value.

```
size_t hash_trie_output(struct outputs *ops, size_t value, size_t position, size_t next_op_index)
{
  size_t hash = ((next_op_index + 313 * position + 361 * value) % ops→lookup_cap) + 1;
  size_t op_index;
  while (true) {
    op_index = ops→lookup[hash];
    if (op_index ≡ 0) {
      return hash;
    }
    else if (ops→data[op_index].value ≡ value ∧ ops→data[op_index].position ≡
        position ∧ ops→data[op_index].next_op_index ≡ next_op_index) {
      return hash;
    }
    else if (hash > 1) {
      hash −= 1;
    }
    else {
      hash = ops→lookup_cap;
    }
  }
  return 0;
}
```

71. Pattern trie. The structure that holds already generated set of patterns. It is simply an aggregation of two other structures:

- **t**: a trie storing the patterns,
- **ops**: outputs array storing the hyphenation information.

Technically speaking, the `aux` pointer of each node that corresponds to a pattern contains the lookup index of its output.

```
struct pattern_trie *init_pattern_trie(size_t trie_capacity, size_t outputs_capacity)
{
  struct pattern_trie *pt = malloc(sizeof(struct pattern_trie));
  if (pt ≡ Λ) {
    return Λ;
  }
  pt→t = init_trie(trie_capacity);
  if (pt→t ≡ Λ) {
    free(pt);
    return Λ;
  }
  if (¬put_first_level(pt→t)) {
    free(pt→t);
    free(pt);
    return Λ;
  }
  pt→ops = init_outputs(outputs_capacity);
  if (pt→ops ≡ Λ) {
    free(pt→t);
    free(pt);
    return Λ;
  }
  return pt;
}
void destroy_pattern_trie(struct pattern_trie *pt)
{
  destroy_trie(pt→t);
  destroy_outputs(pt→ops);
  free(pt);
}
```

72. new_trie_output. Creates an entry for the given output. If successful, returns true and stores output's lookup index to `op_index`.

```
bool new_trie_output(struct pattern_trie *pt, size_t value, size_t position, size_t next_op_index, size_t
        *op_index) {
  if (pt→ops→count ≥ pt→ops→capacity − 1) {
    if (resize_outputs(pt→ops, pt→ops→capacity * 2) ≡ Λ) {
      return false;
    }
  }
  if (pt→ops→lookup_cnt * 4 > pt→ops→lookup_cap * 3) {
    if (¬resize_lookup(pt→ops, pt→ops→lookup_cap * 2, pt→t)) {
      return false;
    }
  }
  size_t hash = hash_trie_output(pt→ops, value, position, next_op_index); if (pt→ops→lookup[hash] ≡ 0)
      { pt→ops→count++; struct output new_op = { . value = value , . position = position , .
      next_op_index = next_op_index } ;
  size_t free_list_head = pt→ops→data[0].next_op_index;
  if (pt→ops→data[free_list_head].next_op_index ≡ 0) {
    pt→ops→data[0].next_op_index = pt→ops→count + 1;
  }
  else {
    pt→ops→data[0].next_op_index = pt→ops→data[free_list_head].next_op_index;
  }
  pt→ops→data[free_list_head] = new_op;
  pt→ops→lookup[hash] = free_list_head;
  pt→ops→lookup_cnt++; } *op_index = hash;
  return true; }
```

73. set_output. Creates a new output and links it to given node in the trie. Returns true if the creation and assignment finished successfully.

```
bool set_output(struct pattern_trie *pt, size_t node, size_t value, size_t position)
{
  size_t op_index;
  if (¬new_trie_output(pt, value, position, pt→ops→lookup[pt→t→aux[node]], &op_index)) {
    return false;
  }
  pt→t→aux[node] = op_index;
  return true;
}
```

74. Pattern counts. The structure that stores the numbers of supporting and contradicting occurrences for patterns. It comprises 4 fields:

- **capacity**: the maximum index that can be used,
- **size**: the highest index currently in use,
- **good**: array of supporting occurrence counts,
- **bad**: array of contradicting occurrence counts.

The counts on given index correspond to the same pattern.

```
struct pattern_counts *init_pattern_counts(size_t capacity)
{
  struct pattern_counts *pc = malloc(sizeof(struct pattern_counts));
  if (pc ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    return Λ;
  }
  pc→good = calloc(capacity, sizeof(size_t));
  pc→bad = calloc(capacity, sizeof(size_t));
  if (pc→good ≡ Λ ∨ pc→bad ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    free(pc→good);
    free(pc→bad);
    free(pc);
    return Λ;
  }
  pc→capacity = capacity;
  pc→size = 1;
  return pc;
}
struct pattern_counts *resize_pattern_counts(struct pattern_counts *pc, size_t new_capacity)
{
  size_t *new_good = realloc(pc→good, new_capacity * sizeof(size_t));
  if (new_good ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    return Λ;
  }
  pc→good = new_good;
  size_t *new_bad = realloc(pc→bad, new_capacity * sizeof(size_t));
  if (new_bad ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    return Λ;
  }
  pc→bad = new_bad;
  size_t diff = new_capacity − pc→capacity;
  memset(pc→good + pc→capacity, 0, diff * sizeof(size_t));
  memset(pc→bad + pc→capacity, 0, diff * sizeof(size_t));
  pc→capacity = new_capacity;
  return pc;
}
void destroy_pattern_counts(struct pattern_counts *pc)
{
  free(pc→good);
```

```
    free(pc→bad);
    free(pc);
}
```

75. Count trie. This structure stores candidate patterns and their occurrence counts. During dictionary file processing, every word contributes to count trie with its substrings as candidate patterns. In implementation, the count trie is composed from 2 distinct substructures:

- **t**: a trie storing the patterns,
- **cnts**: pattern counts with the numbers of occurrences.

Note that the count trie does not need any outputs, since the only pattern template that the current iteration explores is defined by `pat_len` and `pat_dot` parameters. This allows us to store pointers to pattern counts in the `aux` field of occupied trie nodes instead.

```
struct count_trie *init_count_trie(size_t trie_capacity, size_t counts_capacity)
{
  struct count_trie *ct = malloc(sizeof(struct count_trie));
  if (ct ≡ Λ) {
    return Λ;
  }
  ct→t = init_trie(trie_capacity);
  if (ct→t ≡ Λ) {
    free(ct);
    return Λ;
  }
  if (¬put_first_level(ct→t)) {
    free(ct→t);
    free(ct);
    return Λ;
  }
  ct→cnts = init_pattern_counts(counts_capacity);
  if (ct→cnts ≡ Λ) {
    free(ct→t);
    free(ct);
    return Λ;
  }
  return ct;
}
void destroy_count_trie(struct count_trie *ct)
{
  destroy_trie(ct→t);
  destroy_pattern_counts(ct→cnts);
  free(ct);
}
```

76. String buffer. We use buffer for storing text, usually lines read from input files. The structure contains following fields:

- **capacity**: the maximum length of the string that the buffer can hold (including the ending '0x00'),
- **size**: the current length of string stored in the buffer,
- **data**: the array storing the string,
- **eof**: if the buffer is used to read lines from a file, this flag indicates whether the end of file was reached..

```
struct string_buffer *init_buffer(size_t capacity)
{
  struct string_buffer *buf = malloc(sizeof(struct string_buffer));
  if (buf ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  buf→capacity = capacity;
  buf→size = 0;
  buf→data = malloc(capacity * sizeof(char));
  buf→eof = false;
  if (buf→data ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    free(buf);
    return Λ;
  }
  buf→data[0] = '\0';
  return buf;
}
struct string_buffer *resize_buffer(struct string_buffer *buf, size_t new_capacity)
{
  char *new_ptr = realloc(buf→data, new_capacity);
  if (new_ptr ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  buf→data = new_ptr;
  buf→capacity = new_capacity;
  return buf;
}
void reset_buffer(struct string_buffer *buf)
{
  buf→eof = false;
  buf→size = 0;
  buf→data[0] = '\0';
}
void destroy_buffer(struct string_buffer *buf)
{
  free(buf→data);
  free(buf);
}
```

77. append_char. Puts the given character to the current end of buffer. Note that the operation may overwrite the ending '0x00', and the caller should ensure it is not the case. Returns true upon successful write.

```
bool append_char(struct string_buffer *buf, char c)
{
  if (buf→size + 1 ≥ buf→capacity) {
    if (resize_buffer(buf, 2 * buf→capacity) ≡ Λ) {
      return false;
    }
  }
  buf→data[buf→size] = c;
  buf→size++;
  return true;
}
```

78. append_string. Copies the given string to the current end of buffer. Provided string contains the '0x00' at its end by design, and the character is copied to the buffer as well. Returns true if the copying finished successfully.

```
bool append_string(struct string_buffer *buf, const char *str)
{
  size_t len = strlen(str) + 1;
  if (buf→size + len ≥ buf→capacity) {
    if (resize_buffer(buf, 2 * (buf→size + len)) ≡ Λ) {
      return false;
    }
  }
  strcpy(&buf→data[buf→size], str);
  buf→size += len;
  return true;
}
```

79. Translate table. We use this structure to hold the information about character mapping. The translate file documents the relationships between "lowercase" letters and their "uppercase" variants, and the translate table allows easy conversion. Technically the structure is just a combination of 2 substructures:

- **mapping**: a trie that stores all the characters (both variants),
- **alphabet**: a string buffer containing all lowercase characters separated by '0x00'.

In the `aux` field of the trie nodes corresponding to a character we store index of the beginning of its lowercase representation in `alphabet`.

```
struct translate_table *init_tr_table(size_t mapping_capacity, size_t alphabet_capacity)
{
  struct translate_table *tt = malloc(sizeof(struct translate_table));
  if (tt ≡ Λ) {
    fprintf(stderr, "Allocation error\n");
    return Λ;
  }
  struct trie *mapping = init_trie(mapping_capacity);
  if (mapping ≡ Λ) {
    free(tt);
    return Λ;
  }
  if (¬put_first_level(mapping)) {
    destroy_trie(mapping);
    free(tt);
    return Λ;
  }
  struct string_buffer *alphabet = init_buffer(alphabet_capacity);
  if (alphabet ≡ Λ) {
    destroy_trie(mapping);
    free(tt);
    return Λ;
  }
  tt→mapping = mapping;
  tt→alphabet = alphabet;
  tt→index_to_alphabet = malloc(alphabet_capacity * sizeof(size_t));
  if (tt→index_to_alphabet ≡ Λ) {
    destroy_trie(tt→mapping);
    destroy_buffer(tt→alphabet);
    free(tt);
    return Λ;
  }
  tt→letter_count = 0;
  tt→letter_capacity = alphabet_capacity;
  if (¬append_char(tt→alphabet, '\0')) {
    free(tt→index_to_alphabet);
    destroy_trie(tt→mapping);
    destroy_buffer(tt→alphabet);
    free(tt);
    return Λ;
  }
  return tt;
}
```

```
void destroy_tr_table(struct translate_table *tt)
{
  destroy_trie(tt→mapping);
  destroy_buffer(tt→alphabet);
  free(tt→index_to_alphabet);
  free(tt);
}
```

80. get_lower. Returns the lowercase representation of the given letter, or NULL if the letter does not exist in the translate table.

```
char *get_lower(struct translate_table *tt, const char *letter)
{
  size_t index = traverse_trie(tt→mapping, letter);
  if (index ≡ 0 ∨ tt→mapping→aux[index] > tt→letter_count ∨ tt→mapping→aux[index] ≡ 0) {
    return Λ;
  }
  size_t alphabet_offset = tt→index_to_alphabet[tt→mapping→aux[index]];
  if (alphabet_offset ≥ tt→alphabet→size) {
    return Λ;
  }
  return tt→alphabet→data + alphabet_offset;
}
```

81. get_letter_index. Returns the index of the given letter in the translate table, or 0 if the letter does not exist.

```
size_t get_letter_index(struct translate_table *tt, char *letter)
{
  size_t index = traverse_trie(tt→mapping, letter);
  if (index ≡ 0 ∨ tt→mapping→aux[index] > tt→letter_count ∨ tt→mapping→aux[index] ≡ 0) {
    return 0;
  }
  return tt→mapping→aux[index];
}
```

82. convert_index. Converts a letter index to a byte sequence and appends it to the word. While the index is greater than 254, appends 0xFF byte to the word and subtracts 254 from the index. Finally appends the byte with the same value as the remainder.

```
bool convert_index(size_t index, struct word *word)
{
  if (index ≡ 0) {
    return true;
  }
  size_t ff_count = (index − 1)/254;
  size_t remainder = ((index − 1) % 254) + 1;
  for (size_t i = 0; i < ff_count; i++) {
    if (¬append_char_to_word(word, (char) #ff)) {
      return false;
    }
  }
  if (¬append_char_to_word(word, (char) remainder)) {
    return false;
  }
  return true;
}
```

83. convert_byte_sequence. Converts a byte sequence to a translate table index. This is the reverse operation of convert_index.

```
size_t convert_byte_sequence(char **sequence)
{
  if (sequence ≡ Λ ∨ *sequence ≡ Λ) {
    return 0;
  }
  size_t ff_count = 0;
  while (**sequence ≡ (char) #ff) {
    ff_count++;
    (*sequence)++;
  }
  size_t remainder = (uint8_t) **sequence;
  (*sequence)++;
  return ff_count * 254 + remainder;
}
```

84. Params. This structure encompasses all the parameters that influence the computation of `utfpatgen`. We can sort them by their scope into **global** (usually defined once and used throughout the whole run), **level-specific** (defined specifically for each hyphenation level), and **pass-specific** (defined specifically for each iteration within a level):

- **left_hyphen_min** (global): the length of prefix of a word that should not be hyphenated,
- **right_hyphen_min** (global): the length of suffix of a word that should not be hyphenated,
- **bad_hyphen** (global): the symbol representing incorrectly placed hyphen in the dictionary,
- **missed_hyphen** (global): the symbol representing missing hyphen in the dictionary,
- **good_hyphen** (global): the symbol representing correctly placed hyphen in the dictionary,
- **hyph_start** (global): the lowest hyphenation level to be generated,
- **hyph_finish** (global): the highest hyphenation level to be generated,
- **word_weight** (global): the default weight of a hyphen in a word from dictionary,
- **dictionary_file** (global): a pointer to opened file with dictionary entries,
- **pattern_file** (global): a pointer to opened file with initial set of patterns,
- **output_file** (global): a pointer to opened file where the final set of patterns will be written,
- **translate_file** (global): a pointer to opened file with character mapping and language-specific parameters,
- **hyph_level** (level-specific): the current hyphenation level,
- **pat_start** (level-specific): the shortest pattern length,
- **pat_finish** (level-specific): the longest pattern length,
- **good_wt** (level-specific): the supporting occurrence weight,
- **bad_wt** (level-specific): the contradicting occurrence weight,
- **thresh** (level-specific): the pattern acceptance threshold,
- **good_dot** (level-specific): the hyphen type considered as correct,
- **bad_dot** (level-specific): the hyphen type considered as incorrect,
- **pat_len** (pass-specific): the current pattern length,
- **pat_dot** (pass-specific): the current dot position.

```
struct params *init_params( )
{
  struct params *p = malloc(sizeof(struct params));
  if (p ≡ Λ) {
    fputs("Allocation␣error\n", stderr);
    return Λ;
  }
  p→left_hyphen_min = 2;
  p→right_hyphen_min = 3;
  p→bad_hyphen = '.';
  p→missed_hyphen = '-';
  p→good_hyphen = '*';
  p→word_weight = 1;
  p→dictionary_file = Λ;
  p→pattern_file = Λ;
  p→output_file = Λ;
  p→translate_file = Λ;
  return p;
}
void reset_params(struct params *p)
{
  p→left_hyphen_min = 2;
  p→right_hyphen_min = 3;
  p→bad_hyphen = '.';
  p→missed_hyphen = '-';
```

```
  p→good_hyphen = '*';
}
void destroy_params(struct params *p)
{
  if (p→dictionary_file ≠ Λ) {
    fclose(p→dictionary_file);
  }
  if (p→pattern_file ≠ Λ) {
    fclose(p→pattern_file);
  }
  if (p→output_file ≠ Λ) {
    fclose(p→output_file);
  }
  if (p→translate_file ≠ Λ) {
    fclose(p→translate_file);
  }
  free(p);
}
```

85. Pass stats. This structure groups together variables with mostly statistical meaning. We use them to map the progress and quality of the pattern generation, and present it on standard output. Following fields are monitored:

- **good_pat_cnt**: the number of new good patterns from an iteration,
- **bad_pat_cnt**: the number of new bad patterns from an iteration,
- **good_cnt**: the number of hyphens from the dictionary correctly identified with current set of patterns,
- **bad_cnt**: the number of hyphens from the dictionary incorrectly identified with current set of patterns,
- **miss_cnt**: the number of hyphens from the dictionary missed with current set of patterns,
- **level_pattern_cnt**: the number of patterns generated on current hyphenation level,
- **max_level**: the highest hyphenation level found in the initial patterns,
- **more_to_come**: a flag indicating that undecided patterns were found on current hyphenation level.

86. Stack. We decided to implement a stack-like structure to break the recursive functions from `patgen`. In most cases the stack holds data related to previous iterations of a method, for instance, the trie bases on the path to current node. There are only 3 fields:

- **capacity**: the highest index that can be accessed,
- **top**: the index where the last value was inserted,
- **data**: the array of values currently on the stack.

```
struct stack *init_stack(size_t capacity)
{
  struct stack *s = malloc(sizeof(struct stack));
  if (s ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    return Λ;
  }
  s→data = malloc(capacity * sizeof(size_t));
  if (s→data ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    free(s→data);
    free(s);
    return Λ;
  }
  s→capacity = capacity;
  s→top = 0;
  return s;
}
struct stack *resize_stack(struct stack *s, size_t new_capacity)
{
  size_t *new_stack = realloc(s→data, new_capacity * sizeof(size_t));
  if (new_stack ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    destroy_stack(s);
    return Λ;
  }
  s→data = new_stack;
  s→capacity = new_capacity;
  return s;
}
void destroy_stack(struct stack *s)
{
  free(s→data);
  free(s);
}
```

87. put_on_stack. Appends the given value to the top of the stack. Returns true if the insertion finishes successfully.

```
bool put_on_stack(struct stack *s, size_t value)
{
  if (s→top ≥ s→capacity) {
    size_t new_capacity = 2 * (s→top);
    if (resize_stack(s, new_capacity) ≡ Λ) {
      return false;
    }
  }
  s→data[s→top] = value;
  s→top++;
  return true;
}
```

88. get_top_value. Returns the value currently at `top` index. Does not change anything on the stack.

```
size_t get_top_value(struct stack *s)
{
  if (s→top ≡ 0) {
    return 0;
  }
  return s→data[s→top − 1];
}
```

89. set_top_value. Changes the value at `top` index to the given one.

```
void set_top_value(struct stack *s, size_t value)
{
  if (s→top ≡ 0) {
    return;
  }
  s→data[s→top − 1] = value;
}
```

90. Word. The structure that holds information about a word and its hyphens. It consists of 7 fields:

- **capacity**: the maximum length of the word in bytes,
- **size**: current length of the word in bytes,
- **length**: current length of the word in characters,
- **lowercase**: the word translated to lowercase characters according to the translate table, and surrounded by `EDGE_OF_WORD` symbols,
- **true_hyphens**: array of hyphen types and their weights retrieved from the dictionary,
- **found_hyphens**: array of hyphenation level identified by current set of patterns,
- **no_more**: array of flags marking the dot positions that can be skipped during processing.

Array index n corresponds to the dot position between characters n and $n+1$. The lowest two bits of a `true_hyphens` value describe the hyphen type, the rest represent its weight (multiplied by 4).

Note that the `lowercase` array is not strictly a string as we do not require the closing '0x00'.

```
struct word *init_word(size_t capacity)
{
  struct word *word = malloc(sizeof(struct word));
  if (word ≡ Λ) {
    return Λ;
  }
  word→translated = calloc(capacity, sizeof(char));
  if (word→translated ≡ Λ) {
    free(word);
    return Λ;
  }
  word→true_hyphens = calloc(capacity, sizeof(size_t));
  if (word→true_hyphens ≡ Λ) {
    free(word→translated);
    free(word);
    return Λ;
  }
  word→found_hyphens = calloc(capacity, sizeof (uint8_t));
  if (word→found_hyphens ≡ Λ) {
    free(word→translated);
    free(word→true_hyphens);
    free(word);
    return Λ;
  }
  word→no_more = calloc(capacity, sizeof(bool));
  if (word→no_more ≡ Λ) {
    free(word→translated);
    free(word→true_hyphens);
    free(word→found_hyphens);
    free(word);
    return Λ;
  }
  word→size = 0;
  word→length = 0;
  word→capacity = capacity;
  return word;
}
struct word *resize_word(struct word *word, size_t new_capacity)
{
```

```
  char *new_translated = realloc(word→translated, new_capacity * sizeof(char));
  if (new_translated ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    return Λ;
  }
  word→translated = new_translated;
  size_t *new_true_hyphens = realloc(word→true_hyphens, new_capacity * sizeof(size_t));
  if (new_true_hyphens ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    return Λ;
  }
  word→true_hyphens = new_true_hyphens;
  uint8_t * new_found_hyphens = realloc(word→found_hyphens, new_capacity * sizeof (uint8_t));
  if (new_found_hyphens ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    return Λ;
  }
  word→found_hyphens = new_found_hyphens;
  bool *new_no_more = realloc(word→no_more, new_capacity * sizeof(bool));
  if (new_no_more ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    return Λ;
  }
  word→no_more = new_no_more;
  size_t diff = new_capacity − word→capacity;
  memset(word→translated + word→capacity, '\0', diff * sizeof(char));
  memset(word→true_hyphens + word→capacity, 0, diff * sizeof(size_t));
  memset(word→found_hyphens + word→capacity, 0, diff * sizeof (uint8_t));
  memset(word→no_more + word→capacity, false, diff * sizeof(bool));
  word→capacity = new_capacity;
  return word;
}
void reset_word(struct word *word)
{
  word→length = 0;
  word→size = 0;
  memset(word→translated, 0, word→capacity * sizeof(char));
  memset(word→true_hyphens, 0, word→capacity * sizeof(size_t));
  memset(word→found_hyphens, 0, word→capacity * sizeof (uint8_t));
  memset(word→no_more, false, word→capacity * sizeof(bool));
}
void destroy_word(struct word *word)
{
  free(word→translated);
  free(word→true_hyphens);
  free(word→found_hyphens);
  free(word→no_more);
  free(word);
}
size_t get_true_hyphen(struct word *word, size_t index)
```

```
{
  if (index ≥ word→size) {
    return 0;
  }
  return word→true_hyphens[index];
}
bool set_true_hyphen(struct word *word, size_t index, size_t value)
{
  if (index ≥ word→size) {
    return false;
  }
  word→true_hyphens[index] = value;
  return true;
}
uint8_t get_found_hyphen(struct word *word, size_t index)
{
  if (index ≥ word→size) {
    return 0;
  }
  return word→found_hyphens[index];
}
bool set_found_hyphen(struct word *word, size_t index, uint8_t value)
{
  if (index ≥ word→size) {
    return false;
  }
  word→found_hyphens[index] = value;
  return true;
}
bool get_no_more(struct word *word, size_t index)
{
  if (index ≥ word→size) {
    return false;
  }
  return word→no_more[index];
}
bool set_no_more(struct word *word, size_t index, bool value)
{
  if (index ≥ word→size) {
    return false;
  }
  word→no_more[index] = value;
  return true;
}
```

91. append_char_to_word. Puts the given byte to the end of word. Returns true if the insertion was successful.

```
bool append_char_to_word(struct word *word, char c)
{
  if (word→size ≥ word→capacity − 1) {
    if (¬resize_word(word, 2 * word→capacity)) {
      return false;
    }
  }
  word→translated[word→size] = c;
  word→size++;
  if ((uint8_t)c ≠ #ff) {
    word→length++;
  }
  return true;
}
```

92. Pattern. This structure's purpose is to hold information about a pattern parsed from the pattern file. It has similar fields as `word`, but is stripped of the arrays it does not need:

- **capacity**: the maximum length of the pattern in bytes,
- **size**: current length of the pattern in bytes,
- **length**: current length of the pattern in characters,
- **text**: the pattern translated to lowercase characters according to the translate table,
- **hyphens**: array of hyphenation levels of the pattern.

```
struct pattern *init_pattern(size_t capacity)
{
  struct pattern *pat = malloc(sizeof(struct pattern));
  if (pat ≡ Λ) {
    return Λ;
  }
  pat→text = calloc(capacity, sizeof(char));
  if (pat→text ≡ Λ) {
    free(pat);
    return Λ;
  }
  pat→hyphens = calloc(capacity, sizeof (uint8_t));
  if (pat→hyphens ≡ Λ) {
    free(pat→text);
    free(pat);
    return Λ;
  }
  pat→length = 0;
  pat→size = 0;
  pat→capacity = capacity;
  return pat;
}
struct pattern *resize_pattern(struct pattern *pat, size_t new_capacity)
{
  char *new_text = realloc(pat→text, new_capacity * sizeof(char));
  uint8_t * new_hyphens = realloc(pat→hyphens, new_capacity * sizeof (uint8_t));
  if (new_text ≡ Λ ∨ new_hyphens ≡ Λ) {
    fprintf(stderr, "Allocation␣error\n");
    return Λ;
  }
  pat→text = new_text;
  pat→hyphens = new_hyphens;
  memset(pat→text + pat→capacity, '\0', (new_capacity − pat→capacity) * sizeof(char));
  memset(pat→hyphens + pat→capacity, 0, (new_capacity − pat→capacity) * sizeof (uint8_t));
  pat→capacity = new_capacity;
  return pat;
}
void reset_pattern(struct pattern *pat)
{
  pat→length = 0;
  pat→size = 0;
  memset(pat→text, '\0', pat→capacity * sizeof(char));
  memset(pat→hyphens, 0, pat→capacity * sizeof (uint8_t));
}
```

```
void destroy_pattern(struct pattern *pat)
{
  free(pat→text);
  free(pat→hyphens);
  free(pat);
}
uint8_t get_hyphen(struct pattern *pat, size_t index)
{
  if (index ≥ pat→capacity) {
    return 0;
  }
  return pat→hyphens[index];
}
bool set_hyphen(struct pattern *pat, size_t index, uint8_t value)
{
  if (index ≥ pat→capacity) {
    return false;
  }
  pat→hyphens[index] = value;
  return true;
}
```

93. convert_index_to_pattern. Converts a letter index to a byte sequence and appends it to the pattern. While the index is greater than 254, appends 0xFF byte to the pattern and subtracts 254 from the index. Finally appends the byte with the same value as the remainder.

```
bool convert_index_to_pattern(size_t index, struct pattern *pat)
{
  if (index ≡ 0) return true;
  size_t ff_count = (index − 1)/254;
  size_t remainder = ((index − 1) % 254) + 1;
  size_t total_bytes = ff_count + 1;
  if (pat→size ≥ pat→capacity − total_bytes) {
    if (¬resize_pattern(pat, 2 * (pat→capacity + total_bytes))) {
      return false;
    }
  }
  for (size_t i = 0; i < ff_count; i++) {
    pat→text[pat→size] = (char) #ff;
    pat→size ++;
  }
  pat→text[pat→size] = (char) remainder;
  pat→size ++;
  return true;
}
```

94. Index. List of used identifiers

alphabet: 19, 24, 25, 46, 52, 79, 80.
alphabet_capacity: 79.
alphabet_offset: 80.
append_char: 24, 28, 33, 38, 45, 77, 79.
append_char_to_word: 82, 91.
append_string: 19, 24, 25, 78.
argc: 5, 6, 14.
argv: 5, 6, 14.
aux: 19, 24, 25, 35, 38, 41, 42, 43, 45, 49, 56, 57, 59, 60, 63, 64, 66, 69, 73, 80, 81.
aux_dot: 10.
bad: 35, 38, 74.
bad_cnt: 31, 34, 37, 38, 50.
bad_dot: 31, 35.
bad_eff: 37.
BAD_HYF: 31, 33.
bad_hyphen: 23, 33, 52, 84.
BAD_OP_VALUE: 38, 40, 43, 47, 49.
bad_pat_cnt: 37, 38.
bad_wt: 8, 9, 38.
base: 49, 63, 64, 65, 66, 68.
base_max: 56, 57.
bit_index: 59.
buf: 15, 19, 22, 23, 24, 27, 28, 32, 33, 51, 76, 77, 78.
byte: 54.
byte_index: 59.
c: 11, 15, 20, 21, 24, 25, 28, 33, 77, 91.
calloc: 56, 69, 74, 90, 92.
capacity: 15, 35, 40, 56, 57, 58, 59, 63, 64, 67, 69, 72, 74, 76, 77, 78, 86, 87, 90, 91, 92, 93.
char: 19, 25, 46, 52.
child_freed: 41.
cnt_index: 35, 38.
cnts: 35, 38, 75.
collect_count_trie: 31, 37.
conflict: 67.
convert_byte_sequence: 46, 52, 83.
convert_index: 33, 82.
convert_index_to_pattern: 28, 93.
copy_node: 59, 65, 66.
count: 27, 37, 40, 69, 72.
count_dots: 32, 34, 51.
count_trie: 31, 32, 35, 37, 38, 75.
counts_capacity: 75.
ct: 31, 32, 35, 37, 38, 75.
current_index: 34, 35, 49.
current_len: 29, 38.
current_pos: 34, 35, 49.
c1: 22.
c2: 22.
data: 15, 22, 23, 24, 28, 33, 38, 40, 43, 45, 46, 47, 49, 52, 69, 70, 72, 76, 77, 78, 80, 86, 87, 88, 89.
deallocate_node: 41, 42, 65.
default_ascii_mapping: 19, 25.
default_mapping: 19.
delete_bad_patterns: 8, 40.
delete_patterns: 40, 41.
destroy_buffer: 19, 24, 27, 28, 32, 33, 38, 45, 51, 76, 79.
destroy_count_trie: 31, 75.
destroy_outputs: 69, 71.
destroy_params: 5, 6, 8, 10, 11, 84.
destroy_pattern: 27, 92.
destroy_pattern_counts: 74, 75.
destroy_pattern_trie: 5, 6, 8, 10, 11, 71.
destroy_stack: 38, 41, 45, 86.
destroy_tr_table: 5, 6, 8, 10, 11, 79.
destroy_trie: 19, 27, 32, 38, 57, 71, 75, 79.
destroy_word: 32, 51, 90.
dictionary_file: 14, 32, 51, 84.
diff: 69, 74, 90.
dot_index: 34, 35, 49.
dot_len: 32.
dot_max: 32, 35.
dot_min: 32, 35.
dot_pos: 34, 35, 49, 52.
EDGE_OF_WORD: 19, 25, 28, 33, 46, 52.
edge_of_word: 33.
edge_of_word_idx: 46, 52.
EMPTY_OP_VALUE: 40.
end: 63.
end_index: 35, 49.
end_pos: 35, 49.
EOF: 15.
eof: 15, 19, 27, 32, 51, 76.
EXIT_FAILURE: 6, 8, 10, 11.
EXIT_SUCCESS: 5.
false: 10, 11, 14, 15, 19, 22, 23, 24, 25, 27, 28, 29, 31, 32, 33, 35, 37, 38, 40, 41, 45, 49, 50, 51, 57, 59, 63, 64, 65, 66, 67, 69, 72, 73, 76, 77, 78, 82, 87, 90, 91, 92, 93.
fclose: 14, 50, 84.
ff_count: 82, 83, 93.
fgetc: 15.
filename: 50.
find_base_for_first_fit: 66, 67.
first_fit: 63, 64, 66.
first_line: 19.
fit: 63, 64.
fopen: 14, 50.
found_hyphen: 52.

found_hyphens: 90.
fprintf: 14, 24, 28, 33, 34, 46, 52, 74, 79, 86, 90, 92.
fputc: 46, 52.
fputs: 56, 57, 69, 76, 84.
free: 50, 56, 57, 69, 71, 74, 75, 76, 79, 84, 86, 90, 92.
free_list_head: 43, 72.
from: 59, 60, 65.
from_free: 59.
from_index: 59, 65.
get_base_used: 59, 67.
get_found_hyphen: 34, 35, 49, 52, 90.
get_highest_level: 46, 47.
get_hyphen: 29, 92.
get_letter_index: 28, 33, 46, 52, 81.
get_lower: 80.
get_no_more: 35, 90.
get_top_value: 38, 41, 45, 88.
get_true_hyphen: 34, 35, 52, 90.
getchar: 7, 9.
good: 35, 38, 74.
good_cnt: 31, 34, 37, 38, 50.
good_dot: 31, 35.
`GOOD_HYF`: 31, 33.
good_hyphen: 23, 33, 52, 84.
good_pat_cnt: 37, 38.
good_pattern: 35.
good_wt: 8, 9, 37, 38.
h: 40, 43.
has_hyphen: 52.
has_weight: 33.
hash: 70, 72.
hash_trie_output: 69, 70, 72.
helper_trie: 19, 24, 25, 27, 29, 32, 35, 38, 62, 63.
highest: 47.
hyf: 33, 34, 35.
hyph_finish: 7, 8.
hyph_level: 8, 31, 38, 49, 50.
hyph_start: 7, 8.
`HYPHEN_FLAG`: 28, 33.
hyphen_type: 33, 34, 35.
hyphenate_all_words: 50, 51.
hyphenate_dictionary: 11, 50.
hyphenate_word: 32, 49, 51.
hyphenation_value: 29, 34.
hyphens: 92.
i: 5, 8, 10, 24, 28, 29, 33, 49, 57, 65, 67, 82, 93.
index: 59, 61, 63, 68, 80, 81, 82, 90, 92, 93.
index_to_alphabet: 19, 24, 25, 46, 52, 79, 80.
init_buffer: 19, 24, 27, 28, 32, 33, 38, 45, 51, 76, 79.
init_count_trie: 31, 75.
init_outputs: 69, 71.
init_params: 6, 84.
init_pattern: 27, 92.
init_pattern_counts: 74, 75.
init_pattern_trie: 6, 71.
init_stack: 38, 41, 45, 86.
init_tr_table: 6, 79.
init_trie: 19, 27, 32, 38, 56, 71, 75, 79.
init_word: 32, 51, 90.
insert_new_pattern: 27, 29.
insert_pattern: 19, 24, 25, 29, 62.
insert_substring: 35, 38, 62, 63.
int8_t: 22, 23.
is_integer: 20, 22.
is_node_occupied: 58, 61, 67, 69.
is_space: 21, 22, 23.
is_utf_start_byte: 28, 33, 54.
j: 10.
last_free: 58.
left_hyphen_min: 19, 23, 32, 34, 35, 52, 84.
len: 78.
length: 32, 34, 35, 49, 51, 52, 62, 63, 90, 91, 92.
letter: 24, 28, 33, 80, 81.
letter_capacity: 19, 24, 25, 79.
letter_count: 19, 24, 25, 79, 80, 81.
letter_index: 19, 24, 25, 28, 33, 46, 52.
level: 46.
level_pattern_cnt: 8, 27, 29, 37.
link_around_bad_outputs: 41, 43.
links: 38, 41, 42, 45, 49, 56, 57, 59, 60, 63, 64, 66, 67, 68.
lookup: 43, 47, 49, 69, 70, 72, 73.
lookup_cap: 69, 70, 72.
lookup_cnt: 43, 69, 72.
lookup_index: 43.
lower: 24.
main: 5.
malloc: 50, 56, 69, 71, 74, 75, 76, 79, 84, 86, 90, 92.
mapping: 19, 24, 25, 79, 80, 81.
mapping_capacity: 79.
max_level: 7, 8, 27, 29.
max_target_index: 67.
memset: 57, 69, 74, 90, 92.
miss_cnt: 31, 34, 50.
`MISS_HYF`: 31, 33.
missed_hyphen: 23, 33, 52, 84.
more_this_level: 10.
more_to_come: 10, 37, 38.
n: 43.
n_bytes: 57.
new_array: 19, 24, 25.
new_aux: 57.
new_bad: 74.

new_cap: 69.
new_capacity: 19, 24, 25, 35, 57, 59, 63, 64, 67, 74, 76, 86, 87, 90, 92.
new_data: 69.
new_found_hyphens: 90.
new_good: 74.
new_hash: 69.
new_hyphens: 92.
new_links: 57.
new_lookup: 69.
new_no_more: 90.
new_nodes: 57.
new_op: 72.
new_pattern: 63.
new_ptr: 76.
new_stack: 86.
new_taken: 57.
new_taken_bytes: 57.
new_text: 92.
new_translated: 90.
new_trie_output: 72, 73.
new_true_hyphens: 90.
next_hyphen: 28.
next_op_index: 40, 43, 47, 49, 69, 70, 72.
NO_HYF: 31, 33.
no_more: 90.
node: 29, 35, 38, 41, 45, 49, 58, 63, 64, 68, 69, 73.
node_max: 27, 31, 37, 56, 57, 63, 64, 65, 66, 67.
node_prev: 63.
nodes: 38, 41, 42, 45, 49, 56, 57, 59, 61, 63, 64, 65, 66, 67, 68.
occupied: 27, 31, 37, 40, 42, 56, 57, 59, 63.
odd_level: 34.
offset: 67.
old_hash: 69.
old_head: 42.
old_lookup: 69.
old_op_cnt: 40.
old_taken_bytes: 57.
old_trie_cnt: 40.
op: 47, 49, 69.
op_index: 38, 43, 46, 47, 49, 69, 70, 72, 73.
ops: 27, 37, 40, 43, 45, 46, 47, 49, 69, 70, 71, 72, 73.
out: 22.
out_base: 66, 67.
out_index: 19, 24, 25.
out_op_index: 62, 63.
out_pattern: 28.
out_word: 33.
output: 11, 47, 49, 50, 69, 72.
output_file: 11, 14, 45, 46, 84.
output_hyphenated_word: 51, 52.
output_pattern: 45, 46.
output_patterns: 11, 45.
outputs: 46, 47, 69, 70.
outputs_capacity: 71.
p: 84.
params: 5, 6, 7, 8, 9, 10, 11, 14, 19, 23, 27, 31, 32, 33, 34, 35, 37, 38, 49, 50, 51, 52, 84.
parent_c: 41.
parent_node: 41.
parent_root: 41.
parse_header: 19, 23.
parse_input: 6, 14.
parse_letters: 19, 24.
parse_pattern: 27, 28.
parse_two_digit: 22, 23.
parse_word: 32, 33, 51.
pass_stats: 6, 27, 29, 31, 32, 34, 37, 38, 50, 51.
pat: 27, 29, 92, 93.
pat_dot: 10, 31, 32, 35, 38, 49.
pat_finish: 8, 9, 10.
pat_len: 10, 31, 32, 35, 38, 49.
pat_start: 8, 9, 10.
pattern: 27, 28, 29, 38, 45, 46, 62, 63, 68, 92, 93.
pattern_count: 31, 56, 63.
pattern_counts: 74.
pattern_file: 14, 27, 84.
pattern_position: 46.
pattern_trie: 6, 27, 29, 31, 32, 37, 38, 40, 41, 43, 45, 49, 50, 51, 71, 72, 73.
pattmp: 50, 51, 52.
pc: 74.
pos: 22.
position: 47, 49, 69, 70, 72, 73.
print_help: 5, 16.
print_version: 5, 17.
printf: 7, 8, 9, 11, 16, 17, 19, 27, 31, 37, 40, 50, 67.
process_all_words: 31, 32.
process_dictionary: 10, 31.
process_word: 32, 35.
ps: 6, 7, 8, 10, 11, 27, 29, 31, 32, 34, 37, 38, 50, 51.
pt: 5, 6, 8, 10, 11, 27, 29, 31, 32, 37, 38, 40, 41, 43, 45, 49, 50, 51, 71, 72, 73.
put_first_level: 57, 71, 75, 79.
put_on_stack: 38, 41, 45, 87.
q: 64, 66, 67.
q_index: 66, 67.
read_line: 15, 19, 27, 32, 51.
read_patterns: 6, 27.
read_translate: 6, 19.
realloc: 19, 24, 25, 57, 69, 74, 76, 86, 90, 92.
relink_trie: 56, 57, 58.
remainder: 82, 83, 93.

repack: 63, 64.
reset_buffer: 15, 19, 24, 28, 33, 76.
reset_params: 84.
reset_pattern: 28, 92.
reset_word: 33, 90.
resize_buffer: 15, 76, 77, 78.
resize_lookup: 69, 72.
resize_outputs: 69, 72.
resize_pattern: 92, 93.
resize_pattern_counts: 35, 74.
resize_stack: 86, 87.
resize_trie: 57, 59, 63, 64, 67.
resize_word: 90, 91.
result: 7, 9.
rewind: 19, 32, 51.
right_hyphen_min: 19, 23, 32, 34, 35, 49, 52, 84.
root: 38, 41, 45, 57.
s: 86, 87, 88, 89.
s_base: 38, 41, 45.
s_freed: 41.
s_offset: 41.
scanf: 7, 9, 11.
separator: 24.
sequence: 83.
set_base_used: 41, 56, 57, 59, 65, 66.
set_found_hyphen: 49, 90.
set_hyphen: 28, 92.
set_links: 57, 58, 60, 63, 66.
set_no_more: 49, 90.
set_output: 29, 38, 73.
set_top_value: 41, 89.
set_true_hyphen: 33, 90.
size: 15, 19, 22, 23, 24, 25, 28, 29, 33, 35, 38, 45, 46, 52, 74, 76, 77, 78, 80, 90, 91, 92, 93.
sprintf: 50.
stack: 38, 41, 45, 86, 87, 88, 89.
start_index: 35, 47, 49.
start_pos: 35, 49.
stderr: 14, 24, 28, 33, 34, 56, 57, 69, 74, 76, 79, 84, 86, 90, 92.
str: 78.
strcmp: 5.
strcpy: 78.
stream: 15.
string_buffer: 15, 19, 22, 23, 24, 27, 28, 32, 33, 38, 45, 46, 51, 76, 77, 78, 79.
strlen: 62, 68, 78.
t: 42, 56, 57, 58, 59, 60, 61, 62, 63, 64, 66, 67, 68, 69.
t_index: 42, 43, 66, 67.
taken: 56, 57, 59.
TEST: 5.
text: 29, 92, 93.
thresh: 8, 9, 37, 38.
to: 59, 60, 65.
to_free: 59.
to_index: 59.
top: 38, 41, 45, 86, 87, 88, 89.
total_bytes: 93.
translate_file: 14, 19, 84.
translate_table: 6, 19, 24, 25, 27, 28, 31, 32, 33, 45, 46, 50, 51, 52, 79, 80, 81.
translated: 34, 35, 49, 52, 90, 91.
traverse_count_trie: 37, 38.
traverse_trie: 68, 80, 81.
trie: 19, 24, 25, 27, 29, 32, 35, 38, 42, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 79.
trie_capacity: 71, 75.
true: 7, 9, 10, 11, 14, 15, 19, 22, 23, 24, 25, 27, 28, 29, 31, 32, 33, 35, 37, 38, 40, 41, 43, 45, 49, 50, 51, 56, 57, 59, 63, 64, 65, 66, 67, 69, 70, 72, 73, 77, 78, 82, 87, 90, 91, 92, 93.
true_hyphens: 90.
tt: 5, 6, 8, 10, 11, 19, 24, 25, 27, 28, 31, 32, 33, 45, 46, 50, 51, 52, 79, 80, 81.
uint8_t: 7, 9, 10, 28, 29, 32, 33, 34, 35, 38, 41, 45, 46, 49, 54, 57, 63, 64, 65, 66, 67, 68, 83, 90, 91, 92.
unpack: 64, 65.
upper: 25.
used: 59.
UTFPATGEN_VERSION: 17.
val: 23.
value: 40, 43, 47, 49, 64, 69, 70, 72, 73, 87, 89, 90, 92.
weight: 33, 34, 35, 52.
word: 32, 33, 34, 35, 49, 51, 52, 82, 90, 91.
word_index: 46, 52.
word_weight: 31, 33, 35, 50, 52, 84.

⟨ Final pass 11 ⟩ Used in section 5.
⟨ Initialization sequence 6 ⟩ Used in section 5.
⟨ Level generation 10 ⟩ Used in section 8.
⟨ Level hyperparameters input 9 ⟩ Used in section 8.
⟨ Level range specification 7 ⟩ Used in section 5.
⟨ Library includes 4 ⟩ Used in section 5.
⟨ Pattern generation 8 ⟩ Used in section 5.

UTFpatgen

	Section	Page
Introduction	1	1
Terminology	2	2
Algorithm overview	3	3
Dependencies	4	4
Main method	5	5
Initialization sequence	6	6
Level range specification	7	7
Pattern generation loop	8	8
Level hyperparameters input	9	9
Level generation	10	10
Final pass	11	11
Implementation details	12	12
IO procedures	13	13
Translate file processing	18	15
Pattern file processing	26	23
Dictionary file processing	30	28
Pattern collection	36	37
Pattern pruning	39	41
Pattern output	44	45
Hyphenation	48	48
UTF-8 specifics	53	54
Structures	55	55
Trie	56	56
Outputs	69	66
Pattern trie	71	69
Pattern counts	74	71
Count trie	75	73
String buffer	76	74
Translate table	79	76
Params	84	79
Pass stats	85	81
Stack	86	82
Word	90	84
Pattern	92	88
Index	94	90