December 4, 2001 at 18:25

1. Introduction. This is OPATGEN, word hyphenation generator.

This program takes a list of hyphenated words and creates set of hyphenation patterns which can be used by the TEX paragraph breaking algorithm. This is a complete reimplementation of Frank Liang's PATGEN generator in order to be able to handle UNICODE and to remove the restrictions of that program.

For user information of the program see the user manual.

This program is written in ANSI C++ using the standard template library. Written and tested on Linux with glibc-2.2.2 and gcc-2.96. This program should work with any compiler supporting the STL and ANSI C++.

This program uses the PATLIB library and shares its license, coding style, author, and maintainer.

Written and maintained by David Antoš, `xantos (at) fi.muni.cz`

The following two strings define the version number (to be changed whenever the program changes) and the CVS identification string for the source file.

```
const char *opatgen_version = "1.0";
const char *opatgen_cvs_id = "$Id: opatgen.w,v 1.24 2001/12/03 17:51:13 antos Exp $";
```

2. Organization of the code. The code is highly templatized and consists of following main parts. First we prepare methods we want to use in the translate file, the translate file follows and last the input and output file reading and writing services are provided. The *main* function follows after a plethora of type definitions.

All the services are put into one file (because of the templates we can't compile separately, though).

3. The *utf_8* global variable controls if we use UNICODE or 8-bit ASCII to deal with input and output. It is set in *main*.

A note on exception handling. We reuse the PATLIB's exception class, it means that if error occurs we throw the **Patlib_error**. In *main* this has it's **catch** sections.

```
format iterator int
format const_iterator int
format Patlib_error int
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <string>
#include <fstream>
#include "ptl_exc.h"
#include "ptl_gen.h"
#include "ptl_vers.h"
using namespace std;
bool utf_8;
```

4. Services for translate. We want to store the mapping from external data representation to internal alphabet (**Hword** in fact) into the word manipulator. We overload the accessing one-external-symbol fields of the manipulator as we do not want to build vectors to store one symbol.

For the template conditions and other information on word manipulator see the definition of the class parent.

```
format Tpm_pointer int
format Tin_alph int
format Tout_information int
format IO_word_manipulator int
format Trie_pattern_manipulator int
template⟨class Tpm_pointer, class Tin_alph, class Tout_information⟩
class IO_word_manipulator :
public Trie_pattern_manipulator⟨Tpm_pointer, Tin_alph, Tout_information⟩ {
  ⟨ IO word manipulator: constructor 5 ⟩
  ⟨ IO word manipulator: hard insert pattern 6 ⟩
  ⟨ IO word manipulator: word output 7 ⟩
};
```

5. Constructor simply calls the parent. See **Trie_pattern_manipulator** for parameters.

```
⟨ IO word manipulator: constructor 5 ⟩ ≡
public:
  IO_word_manipulator(const Tin_alph &max_i_a, const Tout_information &out_i_z, const
        unsigned &q_thr = 3)
  : Trie_pattern_manipulator⟨Tpm_pointer, Tin_alph, Tout_information⟩(max_i_a, out_i_z, q_thr)
      { }
```

This code is used in section 4.

6. The usual hard pattern inserting uses vector. It is not always needed for the translate service, so we overload the method to be able to handle mere values only. We have to provide the interface of the original method here too, as we do not redefine but overload!

Now we do it using brute force. FIXME: to be optimized later.

```
⟨ IO word manipulator: hard insert pattern 6 ⟩ ≡
  inline void hard_insert_pattern(const vector⟨Tin_alph⟩ &w, const Tout_information &o)
  {    /* Call the parent */
    Trie_pattern_manipulator⟨Tpm_pointer, Tin_alph, Tout_information⟩::hard_insert_pattern(w,
        o);
  }
  void hard_insert_pattern(const Tin_alph &w, const Tout_information &o)
  {
    vector⟨Tin_alph⟩ vec;
    vec.push_back(w);      /* FIXME: to be optimized */
    Trie_pattern_manipulator⟨Tpm_pointer, Tin_alph,
        Tout_information⟩::hard_insert_pattern(vec, o);
  }
```

This code is used in section 4.

7. The same reasons make us to handle "one-character" outputs the same way. Moreover we return single value. Again, the interface of the parent is here too, otherwise it would be redefined.

Reading the output of a one symbol long word is easy and therefore efficient. It reduces to array access in fact.

```
⟨ IO word manipulator: word output 7 ⟩ ≡
  void word_output(const vector⟨Tin_alph⟩ &w, vector⟨Tout_information⟩ &o)
  {     /* Call the parent */
    Trie_pattern_manipulator⟨Tpm_pointer, Tin_alph, Tout_information⟩::word_output(w, o);
  }
  void word_output(const Tin_alph &w, Tout_information &o)
  {
    o = trie_outp[trie_root + w];
  }
```

This code is used in section 4.

8. The reverse mapping store. In the output phase the reverse mapping is needed to print words into files. It does not need to be extremely efficient, therefore we do it using map of internal codes and vectors of external representations.

The **Tinternal** is type of internal code, the **Texternal** is the type of external information, we map **Tinternal** to vectors of **Texternal**. As data we define the appropriate **map**. Please note the order in the template, it is quite stupid but the more intelligent version is not compiled by some compilers (my gcc-2.96, for example).

```
  format Texternal int
  format Tinternal int
  template⟨class Texternal, class Tinternal⟩
  class IO_reverse_mapping {
  protected:
    map⟨Tinternal, vector⟨Texternal⟩⟩ mapping;
    ⟨ IO reverse mapping: insert 9 ⟩
    ⟨ IO reverse mapping: add to string 10 ⟩
  };
```

9. Inserting is easy. We simply put it there.

```
⟨ IO reverse mapping: insert 9 ⟩ ≡
public:
  void insert(const Tinternal &i, const vector⟨Texternal⟩ &v)
  {
    mapping[i] = v;
  }
```

This code is used in section 8.

10. Reading the value goes as follows. We have the internal code of a sequence and the vector of so-far collected external representations. The external representation of the internal code is added to the end of the **basic_string**.

Note that the existence of the field is not checked. FIXME: should throw an exception if out of bounds!

⟨ IO reverse mapping: add to string 10 ⟩ ≡

```
public:
  void add_to_string(const Tinternal &i, basic_string⟨Texternal⟩ &s)
  {
    map⟨Tinternal, vector⟨Texternal⟩⟩::const_iterator it = mapping.find(i);
    s.insert(s.end(), it→second.begin(), it→second.end());
  }
```

This code is used in section 8.

11. Translate service. This service reads the translate file (and/or sets default values if there is none) and translates the input word from the file format into the internal encoding and vice versa.

The **Tindex** type is the type of *max_in_alph* and *left_hyphen_min*, the **Tnum_type** is the type of internal representation of a letter (not here, in the generator). More precisely it must be a supertype of input alphabet, internal codes of numbers and hyphens and all the similar values. We do it like this only for ease of access to the internal codes of external representations.

THword is here only to make *Thyf_type* defined there available here.

```
format Tindex int
format Tnum_type int
format THword int

template⟨class Tindex, class Tnum_type, class THword⟩
class Translate {
  ⟨ Translate: data 12 ⟩
  ⟨ Translate: get next internal code 13 ⟩
  ⟨ Translate: classify 14 ⟩
  ⟨ Translate: prepare fixed defaults 15 ⟩
  ⟨ Translate: prepare default hyfs 16 ⟩
  ⟨ Translate: prepare default alphabet 17 ⟩
  ⟨ Translate: handle preamble of translate 18 ⟩
  ⟨ Translate: handle line of translate 19 ⟩
  ⟨ Translate: read translate 22 ⟩
  ⟨ Translate: constructor 23 ⟩
  ⟨ Translate: gets 24 ⟩
  ⟨ Translate: get xdig 25 ⟩
  ⟨ Translate: get xhyf 26 ⟩
  ⟨ Translate: get xext 27 ⟩
};
```

12. The reading routines recognize character classes in order to parse the input lines, so we provide names for them in the **Tcharacter_class** type.

The **Tfile_unit** type is the type of the codes stored in local structures. The terminology goes crazy.

The **Tclassified_symbol** is the type of class and internal code and/or other useful value.

The *edge_of_word* contains the internal code of "edge of word" character.

The *max_in_alph* is the highest internal code used, *left_hyphen_min* and *right_hyphen_min* are here only as they may be specified in the translate file.

The *classified_symbols* structure stores the classes and internal values of symbols. The three reverse mappings *xdig*, *xhyf*, and *xext* specify the printable values of symbols of *digit_class*, *hyf_class*, and *letter_class*.

```
format Tcharacter_class int
format Tfile_unit int
format Tclassified_symbol int
```

⟨ Translate: data 12 ⟩ ≡

```
public:
  typedef enum {
    space_class,      /* the space character */
    digit_class,      /* the characters '0'...'9' */
    hyf_class,        /* the hyphen characters, '.', '-', '*' by default */
    letter_class,     /* the letters */
    escape_class,     /* character starting a multi-character sequence representing a letter */
    invalid_class     /* character which should not occur */
  } Tcharacter_class;
  typedef unsigned char Tfile_unit;
  typedef pair⟨Tcharacter_class, Tnum_type⟩ Tclassified_symbol;
protected:
  Tnum_type edge_of_word;
  Tindex max_in_alph;
  Tindex left_hyphen_min;
  Tindex right_hyphen_min;
  IO_word_manipulator⟨Tindex, Tfile_unit, Tclassified_symbol⟩ classified_symbols;
  IO_reverse_mapping⟨Tfile_unit, Tnum_type⟩ xdig;
  IO_reverse_mapping⟨Tfile_unit, typename THword::Thyf_type⟩ xhyf;
  IO_reverse_mapping⟨Tfile_unit, Tnum_type⟩ xext;
```

This code is used in section 11.

13. When building the internal alphabet we need to keep track of last used internal code. Using this method only everything goes fine. It increments *max_in_alph* by one and returns it.

⟨ Translate: get next internal code 13 ⟩ ≡

```
protected:
  Tnum_type get_next_internal_code(void)
  {
    max_in_alph++;
    return max_in_alph;
  }
```

This code is used in section 11.

14. Classification of characters. The first parameter is the "file character", the second one is the classification with the internal code. The method is also overloaded for vectors.

```
⟨ Translate: classify 14 ⟩ ≡
public:
  void classify(const Tfile_unit &c, Tclassified_symbol &o)
  {
    classified_symbols.word_output(c, o);
  }
  void classify(const vector⟨Tfile_unit⟩ &vc, Tclassified_symbol &o)
  {
    classified_symbols.word_last_output(vc, o);
  }
```

This code is used in section 11.

15. The internal codes of digits are their values. The printable digits are also set. The space and tab characters are bound to *space_class*. The spaces need no value, so zero is substituted. Printable spaces are not needed. We put all of the symbols into the *classified_symbols*.

Moreover the *edge_of_word* is set to the first free internal code and *edge_of_word_printable* is set to dot character and the representation is written into the *xext* structure.

```
⟨ Translate: prepare fixed defaults 15 ⟩ ≡
protected:
  void prepare_fixed_defaults(void)
  {
    Tnum_type d;
    vector⟨Tfile_unit⟩ repres;
    for (d = 0; d ≤ 9; d++) {
      classified_symbols.hard_insert_pattern((d + '0'), make_pair(digit_class, d));
      repres.clear();
      repres.push_back(d + '0');
      xdig.insert(d, repres);
    }
    classified_symbols.hard_insert_pattern('␣', make_pair(space_class, 0));
    classified_symbols.hard_insert_pattern(9, make_pair(space_class, 0));      /* tab character */
    edge_of_word = get_next_internal_code();
    vector⟨Tfile_unit⟩ edge_of_word_printable;
    edge_of_word_printable.push_back('.');
    xext.insert(edge_of_word, edge_of_word_printable);
  }
```

This code is used in section 11.

16. Preparing default tables for hyfs and letters is used when no translate file exists. The default hyphenation symbols '.', '-', and '*' are set using the *prepare_default_hyfs* procedure, together with the *xhyf* printable values.

```
⟨ Translate: prepare default hyfs 16 ⟩ ≡
protected:
  void prepare_default_hyfs(void)
  {
    vector⟨Tfile_unit⟩ repres;
    classified_symbols.hard_insert_pattern('.', make_pair(hyf_class, THword::err_hyf));
    repres.clear();
    repres.push_back('.');
    xhyf.insert(THword::err_hyf, repres);
    classified_symbols.hard_insert_pattern('-', make_pair(hyf_class, THword::is_hyf));
    repres.clear();
    repres.push_back('-');
    xhyf.insert(THword::is_hyf, repres);
    classified_symbols.hard_insert_pattern('*', make_pair(hyf_class, THword::found_hyf));
    repres.clear();
    repres.push_back('*');
    xhyf.insert(THword::found_hyf, repres);
  }
```

This code is used in section 11.

17. In *prepare_default_alphabet* we set the default English alphabet. All the 'a'...'z' characters and their uppercase counterparts are assigned to internal codes and *letter_class*, the printable values are set to lowercase forms. The *max_in_alph* is increased.

```
⟨ Translate: prepare default alphabet 17 ⟩ ≡
protected:
  void prepare_default_alphabet(void)
  {
    vector⟨Tfile_unit⟩ repres;
    Tnum_type internal;
    for (Tfile_unit c = 'a'; c ≤ 'z'; c++) {
      internal = get_next_internal_code();
      classified_symbols.hard_insert_pattern(c, make_pair(letter_class, internal));
      classified_symbols.hard_insert_pattern(c + 'A' − 'a', make_pair(letter_class, internal));
      repres.clear();
      repres.push_back(c);
      xext.insert(internal, repres);
    }
  }
```

This code is used in section 11.

18. The first line of the translate file is special. It must contain the values of *left_hyphen_min* and *right_hyphen_min* in columns 1–2 and 3–4. Moreover columns 5, 6, and 7 may contain replacements for the default characters `'.'`, `'-'`, and `'*'`, representing hyphens in the word list. The rest of the line is ignored. If the values specified for *left_hyphen_min* and *right_hyphen_min* are invalid, new values are read from the terminal.

```
⟨ Translate: handle preamble of translate 18 ⟩ ≡
protected:
  void handle_preamble_of_translate(const basic_string⟨Tfile_unit⟩ &s)
  {
    Tindex n = 0;
    bool bad = false;
    Tclassified_symbol cs;
    if (s.length() ≥ 4) {      /* we have them */
      classify(s[0], cs);      /* first two chars */
      if (cs.first ≡ space_class) n = 0;
      else {
        if (cs.first ≡ digit_class) n = cs.second;
        else bad = true;
      }
      classify(s[1], cs);
      if (cs.first ≡ digit_class) n = 10 * n + cs.second;
      else bad = true;
      if (n ≥ 1) left_hyphen_min = n;
      else bad = true;
      classify(s[2], cs);      /* the second pair of chars */
      if (cs.first ≡ space_class) n = 0;
      else {
        if (cs.first ≡ digit_class) n = cs.second;
        else bad = true;
      }
      classify(s[3], cs);
      if (cs.first ≡ digit_class) n = 10 * n + cs.second;
      else bad = true;
      if (n ≥ 1) right_hyphen_min = n;
      else bad = true;
    }
    else bad = true;
    if (bad) {      /* wrong, never mind, let's ask the user */
      bad = false;
      Tindex n1;
      Tindex n2;
      cout ≪ "! Values of left_hyphen_min and right_hyphen_min in translate";
      cout ≪ " are invalid." ≪ endl;
      do {
        cout ≪ "left_hyphen_min, right_hyphen_min: ";
        cin ≫ n1 ≫ n2;
        if (n1 ≥ 1 ∧ n2 ≥ 1) {
          left_hyphen_min = n1;
          right_hyphen_min = n2;
        }
        else {
```

```
        n1 = 0;
        cout ≪ "Specify␣1<=left_hyphen_min,␣right_hyphen_min!" ≪ endl;
      }
    } while (¬n1 > 0);
  }    /* closing of if (bad) */
  for (Tindex i = THword::err_hyf; i ≤ THword::found_hyf; i++) {
    /* the last three characters */
    if (s.length() − 1 ≥ i + 3) {    /* there is a symbol */
      classify(s[i + 3], cs);
      if (utf_8 ∧ s[i + 3] > #80) {
        throw Patlib_error("!␣Error␣reading␣translate␣file,␣In␣the␣\
            first␣line,␣""specifying␣hyf␣characters:\n""In␣UTF-8␣mode␣8-bit\
            ␣symbol␣is␣not␣allowed.");
      }
      if (cs.first ≡ space_class) continue;    /* ignore if not specified */
      if (cs.first ≡ invalid_class) {    /* hasn't been used before */
        vector⟨Tfile_unit⟩ v;
        v.push_back(s[i + 3]);
        xhyf.insert((typename THword::Thyf_type) i, v);    /* register it */
        classified_symbols.hard_insert_pattern(s[i + 3], make_pair(hyf_class, i));
      }
      else {
        throw Patlib_error("!␣Error␣reading␣translate␣file.␣In␣the␣\
            first␣line,␣""specifying␣hyf␣characters:\n""Specified␣symbol␣ha\
            s␣been␣already␣assigned.");
      }
    }
  }
}
```

This code is used in section 11.

19. Each line (except the first one) of the translate file is either a comment or specifies the external representation of one "letter" used by the language. Blank lines or lines starting with two equal characters are completely ignored. Other lines contain the external representation of one primary representation of a letter followed by any number of secondary representations. All the representations read from the file are mapped to one internal code. When typing a letter into file, only the primary representation is used. The representations are preceded and separated by a delimiter. The delimiter may be any 7-bit ASCII character not occurring in either version.

The structure is PATGEN compatible, PATGEN only requires the multi-character sequences to be followed by doubled delimiter.

How the line is parsed. We put a pair of delimiters to the end of the string. This assures we do not have to test the end of the string. The "do forever" loop skips the delimiter and tests the following character. Looking at delimiter again, we are done. Otherwise we collect the symbols into the *letter_repres* vector and have it handled. Only for the first representation new internal code is prepared.

The procedure quits, as the line is finite and each step eats a character and it does not overrun the s string as we put double delimiter to the end of it and when reaching two delimiters we always break the loop.

```
⟨ Translate: handle line of translate 19 ⟩ ≡
protected:
  void handle_line_of_translate(basic_string⟨Tfile_unit⟩ &s, const unsigned &lineno)
  {
    if (s.length() ≡ 0) return;      /* nothing to do */
    bool primary_repres = true;      /* the first is the primary representation */
    vector⟨Tfile_unit⟩ letter_repres;
    Tnum_type internal;      /* internal code of this letter */
    Tfile_unit delimiter = *s.begin();
    s = s + delimiter + delimiter;      /* the line ends with a double delimiter for sure */
    basic_string⟨Tfile_unit⟩::const_iterator i = s.begin();
    while (true) {      /* do forever */
      i++;      /* skip the delimiter */
      if (*i ≡ delimiter) break;      /* quit if double delimiter, rest of line ignored */
      letter_repres.clear();
      while (*i ≠ delimiter) {      /* read the representation */
        letter_repres.push_back(*i);
        i++;
      }
      if (primary_repres) internal = get_next_internal_code();      /* if primary, get new code */
      ⟨ Translate: (handle line of translate) handle letter representation 20 ⟩
      primary_repres = false;      /* next is not primary any more */
    }      /* end of do forever */
  }
```

This code is used in section 11.

20. Registering the letter representation. We store letters into *classified_symbols* after some necessary tests.

One-symbol letters must have not been assigned before, first symbol of multi-symbol letter must have escape-class and the symbol must have not been used before, too.

```
⟨ Translate: (handle line of translate) handle letter representation 20 ⟩ ≡
  {
    Tclassified_symbol cs;
    if (letter_repres.size() ≡ 1) {      /* has just one symbol */
      classify(*letter_repres.begin(), cs);
      if (utf_8 ∧ *letter_repres.begin() > 127) {
        cout ≪ "!␣Warning:␣Translate␣file,␣line␣" ≪ lineno ≪ ":" ≪ endl;
        cout ≪ "There␣is␣single␣8-bit␣ASCII␣character,␣it␣is␣probably␣an␣error␣";
        cout ≪ "in␣UTF-8␣mode" ≪ endl;
      }
      if (cs.first ≡ invalid_class) {
        classified_symbols.hard_insert_pattern(letter_repres, make_pair(letter_class, internal));
      }
      else {
        cerr ≪ "!␣Error:␣Translate␣file,␣line␣" ≪ lineno ≪ ":" ≪ endl;
        cerr ≪ "Trying␣to␣redefine␣previously␣defined␣character" ≪ endl;
        throw Patlib_error("");      /*FIXME */
      }
    }
    else {      /* has more symbols than one */
      classify(*letter_repres.begin(), cs);
      if (cs.first ≡ invalid_class)      /* invalid → escape is OK */
        classified_symbols.hard_insert_pattern(*letter_repres.begin(), make_pair(escape_class, 0));
      classify(*letter_repres.begin(), cs);
      if (cs.first ≠ escape_class) {
        cerr ≪ "!␣Error:␣Translate␣file,␣line␣" ≪ lineno ≪ ":" ≪ endl;
        cerr ≪ "The␣first␣symbol␣of␣multi-char␣or␣UTF-8␣sequence␣has␣been␣";
        cerr ≪ "used␣before";
        cerr ≪ endl ≪ "as␣non-escape␣character" ≪ endl;
        throw Patlib_error("");      /*FIXME */
      }      /* OK, now we start with escape, let's test the letter itself */
      classify(letter_repres, cs);
      if (cs.first ≠ invalid_class) {
        cerr ≪ "!␣Error:␣Translate␣file,␣line␣" ≪ lineno ≪ ":" ≪ endl;
        cerr ≪ "Trying␣to␣redefine␣previously␣defined␣character" ≪ endl;
        throw Patlib_error("");      /*FIXME */
      }      /* Now it should be correct, create the letter */
      ⟨ Translate: (handle line of translate) check UTF-8 sequence 21 ⟩
      classified_symbols.hard_insert_pattern(letter_repres, make_pair(letter_class, internal));
    }
    if (primary_repres)      /* Reverse mapping */
      xext.insert(internal, letter_repres);
  }
```

This code is used in section 19.

21. When having UTF-8 sequence, we'd better check it is OK and if it not, we give a warning.

```
⟨ Translate: (handle line of translate) check UTF-8 sequence 21 ⟩ ≡
  if (utf_8) {
    Tfile_unit first = *letter_repres.begin();
    unsigned expected_length = 0;
    while (first & ^80) {      /* do until we reach first binary 0 */
      expected_length++;
      first = first << 1;
    }
    if (letter_repres.size() ≠ expected_length) {
      cout << "! Warning: Translate file, line " << lineno << ":" << endl;
      cout << "UTF-8 sequence seems to be broken, it is probably an error." << endl;
    }
  }
```

This code is used in section 20.

22. The translate file specifies the values of *left_hyphen_min* and *right_hyphen_min* as well as the external representations of letters used by the language. Replacements for the characters `'-'`, `'*'`, and `'.'` representing hyphens in the word list may also be specified. If the translate file is empty default values are used.

This is PATGEN compatible behavior.

```
⟨ Translate: read translate 22 ⟩ ≡
protected:
  void read_translate(const char *tra)
  {
    unsigned lineno = 1;
    ifstream transl(tra);
    basic_string⟨Tfile_unit⟩ s;
    if (getline(transl, s)) {
      handle_preamble_of_translate(s);
      while (getline(transl, s)) handle_line_of_translate(s, ++lineno);
    }
    else {
      cout << "Translate file does not exist or is empty. Defaults used." << endl;
      prepare_default_alphabet();
      left_hyphen_min = 2;
      right_hyphen_min = 3;
    }
    cout << "left_hyphen_min = " << left_hyphen_min << ", right_hyphen_min = " <<
        right_hyphen_min << endl << max_in_alph - edge_of_word << " letters" << endl;
  }
```

This code is used in section 11.

23. The constructor reads the file and builds translating structures. In the beginning the *classified_symbols* structure is initialized with *invalid_class* (with zero internal code, which is not too important) and the *max_in_alph* is set to zero.

```
⟨ Translate: constructor 23 ⟩ ≡
public:
  Translate(const char *tra)
  : max_in_alph(0), classified_symbols(255, make_pair(invalid_class, 0)) {
    prepare_fixed_defaults();
    prepare_default_hyfs();
    read_translate(tra);
  }
```

This code is used in section 11.

24. We must let the higher level know the following values.

```
⟨ Translate: gets 24 ⟩ ≡
public:
  Tindex get_max_in_alph(void)
  {
    return max_in_alph;
  }
  Tindex get_right_hyphen_min(void)
  {
    return right_hyphen_min;
  }
  Tindex get_left_hyphen_min(void)
  {
    return left_hyphen_min;
  }
  Tfile_unit get_edge_of_word(void)
  {
    return edge_of_word;
  }
```

This code is used in section 11.

25. Getting outer representations of a number is the only a bit more complicated problem. We get a number and prepare its external representation using the most stupid way we can. We compute the reverse and append it (reversed, of course) to the e string.

```
⟨ Translate: get xdig 25 ⟩ ≡
public:
  void get_xdig(Tnum_type i, basic_string⟨Tfile_unit⟩ &e)
  {
    basic_string⟨Tfile_unit⟩ inv_rep;
    while (i > 0) {
      xdig.add_to_string((i % 10), inv_rep);
      i = Tnum_type(i/10);
    }
    e.append(inv_rep.rbegin(), inv_rep.rend());
  }
```

This code is used in section 11.

26. Get the external representation of hyphenation character. The representation is appended to the e string.

```
⟨ Translate: get xhyf 26 ⟩ ≡
public:
  void get_xhyf(const typename THword::Thyf_type &i, basic_string⟨Tfile_unit⟩ &e)
  {
    xhyf.add_to_string(i, e);
  }
```

This code is used in section 11.

27. Get the external representation of a letter. Note that we do not have to take care of the length of the representation. The representation is appended to the e string.

```
⟨ Translate: get xext 27 ⟩ ≡
public:
  void get_xext(const Tnum_type &i, basic_string⟨Tfile_unit⟩ &e)
  {
    xext.add_to_string(i, e);
  }
```

This code is used in section 11.

28. Word input file. We have to read the input data, which is a list of words together with the hyphenation information and weights. To make such an object, we have to know the weight type, the **THword**, and the **TTranslate** types.

```
format THword int
format TTranslate int
format Tnum_type int
template⟨class THword, class TTranslate, class Tnum_type⟩
class Word_input_file {
  ⟨ Word input file: data 29 ⟩
  ⟨ Word input file: constructor 30 ⟩
  ⟨ Word input file: handle line 31 ⟩
  ⟨ Word input file: get 40 ⟩
};
```

29. We have to know the translate and the file name. We also prepare the *file* to be **ifstream**. The *lineno* value is only the number of line just read. And finally we make some types available here easily.

The *global_word_wt* holds the word weight which applies to all the next words until it is changed.

```
⟨ Word input file: data 29 ⟩ ≡
protected:
  TTranslate &translate;
  const char *file_name;
  ifstream file;
  unsigned lineno;
  typedef typename TTranslate::Tfile_unit Tfile_unit;
  typedef typename TTranslate::Tclassified_symbol Tclassified_symbol;
  Tnum_type global_word_wt;
```

This code is used in section 28.

30. The constructor sets the values and opens the file. The default word weight is 1.

```
⟨ Word input file: constructor 30 ⟩ ≡
public:
  Word_input_file(TTranslate &t, const char *fn)
  : translate(t), file_name(fn), file(file_name), lineno(0), global_word_wt(1) { }
```

This code is used in section 28.

31. A line of input data in s is always ended by space character and the *hw* word is empty. We parse the line and fill the *hw*. The *edge_of_word* character is put at the very beginning and at the end of word.

The line of input data contains just one word consisting of letters used by the language. "Dots" between the letters may be one of four possibilities: `'-'`—a hyphen, `'*'`—a found hyphen, `'.'`—an error, or nothing, represented internally by *is_hyf*, *found_hyf*, *err_hyf*, and *no_hyf* respectively. When reading a word, we convert *err_hyf* into *no_hyf* and *found_hyf* into *is_hyf*, we ignore whether the hyphen has or has not been found by previous set of patterns.

Digit weights are allowed. A number at some intercharacter position indices weight for that position. A number starting in the very first column indices global word weight which applies to all the positions of all the following words. The global weight is stored in *hw.dotw*[0] as this position is not used by the generator. The other *dotw* positions are their logical weights, it means they have the global weight if there were nothing in the file and the value from the file if that is set. Note that *dotw*[0] is a bit misused.

```
⟨ Word input file: handle line 31 ⟩ ≡
protected:
  void handle_line(const basic_string⟨Tfile_unit⟩ &s, THword &hw)
  {
    hw.push_back(translate.get_edge_of_word());
    hw.dotw[hw.size()] = global_word_wt;      /* may be redefined later */
    Tclassified_symbol i_class;
    basic_string⟨Tfile_unit⟩::const_iterator i = s.begin();
    vector⟨Tfile_unit⟩ seq;
    Tnum_type num;
    do {
      if (utf_8 ∧ (*i & #80)) {
        ⟨ Word input file: (handle line) multibyte sequence 32 ⟩
      }
      else {      /* we have one byte */
        translate.classify(*i, i_class);
        switch (i_class.first) {
        case TTranslate::space_class: goto done;
        case TTranslate::digit_class: ⟨ Word input file: (handle line) digit 34 ⟩
          break;
        case TTranslate::hyf_class: ⟨ Word input file: (handle line) hyf 36 ⟩
          break;
        case TTranslate::letter_class: ⟨ Word input file: (handle line) letter 37 ⟩
          break;
        case TTranslate::escape_class: ⟨ Word input file: (handle line) escape 38 ⟩
          break;
        default:      /* invalid_class is here */
          cerr ≪ "!␣Error␣in␣" ≪ file_name ≪ "␣line␣" ≪ lineno ≪ ":␣" ≪
              "Invalid␣character␣in␣input␣data" ≪ endl;
          throw Patlib_error("");      /*FIXME */
          break;
        }
      }
    } while (i ≠ s.end());
  done: hw.push_back(translate.get_edge_of_word());
    hw.dotw[hw.size()] = global_word_wt;
    hw.dotw[0] = global_word_wt;      /* the flag for the printing routine */
  }
```

This code is used in section 28.

32. A multibyte sequence of symbols meaning one letter. We take all characters bigger than 127 which follow, collect them and test them to be a letter. In that case we put them into the *hw*, otherwise it's an error.

```
⟨ Word input file: (handle line) multibyte sequence 32 ⟩ ≡
  {
    ⟨ Word input file: (handle line) read multibyte sequence 33 ⟩
    translate.classify(seq, i_class);
    if (i_class.first ≡ TTranslate::letter_class) {
      hw.push_back(i_class.second);
      hw.dotw[hw.size()] = global_word_wt;
    }
    else {
      cerr ≪ "!␣Error␣in␣" ≪ file_name ≪ "␣line␣" ≪ lineno ≪ ":␣" ≪
          "Multibyte␣sequence␣is␣invalid" ≪ endl;
      throw Patlib_error("");      /*FIXME */
    }
  }
```

This code is used in section 31.

33. Reading the multibyte sequence. The UTF-8 sequence has all its members > 127, in other words with the most significant bit 1. The first character determines the length of the sequence, it has as many ones as the sequence has members before its first zero. The schema makes it clear.

```
110xxxxx 10xxxxxx
1110xxxx 10xxxxxx 10xxxxxx
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

The UTF-8 characters may also be `0xxxxxxx`, but that is equivalent to 7-bit ASCII and this is not handled by this procedure.

We remember the first character and shift it left and testing the highest bit we count the following characters.

```
⟨ Word input file: (handle line) read multibyte sequence 33 ⟩ ≡
  Tfile_unit first_i = *i;
  seq.clear();
  while ((first_i & #80) ∧ (*i & #80)) {      /* the highest bit is 1 and we check the nth character too */
    seq.push_back(*i);
    i++;
    first_i = first_i ≪ 1;      /* shift left */
  }
```

This code is used in sections 32 and 45.

34. A number is a sequence of decadic digits. In the first column it means the global weight (until changed), otherwise it is only local to a position.

```
⟨ Word input file: (handle line) digit 34 ⟩ ≡
  if (i ≡ s.begin()) {      /* in the first column set also the global weight */
    ⟨ Word input file: (handle line) read number 35 ⟩
    hw.dotw[hw.size()] = num;
    global_word_wt = num;
  }
  else {      /* otherwise only the position is affected */
    ⟨ Word input file: (handle line) read number 35 ⟩
    hw.dotw[hw.size()] = num;
  }
```

This code is used in section 31.

35. Reading the number. We read digit-by-digit. The cycle ends, let us recall that there is always a space in the end of the line.

```
⟨ Word input file: (handle line) read number 35 ⟩ ≡
  num = 0;
  while (i_class.first ≡ TTranslate::digit_class) {
    num = 10 * num + i_class.second;
    i++;
    translate.classify(*i, i_class);
  }
```

This code is used in sections 34 and 46.

36. A hyphen. The default value is *no_hyf*, we have to change it only if we deal with *is_hyf* or *found_hyf*.

```
⟨ Word input file: (handle line) hyf 36 ⟩ ≡
  if (i_class.second ≡ THword::is_hyf ∨ i_class.second ≡ THword::found_hyf)
    hw.dots[hw.size()] = THword::is_hyf;
  i++;
```

This code is used in section 31.

37. A letter.

```
⟨ Word input file: (handle line) letter 37 ⟩ ≡
  hw.push_back(i_class.second);
  hw.dotw[hw.size()] = global_word_wt;
  i++;
```

This code is used in section 31.

38. Escape sequence is a sequence of an escape character and a non-empty mixture of letters and invalid characters. Any non-invalid and non-letter (e.g., space, digit, hyphen) character stops reading the sequence.

If the sequence is followed by spaces (more precisely characters with *space_class*), the spaces are skipped. Keep in mind the line always ends with at least one space.

The escape sequence is checked to be a letter. We insert its internal code in that case.

```
⟨ Word input file: (handle line) escape 38 ⟩ ≡
  ⟨ Word input file: (handle line) read escape sequence 39 ⟩
  if (i_class.first ≡ TTranslate::letter_class) {
    hw.push_back(i_class.second);
    hw.dotw[hw.size()] = global_word_wt;
  }
  else {
    cerr ≪ "!␣Error␣in␣" ≪ file_name ≪ "␣line␣" ≪ lineno ≪ ":␣" ≪
        "Escape␣sequence␣is␣invalid" ≪ endl;
    cerr ≪ "(Are␣you␣using␣correct␣encoding--the␣-u8␣switch?)" ≪ endl;
    throw Patlib_error("");      /* FIXME */
  }
```

This code is used in section 31.

39. Reading the escape sequence.

```
⟨ Word input file: (handle line) read escape sequence 39 ⟩ ≡
  seq.clear();
  seq.push_back(*i);      /* push back the escape */
  i++;
  translate.classify(*i, i_class);
  while (i_class.first ≡ TTranslate::letter_class ∨ i_class.first ≡ TTranslate::invalid_class) {
    seq.push_back(*i);
    i++;
    translate.classify(*i, i_class);
  }     /* we have read the sequence */
  while (i_class.first ≡ TTranslate::space_class ∧ i ≠ s.end()) {
    i++;
    translate.classify(*i, i_class);
  }     /* we have skipped blanks */
  translate.classify(seq, i_class);
```

This code is used in sections 38 and 48.

40. Getting next **THword**. If there is one, we return true and the values, otherwise we return false. Each line must contain just one word with hyphenation information. The *handle_line* method requires the s string to be ended with a space character.

```
⟨ Word input file: get 40 ⟩ ≡
public:
  bool get(THword &hw)
  {
    hw.clear();
    basic_string⟨Tfile_unit⟩ s;
    if (¬getline(file, s)) {
      return false;
    }
    else {      /* we have a line, so let's handle it */
      lineno++;
      s.push_back(Tfile_unit('␣'));
      handle_line(s, hw);
    }
    return true;
  }
```

This code is used in section 28.

41. Pattern input file. Before the first pass is run, we may want to read the patterns, for example selected in previous runs. We must therefore be able to read them in.

```
format Tindex int
format Tin_alph int
format Tval_type int
format TTranslate int
format TOutputs_of_a_pattern int
template⟨class Tindex, class Tin_alph, class Tval_type, class TTranslate, class
    TOutputs_of_a_pattern⟩
class Pattern_input_file {
  ⟨ Pattern input file: data 42 ⟩
  ⟨ Pattern input file: constructor 43 ⟩
  ⟨ Pattern input file: handle line 44 ⟩
  ⟨ Pattern input file: get 49 ⟩
};
```

42. Does the comment of this section bore you?

```
⟨ Pattern input file: data 42 ⟩ ≡
protected:
  TTranslate &translate;
  const char *file_name;
  ifstream file;
  unsigned lineno;
  typedef typename TTranslate::Tfile_unit Tfile_unit;
  typedef typename TTranslate::Tclassified_symbol Tclassified_symbol;
```

This code is used in section 41.

43. The constructor sets the values and opens the file.

```
⟨ Pattern input file: constructor 43 ⟩ ≡
public:
  Pattern_input_file(TTranslate &t, const char *fn)
  : translate(t), file_name(fn), file(file_name), lineno(0) { }
```

This code is used in section 41.

44. We parse the s string (we know it end with at least one space) and return the word and its output.

```
⟨ Pattern input file: handle line 44 ⟩ ≡
  protected:
  void handle_line(const basic_string⟨Tfile_unit⟩ &s, vector⟨Tin_alph⟩ &v, TOutputs_of_a_pattern
        &o){ Tclassified_symbol i_class;
    basic_string⟨Tfile_unit⟩::const_iterator i = s.begin();
    vector⟨Tfile_unit⟩ seq;
    Tval_type num;
    Tindex chars_read = 0; do {
    if (*i ≡ '.') {      /* a dot means edge of word here, let's treat it specially */
      v.push_back(translate.get_edge_of_word());
      chars_read++;
      i++;      /* go to the next character */
      continue;
    }
    if (utf_8 ∧ *i > 127) ⟨ Pattern input file: (handle line) multibyte sequence 45 ⟩
    else {
      translate.classify(*i, i_class);
      switch (i_class.first) {
      case TTranslate::space_class: goto done;
      case TTranslate::digit_class: ⟨ Pattern input file: (handle line) digit 46 ⟩
        break;
      case TTranslate::letter_class: ⟨ Pattern input file: (handle line) letter 47 ⟩
        break;
      case TTranslate::escape_class: ⟨ Pattern input file: (handle line) escape 48 ⟩
        break;
      default:      /* hyf_class (except a dot), invalid_class */
        cerr ≪ "!␣Error␣in␣" ≪ file_name ≪ "␣line␣" ≪ lineno ≪ ":␣" ≪
            "Invalid␣character␣in␣pattern␣data" ≪ endl;
        throw Patlib_error("");      /*FIXME */
      }
    }
    }
    while (i ≠ s.end()) ;
  done: ; }
```

This code is used in section 41.

45. Multibyte sequence.

```
⟨ Pattern input file: (handle line) multibyte sequence 45 ⟩ ≡
  {
    ⟨ Word input file: (handle line) read multibyte sequence 33 ⟩
    translate.classify(seq, i_class);
    if (i_class.first ≡ TTranslate::letter_class) {
      v.push_back(i_class.second);
      chars_read++;
    }
    else {
      cerr ≪ "!␣Error␣in␣" ≪ file_name ≪ "␣line␣" ≪ lineno ≪ ":␣" ≪
          "Multibyte␣sequence␣is␣invalid" ≪ endl;
      throw Patlib_error("");      /*FIXME */
    }
  }
```

This code is used in section 44.

46. A digit.

```
⟨ Pattern input file: (handle line) digit 46 ⟩ ≡
  ⟨ Word input file: (handle line) read number 35 ⟩
  o.insert(make_pair(chars_read, num));
```

This code is used in section 44.

47. A letter.

```
⟨ Pattern input file: (handle line) letter 47 ⟩ ≡
  v.push_back(i_class.second);
  chars_read++;
  i++;
```

This code is used in section 44.

48. Escape sequence.

```
⟨ Pattern input file: (handle line) escape 48 ⟩ ≡
  ⟨ Word input file: (handle line) read escape sequence 39 ⟩
  if (i_class.first ≡ TTranslate::letter_class) {
    v.push_back(i_class.second);
    chars_read++;
  }
  else {
    cerr ≪ "!␣Error␣in␣" ≪ file_name ≪ "␣line␣" ≪ lineno ≪ ":␣" ≪
        "Escape␣sequence␣is␣invalid" ≪ endl;
    cerr ≪ "(Are␣you␣using␣correct␣encoding--the␣-u8␣switch?)" ≪ endl;
    throw Patlib_error("");      /*FIXME */
  }
```

This code is used in section 44.

49. The *get* method returns the vector of internal codes representing the word and its output. The vector and the output is emptied in the beginning. Line is parsed and the values are set, returning *true*. When reaching the end of the file, *false* is returned.

```
⟨ Pattern input file: get 49 ⟩ ≡
public:
  bool get(vector⟨Tin_alph⟩ &v, TOutputs_of_a_pattern &o)
  {
    v.clear( );
    o.clear( );
    basic_string⟨Tfile_unit⟩ s;
    if (¬getline(file, s)) {
      return false;
    }
    else {      /* we have a line, so let's handle it */
      lineno++;
      s.push_back('␣');
      handle_line(s, v, o);
    }
    return true;
  }
```

This code is used in section 41.

50. Word output file. If the user wants to see the work of the patterns on his input data, writing hyphenated words is needed.

```
format Tindex int
format THword int
format TTranslate int
template⟨class Tindex, class THword, class TTranslate⟩
class Word_output_file {
  ⟨ Word output file: data 51 ⟩
  ⟨ Word output file: constructor 52 ⟩
  ⟨ Word output file: put 53 ⟩
};
```

51. We have to know the translate, the file name and the **ofstream**. We also prepare easy access to some type names. The *last_global_word_wt* is the previous word weight. We output the global weight only if it is changed. FIXME: Why this couldn't be compiled with **typename THword**::**Twt_type**?!?

```
⟨ Word output file: data 51 ⟩ ≡
protected:
  TTranslate &translate;
  const char *file_name;
  ofstream file;
  typedef typename TTranslate::Tfile_unit Tfile_unit;
  unsigned last_global_word_wt;
  unsigned global_word_wt;
```

This code is used in section 50.

52. The constructor sets the values and opens the file.

```
⟨ Word output file: constructor 52 ⟩ ≡
public:
  Word_output_file(TTranslate &t, const char *fn)
  : translate(t), file_name(fn), file(file_name), last_global_word_wt(1) { }
```

This code is used in section 50.

53. Writing a **THword** into the file. The representation of *edge_of_word* character is ignored (on both sides), printable version of the *hw* is put to the file.

The global word weight is output in and only if it is changed. The interletter weights are output if they differ from the global word weight.

⟨ Word output file: put 53 ⟩ ≡

```
public:
  void put(THword &hw)
  {
    basic_string⟨Tfile_unit⟩ s;
    global_word_wt = hw.dotw[0];
    if (last_global_word_wt ≠ global_word_wt) {      /* global weight has changed */
      translate.get_xdig(hw.dotw[0], s);
      last_global_word_wt = global_word_wt;
    }
    if (hw.dots[1] ≠ THword::no_hyf) translate.get_xhyf(hw.dots[1], s);
    for (Tindex dpos = 2; dpos ≤ hw.size() − 1; dpos++) {
      translate.get_xext(hw[dpos], s);
      if (hw.dots[dpos] ≠ THword::no_hyf) translate.get_xhyf(hw.dots[dpos], s);
      if (hw.dotw[dpos] ≠ global_word_wt) translate.get_xdig(hw.dotw[dpos], s);
    }
    file ≪ s ≪ endl;
  }
```

This code is used in section 50.

54. Pattern output file. This interface writes the generated patterns into the files.

```
format Tindex int
format Tin_alph int
format Tval_type int
format TTranslate int
format TOutputs_of_a_pattern int
template⟨class Tindex, class Tin_alph, class Tval_type, class TTranslate, class
    TOutputs_of_a_pattern⟩
class Pattern_output_file {
  ⟨Pattern output file: data 55⟩
  ⟨Pattern output file: constructor 56⟩
  ⟨Pattern output file: put 57⟩
};
```

55. Hmm, quite as usual...

```
⟨Pattern output file: data 55⟩ ≡
protected:
  TTranslate &translate;
  const char *file_name;
  ofstream file;
  typedef typename TTranslate::Tfile_unit Tfile_unit;
```

This code is used in section 54.

56. Constructor sets values and opens the file.

```
⟨Pattern output file: constructor 56⟩ ≡
public:
  Pattern_output_file(TTranslate &t, const char *fn)
  : translate(t), file_name(fn), file(file_name) { }
```

This code is used in section 54.

57. Putting a pattern into file. We go through it and handle outputs and characters and the last output in the end.

```
⟨Pattern output file: put 57⟩ ≡
public:
  void put(const vector⟨Tin_alph⟩ &v, const TOutputs_of_a_pattern &o)
  {
    typename TOutputs_of_a_pattern::const_iterator oi;
    basic_string⟨Tfile_unit⟩ s;
    Tindex pos = 0;
    for (vector⟨Tin_alph⟩::const_iterator vi = v.begin(); vi ≠ v.end(); vi++) {
      ⟨Pattern output file: (put) output number on pos if exists 58⟩
      pos++;
      translate.get_xext(*vi, s);
    }
    ⟨Pattern output file: (put) output number on pos if exists 58⟩      /* the last output */
    file ≪ s ≪ endl;
  }
```

This code is used in section 54.

58. If there is an output, handle it.

```
⟨ Pattern output file: (put) output number on pos if exists 58 ⟩ ≡
  oi = o.find(pos);
  if (oi ≠ o.end()) {      /* there is an output for that position */
    translate.get_xdig(oi→second, s);
  }
```

This code is used in section 57.

59. Main function companion. Here we define the types for the generator.

```
format Tindex int
format Tin_alph int
format Tval_type int
format Twt_type int
format Tcount_type int
format THword int
format TTranslate int
format TCandidate_count_structure int
format TCompetitive_multi_out_pat_manip int
format TOutputs_of_a_pattern int
format TWord_input_file int
format TWord_output_file int
format TPattern_input_file int
format TPattern_output_file int
format TPass int
format TLevel int
format Hword int
format Candidate_count_trie int
format Competitive_multi_out_pat_manip int
format Outputs_of_a_pattern int
format Pass int
format Level int
format Generator int

typedef unsigned long Tindex; /* word/pattern index */
typedef unsigned Tin_alph; /* input alphabet type */
typedef unsigned short Tval_type; /* hyph. level number */
typedef unsigned Twt_type; /* weight type */
typedef unsigned Tcount_type; /* good/bad counts */
typedef unsigned Tnum_type;
  /* we need a supertype of Tin_alph, Tval_type, and Twt_type */
typedef Hword⟨Tindex, Tin_alph, Twt_type, Tval_type⟩ THword; /* Hword for generator */
typedef Translate⟨Tindex, Tin_alph, THword⟩ TTranslate; /* translate service */
typedef Candidate_count_trie⟨Tindex, Tin_alph, Tcount_type, Tcount_type⟩
    TCandidate_count_structure; /* candidate manipulator */
typedef Competitive_multi_out_pat_manip⟨Tindex, Tin_alph,
    Tval_type⟩ TCompetitive_multi_out_pat_manip; /* pattern manipulator */
typedef Outputs_of_a_pattern⟨Tindex, Tval_type⟩ TOutputs_of_a_pattern;
  /* outputs of a pattern type */
typedef Word_input_file⟨THword, TTranslate, Tnum_type⟩ TWord_input_file;
  /* word input file */
typedef Word_output_file⟨Tindex, THword, TTranslate⟩ TWord_output_file;
  /* word output file */
typedef Pattern_input_file⟨Tindex, Tin_alph, Tval_type, TTranslate, TOutputs_of_a_pattern⟩
    TPattern_input_file; /* pattern input file */
typedef Pattern_output_file⟨Tindex, Tin_alph, Tval_type, TTranslate, TOutputs_of_a_pattern⟩
    TPattern_output_file; /* pattern output file */
typedef Pass⟨Tindex, Tin_alph, Tval_type, Twt_type, Tcount_type, THword, TTranslate,
    TCandidate_count_structure, TCompetitive_multi_out_pat_manip,
    TOutputs_of_a_pattern, TWord_input_file⟩ TPass; /* the pass */
```

typedef Level⟨**Tindex**, **Tin_alph**, **Tval_type**, **Twt_type**, **Tcount_type**, **THword**, **TTranslate**, **TCandidate_count_structure**, **TCompetitive_multi_out_pat_manip**, **TWord_input_file**, **TPass**⟩ **TLevel**; /* the level */

60. Some prints we use sometimes.

```
void print_banner(void)
{
  cout ≪ endl;
  cout ≪ "Written␣and␣maintained␣by␣David␣Antos,␣xantos␣(at)␣fi.muni.cz" ≪ endl;
  cout ≪ "Copyright␣(C)␣2001␣David␣Antos" ≪ endl;
  cout ≪ "This␣is␣free␣software;␣see␣the␣source␣for␣copying␣";
  cout ≪ "conditions.␣There␣is␣NO" ≪ endl;
  cout ≪ "warranty;␣not␣even␣for␣MERCHANTABILITY␣or␣FITNESS␣";
  cout ≪ "FOR␣A␣PARTICULAR␣PURPOSE." ≪ endl ≪ endl;
  cout ≪ "Thank␣you␣for␣using␣free␣software!" ≪ endl ≪ endl;
}
```

61. The main function. We parse the command line arguments and create the generator.

```
int main(int argc, char *argv[])
{
  cout << "This␣is␣OPATGEN,␣version␣" << opatgen_version << endl;
  if (argc ≥ 2 ∧ (0 ≡ strcmp(argv[1], "--help"))) {
    cout << "Usage:␣opatgen␣[-u8]␣DICTIONARY␣PATTERNS␣OUTPUT␣TRANSLATE" << endl;
    cout << "␣␣Generate␣the␣OUTPUT␣hyphenation␣file␣from␣the" << endl;
    cout << "␣␣DICTIONARY,␣PATTERNS,␣and␣TRANSLATE␣files." << endl << endl;
    cout << "␣␣-u8␣␣␣␣␣files␣are␣in␣UTF-8␣UNICODE␣encoding." << endl << endl;
    cout << "opatgen␣--help␣␣␣␣␣print␣this␣help" << endl;
    cout << "opatgen␣--version␣␣print␣version␣information" << endl;
    print_banner();
    return 0;
  }
  if (argc ≥ 2 ∧ (0 ≡ strcmp(argv[1], "--version"))) {
    cout << "(CVS:␣" << opatgen_cvs_id << ")" << endl;
    cout << "with␣PATLIB,␣version␣" << patlib_version << endl;
    cout << "(CVS:␣" << patlib_cvs_id << ")" << endl;
    print_banner();
    return 0;
  }
  print_banner();
  try {
    if (argc ≡ 5) {      /* file names only */
      utf_8 = false;
      Generator⟨Tindex, Tin_alph, Tval_type, Twt_type, Tcount_type, THword, TTranslate,
          TCandidate_count_structure, TCompetitive_multi_out_pat_manip,
          TOutputs_of_a_pattern, TWord_input_file,
          TWord_output_file, TPattern_input_file, TPattern_output_file, TPass, TLevel⟩
          g(argv[1], argv[2], argv[3], argv[4]);
      g.do_all();
    }
    else if (argc ≡ 6 ∧ (0 ≡ strcmp(argv[1], "-u8"))) {      /* -u8 and file names */
      utf_8 = true;
      Generator⟨Tindex, Tin_alph, Tval_type, Twt_type, Tcount_type, THword, TTranslate,
          TCandidate_count_structure, TCompetitive_multi_out_pat_manip,
          TOutputs_of_a_pattern, TWord_input_file,
          TWord_output_file, TPattern_input_file, TPattern_output_file, TPass, TLevel⟩
          g(argv[2], argv[3], argv[4], argv[5]);
      g.do_all();
    }
    else {      /* this is an error */
      cout << "opatgen:␣needs␣some␣arguments" << endl << "Try␣`opatgen␣--help'" << endl;
      return 1;
    }
  }
  catch(Patlib_error e)
  {
    e.what();
    cerr << endl << "This␣was␣fatal␣error,␣sorry.␣Giving␣up." << endl;
  }
```

```
    catch(...)
    {
      cerr ≪ "An␣unexpected␣exception␣occurred.␣It␣means␣there␣is␣probably" ≪ endl;
      cerr ≪ "a␣bug␣in␣the␣program.␣Please␣report␣it␣to␣the␣maintainer." ≪ endl;
      cerr ≪ "Use␣opatgen␣--version␣to␣find␣out␣who␣the␣maintainer␣is.";
      cout ≪ "Do␣you␣want␣me␣to␣dump␣core?␣<y/n>␣" ≪ endl;
      string s;
      cin ≫ s;
      if (s ≡ "y" ∨ s ≡ "Y") {
        cout ≪ endl ≪ "Now␣I␣dump␣core..." ≪ endl;
        terminate();
      }     /* otherwise quit quietly */
    }
  }     /* end of OPATGEN */
```

OPATGEN