December 4, 2001 at 09:11

1. Introduction. This is PATLIB, PATtern LIBrary.

This library stores a finite language using the packed dynamic trie and manipulates patterns. It is used to write a pattern generator.

This library 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 library should work with any compiler supporting the STL and ANSI C++.

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

The author does not want to forbid anyone to use this software, nevertheless the author considers any military usage unmoral and unethical.

The *ptl_vers.h* header file defines the version number (change it whenever the library changes) and the CVS identification string.

```
⟨ptl_vers.h 1⟩ ≡
#ifndef PTL_VERS_H
#define PTL_VERS_H
  const char *patlib_version = "1.0";
  const char *patlib_cvs_id = "$Id: patlib.w,v 1.127 2001/12/04 08:10:22 antos Exp $";
#endif
```

2. Organization of the code. The code is highly templatized, in fact there is nothing without templates. Therefore the library is divided into header files which contain all the code.

The header file names look like `ptl_<name>.h` where `<name>` is a shorthand of the full section name. The header files correspond usually to the “starred” sections in the CWEB source. The header files define a pre-processor variable `PTL_<name>_H` which is tested so as not to load the same header file more then once.

The library has two more or less independent parts. The first part, the finite language store, may be used separately. It provides methods to store a finite language in the packed version of trie memory. This may be also used to store pattern set in the application using patterns.

The second part, the generator, takes input data with marked points of interest and creates a set of patterns which recognize the points of interest there. The generator uses the finite language store to keep the pattern candidates and the patterns.

3. Dealing with bugs in the library. The software is far from being perfect, so my English is. Note that a library like this is quite a specialized tool. The number of users is small, so if you find a bug, please report it to `xantos (at) fi.muni.cz`. Keep in mind that if *you* don't tell me, maybe nobody will.

Please tell me exactly your platform, compiler, the problem you've fallen into. Please always attach the revision control id you can see above.

So we consider any behaviour different that the documentation says to be a bug. Feel free to send me any notes and remarks, they will be taken seriously. Ask anything you want to.

And something more! This code is complicated, nevertheless the commentary should be clear as much as possible. If the commentary does not match the program, it is a bug. It is a big bug. So please report it.

If the comments are not clear, if they are messy, it is my fault, not yours. Do not sit, do not be depressed, complain! (But please do not mail me before you read it three times :-).)

```
format iterator int
format const_iterator int
```

4. Exception handling in the library. When anything unpleasant happens the library throws the *Patlib_error* exception. The exception class is an offspring of **string** and it has *what*(**void**) function defined. The *what* function prints the text the exception has been constructed with to the standard error output. The header also includes the **exception** although it doesn't use it.

```
⟨ptl_exc.h 4⟩ ≡
#ifndef PTL_EXC_H
#define PTL_EXC_H
#include <iostream>
#include <string>
#include <exception>
  class Patlib_error : public std::string {
  public: Patlib_error(string s)
    : std::string(s) {      /* the constructor */
    }
    void what(void)
    {     /* output of the exception text */
      cout ≪ *this;
    }
  };
#endif
```

5. Finite language store. The first thing we need to do is to prepare the way to store a finite language. For the purpose of pattern generating the trie data structure seems to be a good choice as it behaves well on not very long words.

6. We start with the Growing array object. This serves as a potentially infinite array, simply said it allocates more memory when touching a previously unaccessed field. It makes implementing the pattern manipulator easier as we don't have to take care of memory overflow.

7. The interesting things come next. The Trie pattern manipulator is a finite language store where each word may have an output information attached to its end. The simplest case, storing a dictionary, means that the output information is a boolean value, *true* as end of the word. The main public-accessible methods include inserting a pattern (a word, if you want to), "walk through"—it means getting the language word by word, getting outputs of words, and some optimizing the structure after delete operations.

Sometimes the one "output field" for a word may suit us, sometimes we need pairs of values. Therefore we provide the Candidate count trie. The name goes from the main usage of that structure in our generator, we need to store pairs of numbers when preparing candidate patterns. Nevertheless the structure may be used with any types it can be compiled with, practically with numerical ones. The methods here makes only access to the pairs of values easier.

8. The "heavy weights" follow. The word may also have multiple outputs, e.g., on different positions. We provide the output store. We suppose the number of different outputs will be relatively small, therefore the manipulator with multi-output words consists of two parts, the word manipulator and the output manipulator, the outputs of a word become iterators to the output manipulator. The Multi output pattern manipulator service is an envelope hiding the inner parts. It makes the interface to word manipulations known from the word manipulator itself and adds several ways of deleting patterns depending on their outputs.

The offspring, competitive variant of the manipulator, supposes moreover the output alphabet to be sorted. It provides methods to compute the competitive variant of the pattern output, it means that the higher values win over the smaller ones.

9. And for completeness, we have to drop a word on the simple translate service. When testing applications where the word alphabet is simple we often need the alphabet to be "translated" into $\{1,\ldots,n\}$ numeric set. The service does it simply and slowly.

10. Growing array. Growing array is a service for trie pattern manipulator.

```
⟨ ptl_ga.h 10 ⟩ ≡
#ifndef PTL_GA_H    /* To include the header only once */
#define PTL_GA_H
#include <iostream>
#include <vector>
#include "ptl_exc.h"
  ⟨ Growing array (head) 11 ⟩
#endif
```

11. Growing array. We need an automatically growing array of given type. We want to add to the array mostly, deallocating is done at the end of work. The growing array is addressed by linear address and it grows when addressing previously unaccessed member. Addressing is done with overloaded '[]' operator.

We implement it as a vector of members. When a new member is added, it is initialized to the value given to the constructor. Therefore we need the **Tmember** object to have a '=' copy constructor.

The **Tindex** type is a type of array index (for avoiding compilation warnings, and for historical reasons), **Tmember** is a type of member.

```
  format Tindex int
  format Tmember int
```

```
⟨ Growing array (head) 11 ⟩ ≡
  template⟨class Tindex, class Tmember⟩
  class Growing_array :
  public std::vector⟨Tmember⟩ {
  protected:
    const Tmember default_value;
    ⟨ Growing array: constructor 12 ⟩
    ⟨ Growing array: operator[] 13 ⟩
    ⟨ Growing array: print statistics 14 ⟩
  };
```

This code is used in section 10.

12. The constructor prepares the default value and constructs the vector.

```
⟨ Growing array: constructor 12 ⟩ ≡
public:
  Growing_array(const Tmember &def_val)
  : std::vector⟨Tmember⟩(), default_value(def_val)
  { }
```

This code is used in section 11.

13. Operator [] accesses the array like an usual array, if the accessed member is not allocated, it prepares it.

⟨ Growing array: operator[] 13 ⟩ ≡
```
public:
  inline Tmember &operator[](const Tindex &logical_addr)
  {
    try {
      while (logical_addr ≥ std::vector⟨Tmember⟩::size())
        std::vector⟨Tmember⟩::push_back(default_value);
      return std::vector⟨Tmember⟩::operator[](logical_addr);
    }
    catch(...)
    {
      cerr ≪ endl ≪ "!␣Growing␣array:␣error␣in␣[]␣operator" ≪ endl;
      cerr ≪ "␣␣Logical␣member␣requested␣" ≪ logical_addr ≪ endl;
      cerr ≪ "␣␣No.␣of␣members␣used␣␣␣␣␣␣" ≪ std::vector⟨Tmember⟩::size() ≪ endl;
      cerr ≪ "␣␣No.␣of␣members␣reserved␣␣" ≪ std::vector⟨Tmember⟩::capacity() ≪ endl;
      cerr ≪ "␣␣Member␣size␣(bytes)␣␣␣␣␣␣" ≪ sizeof(Tmember) ≪ endl;
      throw Patlib_error("");
    }
  }
```
This code is used in section 11.

14. Sometimes we want to know the statistics.

⟨ Growing array: print statistics 14 ⟩ ≡
```
public:
  void print_statistics(void) const
  {
    cout ≪ "␣␣No.␣of␣members␣used␣␣␣␣␣␣␣" ≪ std::vector⟨Tmember⟩::size() ≪ endl;
    cout ≪ "␣␣No.␣of␣members␣reserved␣␣␣" ≪ std::vector⟨Tmember⟩::capacity() ≪ endl;
    cout ≪ "␣␣Member␣size␣(bytes)␣␣␣␣␣␣␣" ≪ sizeof(Tmember) ≪ endl;
  }
```
This code is used in section 11.

15. Trie pattern manipulators. The trie pattern manipulator is a finite language store, its offspring, candidate count trie, is only modified to handle pairs of values easily.

```
⟨ptl_tpm.h 15⟩ ≡
#ifndef PTL_TPM_H
#define PTL_TPM_H
#include <iostream>
#include <vector>
#include <set>
#include "ptl_exc.h"
#include "ptl_ga.h"
  ⟨Trie pattern manipulator (head) 16⟩
  ⟨Candidate count trie (head) 37⟩
#endif
```

16. Dynamic packed trie pattern manipulator. This is the implementation of pattern manipulator.

For a pattern $p_1 \ldots p_k$, the information associated with that pattern is accessed by setting $t_1 = \mathit{trie\_root}+p_1$ and then, for $1 < i \le k$, setting $t_i = \mathit{trie\_link}[t_{i-1}] + p_i$. The pattern output information is then stored in location addressed by t_k. Therefore if the information is associated to a transition and/or there is multiple output on various positions, the user must code this knowledge into the output.

Since all trie nodes are packed into a single array, we use a special field to distinguish nodes belonging to different families. Nodes belong to one family if and only if $\mathit{trie\_char}(t_i) = p_i$ for all i.

In addition the trie must support dynamic insertions and deletions. This is done by maintaining a doubly linked list of unoccupied cells and repacking trie families as necessary when insertions are made.

Each trie node consists of following fields:

- *trie_char* is the character of input alphabet,
- *trie_link* points to the next field,
- *trie_back* points to the next free cell, unused if position is used for a transition,
- *trie_outp* contains output of the pattern.

The types in the template must satisfy following conditions, if you really want the template instantiation to be compiled:

1 there must be commutative '+' operation defined between **Tpm_pointer** and **Tin_alph** and the result must be of **Tpm_pointer** type,
2 the **Tpm_pointer** must be ordered and must satisfy $0 \le t$ for all used t of **Tpm_pointer**, the 0 value is special and must be represented as 'nothing'
3 the **Tin_alph** must satisfy $0 \le t \le \mathit{max\_in\_alph}$ for all used t of **Tin_alph** type, the 0 value must not be used in pattern
4 the *out_inf_zero* is a special value meaning 'no output is there'
5 all the template types must have '=' copy constructor
6 we recommend the **Tin_alph** to be a subtype of **Tpm_pointer**,
7 we recommend the types to be unsigned numeric ones and to translate the application alphabet into numbers
(Still wanting to use anything else then numbers?)

```
format Tpm_pointer int
format Tin_alph int
format Tout_information int
```

```
⟨ Trie pattern manipulator (head) 16 ⟩ ≡
  template⟨class Tpm_pointer, class Tin_alph, class Tout_information⟩
  class Trie_pattern_manipulator {
    ⟨ Trie pattern manipulator: data structures 17 ⟩
    ⟨ Trie pattern manipulator: gets and sets 18 ⟩
    ⟨ Trie pattern manipulator: constructor 19 ⟩
    ⟨ Trie pattern manipulator: destructor 20 ⟩
    ⟨ Trie pattern manipulator: first fit 21 ⟩
    ⟨ Trie pattern manipulator: unpack 24 ⟩
    ⟨ Trie pattern manipulator: hard insert pattern 25 ⟩
    ⟨ Trie pattern manipulator: walking through—data and init 27 ⟩
    ⟨ Trie pattern manipulator: get next pattern 28 ⟩
    ⟨ Trie pattern manipulator: word output 31 ⟩
    ⟨ Trie pattern manipulator: word last output 32 ⟩
    ⟨ Trie pattern manipulator: delete hanging 33 ⟩
    ⟨ Trie pattern manipulator: set of my outputs 35 ⟩
    ⟨ Trie pattern manipulator: print statistics 36 ⟩
  };
```

This code is used in section 15.

17. Data structures of the pattern manipulator:

- *trie_max* maximal used trie position
- *trie_bmax* maximal used base position
- *trie_count* number of occupied trie nodes, for statistics
- *pat_count* number of patterns in the trie, for statistics
- *growing_array*s having character, links to and fro, boolean indicator whether the location is used as base and a field of **Tout_information** type used for various needed output and/or count information.
- *trieq_* ∗ arrays for unpacking a state when repack is needed and to pack a new state. The arrays are dynamic, initialized in the constructor.
- *q_max_thresh* value is used to decide where to pack the state stored in *trieq_* ∗ arrays. If this state is too dense, we pack it to the end, otherwise we try to pack it among other states in trie.

```
⟨ Trie pattern manipulator: data structures 17 ⟩ ≡
protected:
  enum {
    min_in_alph = 0
  };    /* it's ugly, but it took me an hour to find out what the linker means */
  enum {
    trie_root = 1
  };    /* this enum means the same as "static const" and this syntax is more portable */
  Tpm_pointer trie_max;
  Tpm_pointer trie_bmax;
  Tpm_pointer trie_count;
  Tpm_pointer pat_count;
  const Tin_alph max_in_alph;
  const Tout_information out_inf_zero;
  Growing_array⟨Tpm_pointer, Tin_alph⟩ trie_char;
  Growing_array⟨Tpm_pointer, Tpm_pointer⟩ trie_link;
  Growing_array⟨Tpm_pointer, Tpm_pointer⟩ trie_back;
  Growing_array⟨Tpm_pointer, char⟩ trie_base_used;
    /* one byte per a bit is a feasible solution (we cannot use vector⟨bool⟩) */
  Growing_array⟨Tpm_pointer, Tout_information⟩ trie_outp;
  unsigned q_max;
  unsigned q_max_thresh;
  Tin_alph *trieq_char;
  Tpm_pointer *trieq_link;
  Tpm_pointer *trieq_back;
  Tout_information *trieq_outp;
```

This code is used in section 16.

18. Several values may be obtained and set by users. The *q_max_thresh* variable controls the *first_fit* packing, the “count” variables are for statistics only.

```
⟨ Trie pattern manipulator: gets and sets 18 ⟩ ≡
public:
  virtual unsigned get_q_max_thresh(void) const
  {
    return q_max_thresh;
  }
  virtual void set_q_max_thresh(const unsigned &new_q_m_t)
  {    /* q_max_thresh must be at least 1 (even though 1 is quite a stupid choice) */
    if (new_q_m_t > 0) q_max_thresh = new_q_m_t;
  }
  virtual Tpm_pointer get_trie_count(void) const     /* get the number of occupied nodes */
  {
    return trie_count;
  }
  virtual Tpm_pointer get_pat_count(void) const     /* get the number of patterns */
  {
    return pat_count;
  }
  virtual Tpm_pointer get_max_in_alph(void) const     /* get the maximum input alphabet number */
  {
    return max_in_alph;
  }
```

This code is used in section 16.

19. Initially, the dynamic packed trie has the only state, the root, with all transitions present but with null links. We have to set up only the *trie_char* array as the other arrays are set to the default values when initializating the arrays.

```
⟨ Trie pattern manipulator: constructor 19 ⟩ ≡
public:
  Trie_pattern_manipulator(const Tin_alph &max_i_a, const Tout_information &out_i_z, const
          unsigned &q_thr = 5)
  :    /* set default values to embedded objects */
  max_in_alph(max_i_a), out_inf_zero(out_i_z), trie_char(min_in_alph), trie_link(min_in_alph),
      trie_back(min_in_alph), trie_base_used(false), trie_outp(out_inf_zero), q_max_thresh(q_thr)
  {
    for (Tpm_pointer c = min_in_alph; c ≤ max_in_alph; c++) {
      trie_char[trie_root + c] = c;
    }    /* In this cycle there was a very nice bug. The type of c was Tin_alph. The problem appeared
           when Tin_alph become unsigned char and max_in_alph was 255. This makes infinite cycle
           then.... This bug lived here more than half a year. BTW, I discovered it on Friday, 13 July
           2001. And when writing this text, my Emacs dumped core. Strange :-) */
    trie_base_used[trie_root] = true;
    trie_bmax = trie_root;
    trie_max = trie_root + max_in_alph;
    trie_count = max_in_alph + 1;
    pat_count = 0;
    trie_link[0] = trie_max + 1;
    trie_back[trie_max + 1] = 0;
    trieq_char = new Tin_alph[max_in_alph + 1];    /* init trieq arrays */
    trieq_link = new Tpm_pointer[max_in_alph + 1];
    trieq_back = new Tpm_pointer[max_in_alph + 1];
    trieq_outp = new Tout_information[max_in_alph + 1];
  }
```

This code is used in section 16.

20. We have to destroy dynamic arrays in the destructor.

```
⟨ Trie pattern manipulator: destructor 20 ⟩ ≡
public:
  virtual ~Trie_pattern_manipulator( )
  {
    delete[ ] trieq_char;
    delete[ ] trieq_link;
    delete[ ] trieq_back;
    delete[ ] trieq_outp;
  }
```

This code is used in section 16.

21. The *first_fit* procedure find a hole on the packed trie into which state in *trieq_* ∗ arrays will fit. This is normally done by going through the linked list of unoccupied cells and testing if the state will fit at each position. However if the state is too dense, we just pack it to the first unoccupied region (at $trie\_max + 1$). The state is considered dense if it has more than *q_max_thresh* states.

```
⟨ Trie pattern manipulator: first fit 21 ⟩ ≡
  protected:
  virtual Tpm_pointer first_fit(void){
      unsigned int q;
      Tpm_pointer s, t;
      ⟨ Trie pattern manipulator: (first fit) set s to the trie base location at which this state should be
          packed 22 ⟩    /* pack it */
      for (q = 1; q ≤ q_max; q++) {
        t = s + trieq_char[q];    /* link around filled cell */
        trie_link[trie_back[t]] = trie_link[t];
        trie_back[trie_link[t]] = trie_back[t];
        trie_char[t] = trieq_char[q];
        trie_link[t] = trieq_link[q];
        trie_back[t] = trieq_back[q];
        trie_outp[t] = trieq_outp[q];
        if (t > trie_max) {
          trie_max = t;
        }
      }
      trie_base_used[s] = true;
      return s;
      }
```

This code is used in section 16.

22. The threshold is used to decide what to do. We may pack the state to the end of used region or pack it among other transitions.

```
⟨ Trie pattern manipulator: (first fit) set s to the trie base location at which this state should be
        packed 22 ⟩ ≡
  if (q_max > q_max_thresh) {
    t = trie_back[trie_max + 1];
  } else {
    t = 0;
  }
  while (1) {      /* I don't like goto, but do it efficiently without it: */
  t = trie_link[t];      /* get next unoccupied cell */
  s = t − trieq_char[1];
  ⟨ Trie pattern manipulator: (first fit) ensure trie linked up to s + max_in_alph + 1 23 ⟩
  if (trie_base_used[s]) {
    goto not_found;
  }
  for (q = q_max; q ≥ 2; q−−) {      /* check if state fits there */
    if (trie_char[s + trieq_char[q]] ≠ min_in_alph) {
      goto not_found;
    }
  }
  goto found;
not_found: ;      /* go to the next loop */
  }
found: ;
```

This code is used in section 21.

23. The trie is only initialized (as a doubly linked list of empty cells) as far as necessary. Here we extend the initialization.

```
⟨ Trie pattern manipulator: (first fit) ensure trie linked up to s + max_in_alph + 1 23 ⟩ ≡
  while (trie_bmax < s) {
    trie_bmax++;
    trie_base_used[trie_bmax] = false;
    trie_char[trie_bmax + max_in_alph] = min_in_alph;
    trie_outp[trie_bmax + max_in_alph] = out_inf_zero;
      /* is this necessary? it is done by growing array! */
    trie_link[trie_bmax + max_in_alph] = trie_bmax + max_in_alph + 1;
    trie_back[trie_bmax + max_in_alph + 1] = trie_bmax + max_in_alph;
  }
```

This code is used in section 22.

24. The *unpack* procedure finds all transitions associated with the state based on s and puts them into *trieq_* $*$ arrays and sets *q_max* to one more than the number of transitions found. Freed cells are put at the beginning of the free list.

```
⟨ Trie pattern manipulator: unpack 24 ⟩ ≡
protected:
  virtual void unpack(const Tpm_pointer &s)
  {
    Tpm_pointer t;
    q_max = 1;
    for (Tpm_pointer c = min_in_alph; c ≤ max_in_alph; c++) {
      t = s + c;
      if (trie_char[t] ≡ c ∧ c ≠ min_in_alph) {      /* found one */
        trieq_char[q_max] = c;
        trieq_link[q_max] = trie_link[t];
        trieq_back[q_max] = trie_back[t];
        trieq_outp[q_max] = trie_outp[t];
        q_max++;      /* now free trie node */
        trie_back[trie_link[0]] = t;
        trie_link[t] = trie_link[0];
        trie_link[0] = t;
        trie_back[t] = 0;
        trie_char[t] = min_in_alph;
        trie_outp[t] = out_inf_zero;      /* not needed */
      }
    }
    trie_base_used[s] = false;
  }
```

This code is used in section 16.

25. Here is a procedure that inserts a pattern into the pattern trie. The pattern is a **vector** of **Tin_alph** symbols and adjacent **Tout_information** is put to the end of the pattern in the trie.

By hard inserting we mean that previous output is rewritten. We sometimes want to add to the output, so we have to get the last output of the word, change the output and "hard" insert it back.

```
⟨ Trie pattern manipulator: hard insert pattern 25 ⟩ ≡
public:
  virtual void hard_insert_pattern(const vector⟨Tin_alph⟩ &w, const Tout_information &o)
  {
    if (w.empty()) return;      /* if you want to insert a void pattern, we will ignore you */
    Tpm_pointer s, t;
    vector⟨Tin_alph⟩::const_iterator i = w.begin();
    s = trie_root + *i;
    t = trie_link[s];
    while ((t > 0) ∧ ((i + 1) ≠ w.end())) {      /* follow existing trie */
      i++;
      t += *i;
      if (trie_char[t] ≠ *i) {
        ⟨ Trie pattern manipulator: (hard insert pattern) insert critical transition, possibly repacking 26 ⟩
      }
      s = t;
      t = trie_link[s];
    }
    trieq_link[1] = 0;
    trieq_back[1] = 0;
    trieq_outp[1] = out_inf_zero;
    q_max = 1;
    while ((i + 1) ≠ w.end()) {      /* insert rest of pattern */
      i++;
      trieq_char[1] = *i;
      t = first_fit();
      trie_link[s] = t;
      s = t + *i;
      trie_count++;
    }
    if ((trie_outp[s] ≡ out_inf_zero) ∧ (o ≠ out_inf_zero)) {      /* we rewrite "no-pattern" by "pattern" */
      pat_count++;
    }
    if ((trie_outp[s] ≠ out_inf_zero) ∧ (o ≡ out_inf_zero)) {
        /* we rewrite "pattern" by "no-pattern", we delete in fact */
      pat_count--;
    }
    trie_outp[s] = o;
  }
```

This code is used in section 16.

26. We have accessed transition not in the trie. We insert it, repacking the state if necessary.

⟨ Trie pattern manipulator: (hard insert pattern) insert critical transition, possibly repacking 26 ⟩ ≡

```
  if (trie_char[t] ≡ min_in_alph) {      /* no repacking needed */
    trie_link[trie_back[t]] = trie_link[t];
    trie_back[trie_link[t]] = trie_back[t];
    trie_char[t] = *i;
    trie_link[t] = 0;
    trie_back[t] = 0;
    trie_outp[t] = out_inf_zero;
    if (t > trie_max) {
      trie_max = t;
    }
  }
  else {      /* whoops, have to repack */
    unpack(t − *i);
    trieq_char[q_max] = *i;
    trieq_link[q_max] = 0;
    trieq_back[q_max] = 0;
    trieq_outp[q_max] = out_inf_zero;
    t = first_fit();
    trie_link[s] = t;
    t += *i;
  }
  trie_count++;
```

This code is used in section 25.

27. Walking through the pattern manipulator language. This is something like an "iterator", we init walk through and then we may get patterns and their outputs iteratively, one by one. We use a stack where we store current path in the manipulator. The *init_walk_through* procedure sets the initial stack values to start searching.

⟨ Trie pattern manipulator: walking through—data and init 27 ⟩ ≡

```
protected:
  vector⟨Tpm_pointer⟩ pointer_stack;
  vector⟨Tpm_pointer⟩ char_stack;
public:
  virtual void init_walk_through()
  {
    pointer_stack.clear();
    char_stack.clear();
    pointer_stack.push_back(trie_root);
    char_stack.push_back(min_in_alph);
  }
```

This code is used in section 16.

28. Getting the next pattern is here. When we reach state having output we stop depth-first-search and return the values. When *get_next_pattern* is called again, it finds next pattern. If find was successful, *true* is returned by *get_next_pattern*, otherwise *false*. It allows to use construction **while** (*get_next_pattern*(...)). The pattern and its output is returned in *w* and *o* variables. Moreover when no more pattern remains (*false* is returned), the *w* vector is emptied and *o* is set to *out_inf_zero*.

Important: Inserting and deleting patterns may interfere with the walk-through process. Nevertheless you may change the output of an *existing* pattern independently on the walk-through process using *hard_insert_pattern*.

And never forget to use *init_walk_through* first!

```
⟨ Trie pattern manipulator: get next pattern 28 ⟩ ≡
public:
  virtual bool get_next_pattern(vector⟨Tin_alph⟩ &w, Tout_information &o)
  {
    Tpm_pointer t, tstart;
    Tpm_pointer c, cstart;
    w.clear();
    o = out_inf_zero;
    while (true) {
      if (pointer_stack.empty()) return false;
      tstart = pointer_stack.back();
      pointer_stack.pop_back();      /* we have where to go */
      cstart = char_stack.back();
      char_stack.pop_back();
      for (c = cstart; c ≤ max_in_alph; c++) {      /* find transitions belonging to this family */
        t = tstart + c;
        if (trie_char[t] ≡ c ∧ c ≠ min_in_alph) {      /* found one */
          pointer_stack.push_back(tstart);
          char_stack.push_back(c + 1);
          if (trie_outp[t] ≠ out_inf_zero) {      /* state with output */
            ⟨ Trie pattern manipulator: (get next pattern) output the pattern and belonging output 30 ⟩
              /* state found successfully */
            ⟨ Trie pattern manipulator: (get next pattern) go deeper if possible 29 ⟩
            return true;
          }
          ⟨ Trie pattern manipulator: (get next pattern) go deeper if possible 29 ⟩
          break;
        }
      }
    }
  }
```

This code is used in section 16.

29. Prepare the way to go deeper in the trie.

```
⟨ Trie pattern manipulator: (get next pattern) go deeper if possible 29 ⟩ ≡
  if (trie_link[t] ≠ 0) {      /* we have to go deeper, if possible */
    pointer_stack.push_back(trie_link[t]);
    char_stack.push_back(min_in_alph);
  }
```

This code is used in section 28.

30. We have come to the end of a pattern, so we now output it. We fill the *w* vector with the pattern and the *o* with the output of the pattern.

Note that the last stack value is ignored by this routine as it shows where to go next. The value in *char_stack* is the character plus one.

```
⟨ Trie pattern manipulator: (get next pattern) output the pattern and belonging output 30 ⟩ ≡
  vector⟨Tpm_pointer⟩::iterator it = pointer_stack.begin( );
  vector⟨Tpm_pointer⟩::iterator ic = char_stack.begin( );
  Tpm_pointer u;     /* now let's go to the last-to-end stack character */
  for (Tpm_pointer i = 0; i < pointer_stack.size( ); i++) {
    u = (*it) + Tpm_pointer(*ic) − 1;     /* in stack, there is one more */
    w.push_back(trie_char[u]);
    it++;
    ic++;
  }
  o = trie_outp[u];
```

This code is used in section 28.

31. This is a procedure to produce output of pattern on given word. All outputs of patterns matching the beginning of the *w* are pushed back to the *o* vector to the same position. If no output is associated with this transition, *o* vector is filled with *out_inf_zero*. If the *w* is longer than any matching pattern, the *o* vector is filled to the same length with the *out_inf_zero* value.

Note that this is not the same as "hyphenate" procedure, we need to use *word_output* with all postfixes of the word to get complete information. This function must nevertheless be implemented in application layer of pattern generator as the pattern manipulator does not know anything about the semantics of output information. So we are not able to make the patterns outputs compete here, we do not know the order of the symbols.

```
⟨ Trie pattern manipulator: word output 31 ⟩ ≡
public:
  void word_output(const vector⟨Tin_alph⟩ &w, vector⟨Tout_information⟩ &o)
  {
    Tpm_pointer t;
    vector⟨Tin_alph⟩::const_iterator i = w.begin( );
    o.clear( );
    if (w.empty( )) return;
    t = trie_root;
    do {     /* follow trie and get the outputs */
      t += *i;
      if (trie_char[t] ≡ *i) o.push_back(trie_outp[t]);
      else break;
      t = trie_link[t];
      i++;
    } while (t ≠ 0 ∧ i ≠ w.end( ));
    while (i ≠ w.end( )) {     /* fill outputs to the same length */
      o.push_back(out_inf_zero);
      i++;
    }
  }
```

This code is used in section 16.

32. This procedure gives only the last output of a word. It is equivalent to previous procedure if you use only the last field of *o* vector, but this is optimized.

So the last output is given, if no output of the word exists, *out_inf_zero* is returned.

```
⟨ Trie pattern manipulator: word last output 32 ⟩ ≡
public:
  void word_last_output(const vector⟨Tin_alph⟩ &w, Tout_information &o)
  {
#if 0 ≡ 1     /* unoptimized version */
    o = out_inf_zero;
    vector⟨Tout_information⟩ whole_o;
    word_output(w, whole_o);
    if (whole_o.size() ≥ 1) o = *(whole_o.end() − 1);
#endif
    Tpm_pointer s, t;
    vector⟨Tin_alph⟩ :: const_iterator i = w.begin();
    o = out_inf_zero;
    if (w.empty()) return;
    t = trie_root;     /* follow the trie */
    do {
      t += *i;
      if (trie_char[t] ≡ *i) s = t;
      else break;
      t = trie_link[t];
      i++;
    } while (t ≠ 0 ∧ i ≠ w.end());     /* if we are at the end of the word, we have the output */
    if (i ≡ w.end()) {
      o = trie_outp[s];
    }
  }
```

This code is used in section 16.

33. We delete patterns by overwriting their outputs by *out_inf_zero*. Hanging parts of patterns stay in their places. They may be removed at once if we want to reduce the number of occupied cells. The *delete_hanging* procedure removes the nodes which have no output or continue by branches with no output.

The following recursive procedure is a subroutine run from the root node of the trie by public-accessible procedure *delete_hanging* and returns *true* if and only if entire subtrie is removed.

```
⟨ Trie pattern manipulator: delete hanging 33 ⟩ ≡
private:
  virtual bool delete_hanging_level(const Tpm_pointer &s)
  {
    Tpm_pointer t;
    bool all_freed;
    all_freed = true;
    for (Tpm_pointer c = min_in_alph; c ≤ max_in_alph; c++) {
        /* find transitions belonging to this family */
      t = s + c;
      if (trie_char[t] ≡ c ∧ c ≠ min_in_alph) {      /* found one */
        if (trie_link[t] ≠ 0) {
          if (delete_hanging_level(trie_link[t])) trie_link[t] = 0;
        }
        if ((trie_link[t] ≠ 0) ∨ (trie_outp[t] ≠ out_inf_zero) ∨ (s ≡ trie_root)) all_freed = false;
        else ⟨ Trie pattern manipulator: (delete hanging) deallocate this node 34 ⟩
      }
    }
    if (all_freed) {      /* entire state is freed */
      trie_base_used[s] = false;
    }
    return all_freed;
  }
public:
  virtual void delete_hanging(void)
  {
    delete_hanging_level(trie_root);
  }
```

This code is used in section 16.

34. Cells freed by *delete_hanging_level* are put at the end of the free list.

```
⟨ Trie pattern manipulator: (delete hanging) deallocate this node 34 ⟩ ≡
  {
    trie_link[trie_back[trie_max + 1]] = t;
    trie_back[t] = trie_back[trie_max + 1];
    trie_link[t] = trie_max + 1;
    trie_back[trie_max + 1] = t;
    trie_char[t] = min_in_alph;
    trie_count--;
  }
```

This code is used in section 33.

35. We sometimes need to know all the outputs used in the pattern manipulator. We may need it for example to clean complicated output object which are no longer referenced. The **set**⟨**Tout_information**⟩ is first cleared and then filled up with all the values used as outputs.

```
⟨ Trie pattern manipulator: set of my outputs 35 ⟩ ≡
public:
  void set_of_my_outputs(set⟨Tout_information⟩ &s)
  {
    s.clear( );
    for (Tpm_pointer i = 0; i ≤ trie_max; i++) {
      s.insert(trie_outp[i]);
    }
  }
```

This code is used in section 16.

36. Print statistics on trie structure. If *detail* is non-zero, statistics on manipulator internal structures are also printed.

```
⟨ Trie pattern manipulator: print statistics 36 ⟩ ≡
public:
  void print_statistics(int detail = 0) const
  {
    cout ≪ "␣␣nodes:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" ≪ trie_count ≪ endl;
    cout ≪ "␣␣patterns:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" ≪ pat_count ≪ endl;
    cout ≪ "␣␣trie_max:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" ≪ trie_max ≪ endl;
    cout ≪ "␣␣current␣q_max_thresh:␣␣␣␣␣␣␣␣␣␣␣␣" ≪ q_max_thresh ≪ endl;
    if (detail) {
      cout ≪ "Trie␣char" ≪ endl;
      trie_char.print_statistics( );
      cout ≪ "Trie␣link" ≪ endl;
      trie_link.print_statistics( );
      cout ≪ "Trie␣back" ≪ endl;
      trie_back.print_statistics( );
      cout ≪ "Trie␣base␣used" ≪ endl;
      trie_base_used.print_statistics( );
      cout ≪ "Trie␣outp" ≪ endl;
      trie_outp.print_statistics( );
    }
  }
```

This code is used in section 16.

37. Candidate count trie. This trie is to collect statistics about the patterns generated in the current level.

The most useful statistics (and the statistics we use) is the pair of numbers *good* and *bad*, meaning the number of cases when the candidate pattern works well or badly in the context of patterns collected so far.

The main reason of providing this service is to hide uncomfortable work with pairs of values.

We suppose the candidate patterns stored in the count trie at the same time to have the same length and to be about the only position of interest. This condition is useful for us as it allows to simplify candidate pattern handling rapidly. We would have to have multiple outputs for a pattern and store information on what position the output is related to. It would also make some optimizations, namely "knocking out", much more complicated.

The **Tcount_good** and **Tcount_bad** template argument is a number type of *good* and *bad* values. For information on the other arguments see the **Trie_pattern_manipulator** definition.

```
format Tpm_pointer int
format Tin_alph int
format Tcount_good int
format Tcount_bad int
```

⟨ Candidate count trie (head) 37 ⟩ ≡
```
  template⟨class Tpm_pointer, class Tin_alph, class Tcount_good, class Tcount_bad⟩
  class Candidate_count_trie :
  public Trie_pattern_manipulator⟨Tpm_pointer, Tin_alph, pair⟨Tcount_good, Tcount_bad⟩⟩ {
  public:
    typedef pair⟨Tcount_good, Tcount_bad⟩ Tcount_pair;
    ⟨ Candidate count trie: constructor 38 ⟩
    ⟨ Candidate count trie: increment counts 39 ⟩
    ⟨ Candidate count trie: get next pattern 40 ⟩
  };
```
This code is used in section 15.

38. We have to initialize the default values. The default packing threshold is less (at least we recommend it) then in **Trie_pattern_manipulator** as speed is more important here.

⟨ Candidate count trie: constructor 38 ⟩ ≡
```
public:
  Candidate_count_trie(const Tin_alph &max_i_a, const Tcount_good &out_i_z_good, const
          Tcount_bad &out_i_z_bad, const unsigned &q_thr = 3)
  : Trie_pattern_manipulator⟨Tpm_pointer, Tin_alph, Tcount_pair⟩(max_i_a,
      make_pair(out_i_z_good, out_i_z_bad), q_thr)
  { }
```
This code is used in section 37.

39. The following procedures increment the good and the bad counts in patterns. If you want to change the only one parameter, change the other with zero :-). If the pattern has not been in the trie, it is inserted and the counts are added to the default values.

The procedure may be optimized, we may manipulate the data directly. In the way we do it the trie must be traversed twice. This is conceptually nicer.

```
⟨ Candidate count trie: increment counts 39 ⟩ ≡
public:
  virtual void increment_counts(const vector⟨Tin_alph⟩ &w, const Tcount_good &good_inc, const
          Tcount_bad &bad_inc)
  {
    Tcount_pair counts;
    word_last_output(w, counts);
    counts.first += good_inc;
    counts.second += bad_inc;
    hard_insert_pattern(w, counts);      /* now we overwrite the output or insert it if it was void */
  }
```

This code is used in section 37.

40. Get next pattern overloads the procedure from the **Trie_pattern_manipulator**, the *good* and *bad* counts are returned instead of a pair of them. Please see discussion on the original method to learn how to use it. Do not forget to run *init_walk_through* first!

```
⟨ Candidate count trie: get next pattern 40 ⟩ ≡
public:
  virtual bool get_next_pattern(vector⟨Tin_alph⟩ &w, Tcount_good &good, Tcount_bad &bad)
  {
    Tcount_pair counts;
    bool ret_val;
    ret_val = Trie_pattern_manipulator⟨Tpm_pointer, Tin_alph, pair⟨Tcount_good,
        Tcount_bad⟩⟩ :: get_next_pattern(w, counts);
    good = counts.first;
    bad = counts.second;
    return ret_val;
  }
```

This code is used in section 37.

41. Multiple output pattern handling. The pattern manipulators we have seen in previous sections are able to keep the only (well, sometimes a pair of) thing as the output. But we need outputs on different positions of a word. Therefore we provide simple methods to keep outputs of word positions and the manipulator with such kind of language.

```
⟨ ptl_mopm.h 41 ⟩ ≡
#ifndef PTL_MOPM_H
#define PTL_MOPM_H
#include <iostream>
#include <map>
#include <set>
#include "ptl_tpm.h"
  ⟨ Outputs of a pattern (head) 42 ⟩
  ⟨ Outputs of patterns (head) 43 ⟩
  ⟨ Multi output pattern manipulator (head) 44 ⟩
  ⟨ Competitive multi out pat manip (head) 57 ⟩
#endif
```

42. Multiple pattern outputs. The pattern manipulator recognizes the end of a word by having the output on that position. We have to store information on what position of the pattern the output is related to, e.g. the pattern is *qwerty*$(2,4)$ means that the pattern *qwerty* has output 4 after the second character, the usual form of writing this is *qw*4*erty*.

Another complication is that the input word may have several outputs, so we have to store *sets* of outputs instead of the outputs themselves. If we have pattern *qw*4*ert*5*y*, we store it as *qwerty*$\{(2,4),(5,5)\}$ into the manipulator.

We will do it really easily, we inherit all the operations we need from the standard container **multimap**. The **Tposition** is the position in a pattern (convention: the output before the first character is 0th, after the first character is 1st and so on). The **Toutput** is the real output of the position.

```
format Tposition int
format Toutput int
```

⟨ Outputs of a pattern (head) 42 ⟩ ≡

```
  template⟨class Tposition, class Toutput⟩
  class Outputs_of_a_pattern :
  public std::multimap⟨Tposition, Toutput⟩
  { };
```

This code is used in section 41.

43. We have not said the whole story yet. We may now store the previous objects as outputs of the **Trie_pattern_manipulator**. But the **Outputs_of_a_pattern** objects are quite large and there will not be a lot of different ones.

Therefore we put the **Outputs_of_a_pattern** into a **set** and we store iterators to the **set** as the trie outputs. The **set** iterators are guaranteed not to change when **set** operations are performed. (Of course the iterator becomes invalid if you delete a **set** member it points to, but nobody makes us do it.)

If the number of different outputs is supposed to be high we should not use this and we would better make the previously defined objects as outputs of the pattern manipulator.

Here we inherit the **set**. We also need the empty member to be present all the time. Therefore we provide the *empty_iter* value and in the constructor we create the empty member. The *empty_iter* value may be used as "no output is here" special value for the pattern manipulator. Of course, never remove the empty output field from the object.

Also printing statistics is here.

```
⟨ Outputs of patterns (head) 43 ⟩ ≡
  template⟨class Tposition, class Toutput⟩
  class Outputs_of_patterns :
  public std::set⟨Outputs_of_a_pattern⟨Tposition, Toutput⟩⟩
  {
  private:
    Outputs_of_patterns⟨Tposition, Toutput⟩::iterator empty_iter;
  public:
    Outputs_of_patterns(void)
    {
      Outputs_of_a_pattern⟨Tposition, Toutput⟩ empty;
      empty_iter = (this→insert(empty)).first;
    }
    Outputs_of_patterns⟨Tposition, Toutput⟩::iterator get_empty_iter(void) const
    {
      return empty_iter;
    }
    void print_statistics( ) const
    {
      cout ≪ "␣␣number␣of␣different␣outputs:␣␣␣␣␣" ≪ size( ) ≪ endl;
    }
  };
```

This code is used in section 41.

44. Multi-output pattern manipulator. This service provides a language storage with handling of multiple outputs for a word. A word may have outputs on different positions.

This is a composition of two objects, a pattern manipulator and an output handling service.

Well, why don't we inherit the pattern manipulator and override only things needed to different kind of output? It is a good question. The conceptual problem is, whether the multi-output manipulator is a special pattern manipulator, or if it has a pattern manipulator and an output manipulator. Both the cases are acceptable, I think. I would also prefer inheritance to composition in this case, but there is a problem.

If we inherited the pattern manipulator, we must call it's constructor and tell it the parameter *out_inf_zero*. Therefore we need to construct the outputs of patterns manipulator first. But we can't as the order of constructing subobject is strict. We would have to inherit the output manipulator first, which is conceptually absolutely stupid as the multi-output manipulator is not an output manipulator.

It also brings more writing for me as public procedures of the pattern manipulator must have their interfaces here too. It's a real C++ puzzle.

```
format Tindex int
format Tin_alph int
format Tout_alph int
```

⟨ Multi output pattern manipulator (head) 44 ⟩ ≡

```
  template⟨class Tindex, class Tin_alph, class Tout_alph⟩
  class Multi_output_pattern_manipulator {
    ⟨ Multi output pattern manipulator: data 45 ⟩
    ⟨ Multi output pattern manipulator: constructor and destructor 46 ⟩
    ⟨ Multi output pattern manipulator: gets and sets 47 ⟩
    ⟨ Multi output pattern manipulator: walking through 48 ⟩
    ⟨ Multi output pattern manipulator: word output 49 ⟩
    ⟨ Multi output pattern manipulator: word last output 50 ⟩
    ⟨ Multi output pattern manipulator: insert pattern 51 ⟩
    ⟨ Multi output pattern manipulator: delete values 52 ⟩
    ⟨ Multi output pattern manipulator: delete position 53 ⟩
    ⟨ Multi output pattern manipulator: delete pattern 54 ⟩
    ⟨ Multi output pattern manipulator: delete hanging 55 ⟩
    ⟨ Multi output pattern manipulator: print statistics 56 ⟩
  };
```

This code is used in section 41.

45. Data structures of multi-output pattern manipulator. We have a pattern trie manipulator to hold input words and an output set to hold their outputs. The pattern manipulator has iterators to the set as the output information associated with a word.

⟨ Multi output pattern manipulator: data 45 ⟩ ≡

```
protected:
  typedef Outputs_of_patterns⟨Tindex, Tout_alph⟩::iterator Tout_iter;
  Outputs_of_patterns⟨Tindex, Tout_alph⟩ outputs;
  Trie_pattern_manipulator⟨Tindex, Tin_alph, Tout_iter⟩ words;
```

This code is used in section 44.

46. In the constructor, we have to set initial values to the embedded objects. The *words* manipulator is initialized with the *max_in_alph* value and the empty set iterator as the "zero output information." Once more: never remove the empty field from the *outputs* set, or strange things will happen.

The other constructor copies the manipulator. The "logical" content of the original (*old*) object is copied, so the copy may use less memory. Nevertheless the copying process is quite time-consuming as it is linear in number of all (position, value) pairs in the manipulator's language. The *old* object is passed by value and cannot be **const** as the walk-through procedure changes it's hidden data structures. But the language of *old* is not changed.

This copying may be also used to decrease the number of unused outputs of patterns as it is no other reasonable way to delete them, so we let them rest in peace even if they are no longer used.

```
⟨ Multi output pattern manipulator: constructor and destructor 46 ⟩ ≡
public:
  Multi_output_pattern_manipulator(const Tin_alph &max_i_a)
  : outputs( ), words(max_i_a, outputs.get_empty_iter( ))
  { }
  Multi_output_pattern_manipulator(Multi_output_pattern_manipulator &old)
  : outputs( ), words(old.get_max_in_alph( ), outputs.get_empty_iter( ))
  {
    vector⟨Tin_alph⟩ w;
    Outputs_of_a_pattern⟨Tindex, Tout_alph⟩ o;
    old.init_walk_through( );
    while (old.get_next_pattern(w, o))
      for (Outputs_of_a_pattern⟨Tindex, Tout_alph⟩::iterator i = o.begin( ); i ≠ o.end( ); i++)
        this→insert_pattern(w, i→first, i→second);
  }
  virtual ~Multi_output_pattern_manipulator( )
  {    /* nothing to do here */
  }
```

This code is used in section 44.

47. Several values may be public-available. Self-commenting, isn't it?

```
⟨ Multi output pattern manipulator: gets and sets 47 ⟩ ≡
public:
  virtual Tindex get_max_in_alph(void) const
  {
    return words.get_max_in_alph( );
  }
  virtual Tindex get_pat_count(void)
  {
    return words.get_pat_count( );
  }
```

This code is used in section 44.

48. Walking through the manipulator language. First, call the *init_walk_through* method and then you may *get_next_pattern*. The *get_next_pattern* procedure returns patterns with their outputs one by one. The return value is *true*. When no pattern remains, *false* is returned and w is an empty vector, o is empty, too.

This process may be broken by deleting and inserting patterns. The operation of changing outputs of an *existing* word is safe.

For full details, consult documentation of *get_next_pattern* method of the parent class. And again (it starts to be boring): never forget to call *init_walk_through* first.

⟨ Multi output pattern manipulator: walking through 48 ⟩ ≡

```
public:
  inline virtual void init_walk_through(void)
  {
    words.init_walk_through();
  }
  virtual bool get_next_pattern(vector⟨Tin_alph⟩ &w, Outputs_of_a_pattern⟨Tindex, Tout_alph⟩
        &o)
  {
    bool ret_val;
    Outputs_of_patterns⟨Tindex, Tout_alph⟩ :: iterator i;
    ret_val = words.get_next_pattern(w, i);
    o = *i;
    return ret_val;
  }
```

This code is used in section 44.

49. If we want to know the output of a word, we fill a **multimap** with (position, value) pairs. There may be more values per position, we cannot choose the highest one as we do not know the order.

Note this is not the "hyphenate" procedure, this returns outputs of all patterns matching the beginning of the w.

Well, how shall we do it? We get all outputs of the w from the *words* manipulator. It is a **vector** of **set iterator**s. Dereferencing them, we get **multimap**s having pairs (position, value). We collect them and put them into the **multimap**.

FIXME: It would be nicer to use a bit more optimal structure than vector when asking the *words* for outputs. There is a lot of *out_inf_zero* paddings.

⟨ Multi output pattern manipulator: word output 49 ⟩ ≡

```
public:
  void word_output(const vector⟨Tin_alph⟩ &w, Outputs_of_a_pattern⟨Tindex, Tout_alph⟩ &o)
  {
    vector⟨Tout_iter⟩ out_pointers;    /* iterators to outputs */
    words.word_output(w, out_pointers);
    o.clear();    /* no outputs */
    for (vector⟨Tout_iter⟩ :: const_iterator i = out_pointers.begin(); i ≠ out_pointers.end(); i++) {
        /* go through all outputs */
      for (Outputs_of_a_pattern⟨Tindex, Tout_alph⟩ :: const_iterator j = (*i)→begin();
           j ≠ (*i)→end(); j++) {    /* go through the map */
        o.insert(*j);
      }
    }
  }
```

This code is used in section 44.

50. We may also want only the last output. We get output of the *w* from the *words* manipulator. It is a **set iterator**. Dereferencing it, we get a **multimap** having pairs (position, value). We put them into the **multimap**.

```
⟨ Multi output pattern manipulator: word last output 50 ⟩ ≡
public:
  void word_last_output(const vector⟨Tin_alph⟩ &w, Outputs_of_a_pattern⟨Tindex, Tout_alph⟩
        &o)
  {
    Outputs_of_patterns⟨Tindex, Tout_alph⟩ :: iterator i;
    words.word_last_output(w, i);
    o = *i;
  }
```

This code is used in section 44.

51. We insert patterns in this way. We find the previous output set of the pattern, we copy the outputs, update them and insert into the set again. We are not allowed to do it in-place as we cannot change outputs of other patterns. The *p* parameter is word position associated with the *v* value.

If the *with_erase* is *true* the previous outputs are deleted. The value defaults to *false*.

```
⟨ Multi output pattern manipulator: insert pattern 51 ⟩ ≡
public:
  void insert_pattern(const vector⟨Tin_alph⟩ &w, const Tindex &p, const Tout_alph &v, bool
        with_erase = false)
  {
    Outputs_of_a_pattern⟨Tindex, Tout_alph⟩ o;
    word_last_output(w, o);     /* now we have the outputs of the word */
    if (with_erase) o.erase(p);
    o.insert(make_pair(p, v));     /* we add the new output to the copy and put it back */
    words.hard_insert_pattern(w, outputs.insert(o).first);
  }
```

This code is used in section 44.

52. Deleting patterns may be done in several ways. Here all outputs with given value v are removed. Nothing else is changed. (It is PATGENs deleting bad patterns.)

We walk through all the words, all their outputs. We take a copy of the output, delete unwanted values from there (by copying again) and put the output back using the *hard_insert_pattern* method of *word*. This overwrites the outputs at once.

⟨ Multi output pattern manipulator: delete values 52 ⟩ ≡

```
public:
  void delete_values(const Tout_alph &v)
  {
    vector⟨Tin_alph⟩ w;
    Outputs_of_a_pattern⟨Tindex, Tout_alph⟩ o;
    Outputs_of_a_pattern⟨Tindex, Tout_alph⟩ n;
    init_walk_through( );
    while (get_next_pattern(w, o)) {
      n.clear( );
      for (Outputs_of_a_pattern⟨Tindex, Tout_alph⟩::iterator i = o.begin( ); i ≠ o.end( ); i++)
        if (i→second ≠ v) n.insert(∗i);    /∗ delete given output ∗/
      words.hard_insert_pattern(w, outputs.insert(n).first);    /∗ put it back ∗/
    }
  }
```

This code is used in section 44.

53. We may also want to delete an output of a pattern on given position. This deletes all the output(s) of word w on position p.

⟨ Multi output pattern manipulator: delete position 53 ⟩ ≡

```
public:
  void delete_position(const vector⟨Tin_alph⟩ &w, const Tindex &p)
  {
    Outputs_of_a_pattern⟨Tindex, Tin_alph⟩ o;
    word_last_output(w, o);    /∗ outputs of a word ∗/
    o.erase(p);    /∗ erase all outputs with p as the index ∗/
    words.hard_insert_pattern(w, outputs.insert(o).first);    /∗ put it back ∗/
  }
```

This code is used in section 44.

54. The *delete_pattern* procedure removes all outputs of a word w.

⟨ Multi output pattern manipulator: delete pattern 54 ⟩ ≡

```
public:
  void delete_pattern(vector⟨Tin_alph⟩ &w)
  {
    words.hard_insert_pattern(w, outputs.get_empty_iter( ));
  }
```

This code is used in section 44.

55. The *delete_hanging* procedure removes the unused manipulator fields which have been left on their places after pattern deletions. This does not change the manipulator language.

Now the procedure does not dispose the outputs. We assume that the number of deleted outputs is much smaller to the number of outputs and the deleting procedure would be quite expensive.

```
⟨ Multi output pattern manipulator: delete hanging 55 ⟩ ≡
public:
  virtual void delete_hanging(void)
  {
    words.delete_hanging();
  }
```

This code is used in section 44.

56. Printing statistics (ehm, nothing to say here).

```
⟨ Multi output pattern manipulator: print statistics 56 ⟩ ≡
public:
  void print_statistics(void) const
  {
    words.print_statistics();
    outputs.print_statistics();
  }
```

This code is used in section 44.

57. Competitive multi output pattern manipulator. The multi output pattern manipulator does not know anything about the order of output symbols. To make the pattern output handling easy and to avoid unpleasant pattern output constructing in higher application levels, we provide following object enlarging the capabilities of the manipulator. The output information is a number (with standard order) in most cases, so this will be usually more suitable solution for multi output pattern handling.

Now we suppose the **Tout_alph** to have the "<" operation defined. The knowledge of order of the symbols lets us to make the *competitive_pattern_output* routine with at most one output information per position.

I personally prefer longer names, but overfull hboxes made me to use such a terrible abbreviation for this class.

```
  format Tindex int
  format Tin_alph int
  format Tout_alph int
⟨ Competitive multi out pat manip (head) 57 ⟩ ≡
  template⟨class Tindex, class Tin_alph, class Tout_alph⟩
  class Competitive_multi_out_pat_manip :
  public Multi_output_pattern_manipulator⟨Tindex, Tin_alph, Tout_alph⟩ {
    ⟨ Competitive multi out pat manip: constructor and destructor 58 ⟩
    ⟨ Competitive multi out pat manip: competitive word output 59 ⟩
    ⟨ Competitive multi out pat manip: competitive pattern output 60 ⟩
  };
```

This code is used in section 41.

58. Constructor and destructor have nothing interesting. They propagate the values to their parents. Also the copy constructor may be used to rebuild the data structures in more efficient way.

```
⟨ Competitive multi out pat manip: constructor and destructor 58 ⟩ ≡
public:
  Competitive_multi_out_pat_manip(const Tin_alph &max_i_a)
  : Multi_output_pattern_manipulator⟨Tindex, Tin_alph, Tout_alph⟩(max_i_a) { }
  Competitive_multi_out_pat_manip(Competitive_multi_out_pat_manip &old)
  : Multi_output_pattern_manipulator⟨Tindex, Tin_alph, Tout_alph⟩(old) { }
  ~Competitive_multi_out_pat_manip( )
  {    /* nothing to do here */
  }
```

This code is used in section 57.

59. We need to collect the intercharacter values in a bit different way, we take the new one if and only if the new one wins over the old one. Here we compute the word output o for the word w. Therefore we put the shift s value so that the position information may be stored according to the original word. Strictly speaking, all the positions in the word outputs are increased with s, the values are not affected by this process.

Note that this method does not delete the previously collected outputs, only changes the ones on appropriate positions. Go on reading the next method to understand what is this index mess good for.

Only values <*ignore_bigger* are collected and taken into account.

```
⟨ Competitive multi out pat manip: competitive word output 59 ⟩ ≡
protected:
  void competitive_word_output(const vector⟨Tin_alph⟩ &w, Outputs_of_a_pattern⟨Tindex,
          Tout_alph⟩ &o, const Tindex &s, const Tout_alph &ignore_bigger)
  {
    vector⟨Tout_iter⟩ out_pointers;     /* iterators to outputs */
    Outputs_of_a_pattern⟨Tindex, Tout_alph⟩ :: iterator oi;

    words.word_output(w, out_pointers);
    for (vector⟨Tout_iter⟩ :: const_iterator i = out_pointers.begin( ); i ≠ out_pointers.end( ); i++) {
        /* go through all outputs */
      for (Outputs_of_a_pattern⟨Tindex, Tout_alph⟩ :: const_iterator j = (*i)→begin( );
            j ≠ (*i)→end( ); j++) {     /* go through the map */
        if (j→second ≥ ignore_bigger) continue;     /* value to ignore */
        oi = o.find(s + (j→first));     /* find that position */
        if (oi ≡ o.end( ))     /* has not been there before */
          o.insert(make_pair(s + (j→first), j→second));
        else {     /* has been there */
          if (oi→second < j→second) {     /* we'll help */
            o.erase(s + (j→first));
            o.insert(make_pair(s + (j→first), j→second));
          }
        }
      }
    }
  }
```

This code is used in section 57.

60. This procedure computes all outputs *o* of all patterns matching the word *w*. The output contains at most one value per position, if there were more values per position, the maximum is output.

We go through all the postfixes of the word and collect (in the competitive sense) all the outputs using the previous routine. Only values <*ignore_bigger* are collected. It means if a smaller value should be overridden by value ≥*ignore_bigger*, the original value is left. If all outputs are needed, suply the *ignore_bigger* value with something bigger than anything in the manipulator.

```
⟨ Competitive multi out pat manip: competitive pattern output 60 ⟩ ≡
public:
  void competitive_pattern_output(const vector⟨Tin_alph⟩ &w, Outputs_of_a_pattern⟨Tindex,
          Tout_alph⟩ &o, const Tout_alph &ignore_bigger)
  {
    o.clear( );
    Tindex s = 0;      /* shift for the whole word */
    for (vector⟨Tin_alph⟩::const_iterator i = w.begin( ); i ≠ w.end( ); i++) {
        /* for all postfixes */
      vector⟨Tin_alph⟩ v(i, w.end( ));      /* take the postfix */
      competitive_word_output(v, o, s, ignore_bigger);
      s++;
    }
  }
```

This code is used in section 57.

61. Simple translation service. This service is meant usually for testing purposes.

```
⟨ ptl_sts.h 61 ⟩ ≡
#ifndef PTL_STS_H
#define PTL_STS_H
#include <map>
#include <set>
  ⟨ Simple translation service (head) 62 ⟩
#endif
```

62. Simple translation service. We map the application used values into unsigned numbers for the purpose of manipulating patterns. Therefore we provide the translation service. The service is initialized with a set of external symbols and it builds a bijection from the set into the numbers $1, \ldots, n$, where n is the number of symbols used.

The bijection is internally stored as array of external type values and as **map** for the inverse. The mapping into external value is computed in constant time, the inversion is computed in at most logarithmic time to the number of used symbols. Note that this time may be optimized if we know the external type.

The **Texternal** is type of external values, this type must be able to be **map**ped.

format *Texternal* *int*

```
⟨ Simple translation service (head) 62 ⟩ ≡
  template⟨class Texternal⟩
  class Simple_translation_service {
  public:
    typedef unsigned Tinternal;
  protected:
    Texternal *to_external;
    map⟨Texternal, Tinternal⟩ to_internal;
    Tinternal last_used_internal;
    ⟨ Simple translation service: constructor and destructor 63 ⟩
    ⟨ Simple translation service: gets and sets 64 ⟩
    ⟨ Simple translation service: internal 65 ⟩
    ⟨ Simple translation service: external 66 ⟩
  };
```

This code is used in section 61.

63. The constructor of **Simple_translation_service** builds up the translation and the inverse translation tables. The parameter is a **set** of external values.

The destructor cleans up used storage.

```
⟨ Simple translation service: constructor and destructor 63 ⟩ ≡
public:
  Simple_translation_service(const set⟨Texternal⟩ &ext_set)
  {
    to_internal.clear();
    last_used_internal = 0;
    to_external = new Texternal[ext_set.size() + 1];
    for (set⟨Texternal⟩::const_iterator i = ext_set.begin(); i ≠ ext_set.end(); i++) {
      last_used_internal++;
      to_external[last_used_internal] = *i;
      to_internal.insert(make_pair(*i, last_used_internal));
    }
  }
  virtual ~Simple_translation_service()
  {
    delete[] to_external;
  }
```

This code is used in section 62.

64. The user may want to know the last used internal code.

```
⟨ Simple translation service: gets and sets 64 ⟩ ≡
public:
  inline virtual Tinternal get_last_used_internal(void) const
  {
    return last_used_internal;
  }
```

This code is used in section 62.

65. The *internal* function returns internal code of external value. Note that no range checks are done for the reason of speed. This function is nevertheless logarithmic to the number of internal values used so we do not want to slow it down more.

The second function is overloaded for handling vectors.

```
⟨ Simple translation service: internal 65 ⟩ ≡
public:
  inline virtual Tinternal internal(const Texternal &e) const
  {
    map⟨Texternal, Tinternal⟩ :: const_iterator it = to_internal.find(e);
    return (∗it).second;
  }
  inline virtual vector⟨Tinternal⟩ internal(const vector⟨Texternal⟩ &ve) const
  {
    vector⟨Tinternal⟩ vi;
    for (vector⟨Texternal⟩ :: const_iterator i = ve.begin( ); i ≠ ve.end( ); i++) {
      vi.push_back(internal(∗i));
    }
    return vi;
  }
```

This code is used in section 62.

66. This function returns external values for internal ones. It may be used for single values or for vectors. No range checks are performed for efficiency reasons.

```
⟨ Simple translation service: external 66 ⟩ ≡
public:
  inline virtual Texternal external(const Tinternal &i) const
  {
    return to_external[i];
  }
  inline virtual vector⟨Texternal⟩ external(const vector⟨Tinternal⟩ &vi) const
  {
    vector⟨Texternal⟩ ve;
    for (vector⟨Tinternal⟩ :: const_iterator i = vi.begin( ); i ≠ vi.end( ); i++) {
      ve.push_back(external(∗i));
    }
    return ve;
  }
```

This code is used in section 62.

67. The generator companion. The generator suite fixes the pattern generating strategy, nevertheless most of the code may be reused in many cases. Terminological note: we speak about "hyphenation", "hyphenating points" and so on. We are convinced this makes the code easier to read, although the better terms would speak about "points of interest" we want to find in our data. This would be quite hard to type and hard to say, we think. Moreover it sounds terribly.

68. We start with the Hyphenable word, **Hword** in short. We think about it as a fill-in form containing the input word in the beginning together with its hyphenation information (information on correct points of interest, if gentle reader wants to) which is later filled with the results of pattern application on the word. The odd hyphenation values allow hyphenation, the even values disallow.

69. The patterns are generated in series of passes through the input file. In each pass, we deal with pattern candidates of certain length and certain hyphen position. We collect statistics for that type of patterns taking into account the effect of patterns collected in previous passes. At the end of a pass the candidates are checked and new patterns are added to the pattern set. The *left_hyphen_min* and *right_hyphen_min* values say the left and right border of a word where hyphenating is ignored.

Pattern candidates are selected one level at a time, in order of increasing hyphenating value. Pattern candidates at each level are chosen in order of increasing pattern length. Candidates at each level applying to different intersymbol positions are chosen in separate passes through the input data file. The selection of patterns out of candidates is controlled by the *good_wt*, *bad_wt*, and *thresh* parameters. A pattern is selected if $good * good\_wt - bad * bad\_wt \geq thresh$, where *good* and *bad* are the numbers of times the pattern is/is not good at the particular point. For inhibiting levels, *good* is the number of errors inhibited, and *bad* is the number of previously found (good) hyphens inhibited.

70. The generator consists of three parts: the Pass, the Level and the Generator. The Pass is the basic unit of the pattern generating process. It reads the list of input words and chooses the candidate patterns with certain length and dot position. At the end of a pass the candidates are collected, it means the good ones are inserted into the pattern structure, the bad ones are inserted too, but with special "hyphenation information", to make them affect the following passes of that level and to be able to delete them later.

The Level generates passes for the current level. It reads the pattern length range and for all the pattern lengths and all dot positions there creates a Pass. At the end of a Level the bad patterns are deleted.

The Generator creates Levels. It reads the level range and for all the levels creates a Level. Before beginning the generating process the list of patterns may be read in. It makes step-wise generating of the levels possible, as the runs are quite time-consuming. In the end of the generating process the input word list may be hyphenated with the set of the patterns just created. It allows the user to see the work of them.

71. Now we must say something about the input and output. The generator reads input data file (Word input file), writes hyphenated word list (Word output file), reads patterns (Pattern input file), and writes patterns (Pattern output file). All the services use the Translate. The Translate reads the translate file which is used to define input transformation on input file and the alphabet. The Translate is constructed with a file name of the translate file. The files get the reference to the Translate and the file name to be constructed. The interface between the Translate and the file object is their internal problem, the parts of the generator never touch the Translate. The only exception is that Translate must provide methods *get_max_in_alph*, *get_right_hyphen_min*, and *get_left_hyphen_min*. The values obtained are needed to construct the pattern storage. The origin of that solution comes from OPATGEN, where the values depend on settings in the translate file. The input files must provide the **bool** *get*(**THword** &*w*) operation returning *true* if the **Hword** has been successfully read and *false* at the end of the file. The output files must provide the *put*(**THword** &*w*) operation to write the **Hword** into the file.

72. Hword. The **Hword** (as hyphenated/hyphenable word) is a sequence of characters together with the hyphenation information. Let us define the special information.

```
⟨ptl_hwrd.h 72⟩ ≡
#ifndef PTL_HWRD_H
#define PTL_HWRD_H
#include <iostream>
#include <vector>
#include "ptl_ga.h"
  ⟨Hword (head) 73⟩
#endif
```

73. The first thing we want to know is the type of the hyphen point. The values in **Thyf_type** mean (in order of appearance) no hyphen is here, this is an erroneous hyphen, this is a correct one and this is the correct one but found sometimes in the past. Those values are defined as a part of this object to ensure consistency.

The **Hword** itself is a collection of the sequence of characters and the hyphenation information. The hyphenation information is stored in the following public-accessible growing arrays:

- *dots* is the hyphen type
- *dotw* is the dot weight of **Twt_type**
- *hval* is the hyphenation value (simply the level number) of **Tval_type**
- *no_more* is the flag indicating the "knocked-out" position

The remaining template types are the word alphabet (**Tin_alph**) and the **Tindex** type for indexing the growing arrays (**unsigned** seems to be a good choice).

Please follow the convention that the values applying between the characters *word*[*i*] and *word*[*i* + 1] is stored in *dots*[*i*], *dotw*[*i*] and so on. The fields are indexed from zero, except the word itself, which is indexed from one. This makes storing information about the position left to the leftmost words character possible.

```
  format Hword int
  format Tindex int
  format Tin_alph int
  format Twt_type int
  format Tval_type int
  format Thyf_type int
⟨ Hword (head) 73 ⟩ ≡
  template⟨class Tindex, class Tin_alph, class Twt_type, class Tval_type⟩
  class Hword :
  public std::vector⟨Tin_alph⟩ {
  public:
    typedef enum {
      no_hyf, err_hyf, is_hyf, found_hyf
    } Thyf_type;
    Growing_array⟨Tindex, Thyf_type⟩ dots;
    Growing_array⟨Tindex, Twt_type⟩ dotw;
    Growing_array⟨Tindex, Tval_type⟩ hval;
    Growing_array⟨Tindex, char⟩ no_more;
    ⟨ Hword: constructors 74 ⟩
    ⟨ Hword: clear 76 ⟩
    ⟨ Hword: operator[] 75 ⟩
#ifdef DEBUG
    ⟨ Hword: print 77 ⟩    /* Test code! */
#endif
  };
```

This code is used in section 72.

74. In the constructor we set up the values for embedded objects. The constructor of the word is called and the default values to return when accessing previously unacessed field of the arrays are set as follows. The *dots* defaults to *no_hyf*, the *dotw* should be usually 1 and the *hval* should be zero. The positions are not knocked out, so *false*.

Nevertheless we provide the constructor with parameters, where the user may set the default values.

```
⟨ Hword: constructors 74 ⟩ ≡
  Hword( )
  : std::vector⟨Tin_alph⟩( ),
  dots(no_hyf), dotw(1), hval(0), no_more(false) { }
  Hword(const Thyf_type &s, const Twt_type &w, const Tval_type &l, const char &e)
  : std::vector⟨Tin_alph⟩( ),
  dots(s), dotw(w), hval(l), no_more(e) { }
```

This code is used in section 73.

75. Operator [ ]. We want the **Hword** to be indexed from 1.

```
⟨ Hword: operator[ ] 75 ⟩ ≡
public:
  inline Tin_alph &operator[ ](const Tindex &i)
  {
    return std::vector⟨Tin_alph⟩::operator[ ](i − 1);
  }
```

This code is used in section 73.

76. If the **Hword** is cleared, we must also clear the associated information.

```
⟨ Hword: clear 76 ⟩ ≡
public:
  void clear(void)
  {
    std::vector⟨Tin_alph⟩::clear( );      /* clear the word itself */
    dots.clear( );
    dotw.clear( );
    hval.clear( );
    no_more.clear( );
  }
```

This code is used in section 73.

77. Testing code—writing the word content.

```
⟨ Hword: print 77 ⟩ ≡
  void print(void)
  {
    cout ≪ "Hword";
    for (std::vector⟨Tin_alph⟩::iterator i = this→begin( ); i ≠ this→end( ); i++) cout ≪ "␣" ≪ *i;
    cout ≪ endl ≪ "dots";
    for (Tindex i = 0; i ≤ this→size( ); i++) cout ≪ "␣" ≪ dots[i];
    cout ≪ endl ≪ "dotw";
    for (Tindex i = 0; i ≤ this→size( ); i++) cout ≪ "␣" ≪ dotw[i];
    cout ≪ endl ≪ "hval";
    for (Tindex i = 0; i ≤ this→size( ); i++) cout ≪ "␣" ≪ hval[i];
    cout ≪ endl ≪ "no_m";
    for (Tindex i = 0; i ≤ this→size( ); i++)
      if (no_more[i]) cout ≪ "␣t";
      else cout ≪ "␣f";
    cout ≪ endl;
  }
```

This code is used in section 73.

78. The generator companion. The generating process is described above. Here we talk about the types in the templates of the generator companion. Note that not all the objects (Generator, Level, and Pass) must take all the types. But most of the types bubble down to the whole structure, therefore we consider commenting them at one place more then suitable. (Moreover we must tell cweave that all the ugly names are types.)

The **Tindex** is the type used to index the fields in the words and outputs. The **Tin_alph** is the input alphabet (preferably numerical).

The **Tval_type** is the level number, this is the type of output of the pattern.

The **Twt_type** is the weight in the word, the hyphen position (or the whole word) may have a weight which makes the position (all the positions of the word) counted as more occurrences of the hyphenation point. Non-negative number.

The **Tcount_type** is the count of the times the position works OK/wrong in the candidate taking process, non-negative number, note should be big enough.

The **THword** is the Hyphenable word. More precisely said, this is an object with the same interface as the *HWord* presented above. It should also behave similarly :-). The same note applies to the following types too.

The **TTranslate** is application dependent translate, may be also a fake object (doing nothing) in case it's not needed. The object is defined in the application, see the OPATGEN source for an example.

The **TCandidate_count_structure** is the object to store candidates related to certain hyphenation level, pattern length and dot position. In our case it stores the pairs of good/bad counts of **Tcount_type**.

The **TCompetitive_multi_out_pat_manip** is the pattern store.

The **TOutputs_of_a_pattern** is the output of the word from the previous type.

The **TWord_input_file** and **TWord_output_file** for words are application dependent interfaces to the files. They use the **TTranslate** to handle "file alphabet" to internal alphabet translations.

The **TPattern_input_file** and **TPattern_output_file** are analogous interfaces for pattern files.

The **TPass** is the type of the Pass generated by the Level.

The **TLevel** is the type of the Level generated by the Generator.

The "overtemplatization" may seem funny at first sight, but note that quite global and big changes may be done by inheriting and changing the template parameter. If another order of dot position selecting is needed, you may inherit the Level, change one of its methods and change one of the template parameters. If you want to change (for example) the pattern selecting rule, there is nothing easier then to inherit the Pass and to change the *collect_candidates* method. (Well, there is something easier: to prove the collecting is good enough for you:-).)

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*

79. All the objects of the generator come to one header.

```
⟨ ptl_gen.h 79 ⟩ ≡
#ifndef PTL_GEN_H
#define PTL_GEN_H
#include <iostream>
#include <cstdio>
#include <iomanip>
#include "ptl_exc.h"
#include "ptl_mopm.h"
#include "ptl_hwrd.h"
  ⟨ Pass (head) 80 ⟩
  ⟨ Level (head) 91 ⟩
  ⟨ Generator (head) 95 ⟩
#endif
```

80. Pass. The pass is the basic unit of pattern generating process. We go through the input data and collect candidates of one length and one hyphenating position. After that we choose good candidates and make them patterns.

```
⟨ Pass (head) 80 ⟩ ≡
  template⟨class Tindex, class Tin_alph, class Tval_type, class Twt_type, class
      Tcount_type, class THword, class TTranslate, class TCandidate_count_structure, class
      TCompetitive_multi_out_pat_manip, class TOutputs_of_a_pattern, class
      TWord_input_file⟩
  class Pass {
    ⟨ Pass: data 81 ⟩
    ⟨ Pass: constructor and destructor 82 ⟩
    ⟨ Pass: hyphenate 83 ⟩
    ⟨ Pass: change dots 84 ⟩
    ⟨ Pass: do word 85 ⟩
    ⟨ Pass: print pass statistics 87 ⟩
    ⟨ Pass: do dictionary 88 ⟩
    ⟨ Pass: collect candidates 89 ⟩
    ⟨ Pass: do all 90 ⟩
  };
```

This code is used in section 79.

81. A number (mess, if you want to) of values follows.

```
⟨ Pass: data 81 ⟩ ≡
protected:
  TTranslate &translate;    /* the translate service */
  const Tval_type hyph_level;    /* current hyphenating symbol */
  const Tval_type hopeless_hyph_val;    /* fake number for bad patterns */
  const Tindex left_hyphen_min;    /* ignore leftmost part of words */
  const Tindex right_hyphen_min;    /* the same for right */
  const Tindex pat_len;    /* length of candidates we're dealing with */
  const Tindex pat_dot;    /* current position */
  const Tcount_type good_wt, bad_wt, thresh;    /* pattern choosing weights */
  TCompetitive_multi_out_pat_manip &patterns;    /* patterns */
  Tcount_type good_count, bad_count, miss_count;    /* statistics, hyphen counts */
  TCandidate_count_structure candidates;    /* candidates we collect */
  TWord_input_file word_input;    /* the file we read */
  Tindex hyf_min, hyf_max, hyf_len;    /* see constructor for explanation */
  Tindex dot_min, dot_max, dot_len;    /* see constructor for explanation */
  typename THword::Thyf_type good_dot, bad_dot;    /* see constructor for explanation */
```

This code is used in section 80.

82. We simply set up the values. Moreover—to avoid unnecessary tests—the variables *hyf_min*, *hyf_max*, and *hyf_len* are set up such that only positions from *hyf_min* up to word length minus *hyf_max* of the word need to be checked. The words with length <*hyf_len* need not be checked at all.

The *dot_min*, *dot_max*, and *dot_len* values are analogous and they mean limits of legal dots.

The *good_dot* and *bad_dot* values depend on hyphenation level and are used in *do_word* routine.

```
⟨ Pass: constructor and destructor 82 ⟩ ≡
public:
  Pass(TTranslate &tra, const char *i_d_f_n,
  const Tval_type &l, const Tval_type &h,
  const Tindex &lhm, const Tindex &rhm,
  const Tindex &p_l, const Tindex &p_d,
  const Twt_type &g_w, const Twt_type &b_w, const Twt_type &t,
  TCompetitive_multi_out_pat_manip &pat)
  : translate(tra),
  hyph_level(l), hopeless_hyph_val(h),
  left_hyphen_min(lhm), right_hyphen_min(rhm),
  pat_len(p_l), pat_dot(p_d),
  good_wt(g_w), bad_wt(b_w), thresh(t),
  patterns(pat),
  good_count(0), bad_count(0), miss_count(0),
  candidates(patterns.get_max_in_alph( ), 0, 0),
  word_input(translate, i_d_f_n) {
    hyf_min = left_hyphen_min + 1;
    hyf_max = right_hyphen_min + 1;
    hyf_len = hyf_min + hyf_max;
    dot_min = pat_dot;
    dot_max = pat_len − pat_dot;
    if (dot_min < hyf_min) dot_min = hyf_min;
    if (dot_max < hyf_max) dot_max = hyf_max;
    dot_len = dot_min + dot_max;
    if (hyph_level % 2 ≡ 1) {
      good_dot = THword :: is_hyf;
      bad_dot = THword :: no_hyf;
    }
    else {
      good_dot = THword :: err_hyf;
      bad_dot = THword :: found_hyf;
    }
  }
```

This code is used in section 80.

83. This procedure uses current patterns to hyphenate a word. The hyphenation value applying between the characters *word*[i] and *word*[$i+1$] is stored in *hval*[i]. The bad patterns at this level are ignored.

Moreover, *no_more*[i] is set to *true* iff the position is “knocked out” by either good or bad pattern at this level. It means, the pattern with current length and hyphen position is superstring of a pattern at this level. In that case we don't have to collect count statistics for that pattern as it can't be chosen in this pass. In addition, there is no need to insert that pattern into the count repository. Note that we have to hyphenate again to get the bad patterns of this level into account.

```
⟨ Pass: hyphenate 83 ⟩ ≡
public:
  void hyphenate(THword &w)
  {
    TOutputs_of_a_pattern o;
    typename TOutputs_of_a_pattern::iterator i;
    patterns.competitive_pattern_output(w, o, hopeless_hyph_val);
    for (i = o.begin( ); i ≠ o.end( ); i++) {      /* go through the output */
      w.hval[i→first] = i→second;      /* copy it into w */
    }
    patterns.competitive_pattern_output(w, o, hopeless_hyph_val + 1);
      /* now compute the output including the bad values */
    for (i = o.begin( ); i ≠ o.end( ); i++) {      /* go through the output */
      if (i→second ≥ hyph_level ∧ i→first ≥ dot_min ∧ i→first ≤ w.size( ) − dot_max) {
        vector⟨Tin_alph⟩ subw;
        for (Tindex j = i→first + 1 − pat_dot; j ≤ i→first + pat_len − pat_dot; j++) {
          subw.push_back(w[j]);
        }
        TOutputs_of_a_pattern subwo;
        patterns.competitive_pattern_output(subw, subwo, hopeless_hyph_val + 1);
        typename TOutputs_of_a_pattern::iterator val_on_pat_dot = subwo.find(pat_dot);
          /* The value has been obtained by competitive outputting, it means there is at most one value
             on each position. */
        if (val_on_pat_dot ≠ subwo.end( ))
          if (val_on_pat_dot→second ≥ hyph_level) w.no_more[i→first] = true;
      }
    }
  }
```

This code is used in section 80.

84. The *change_dots* procedure updates the meaning of hyphen values. Initially the hyphens in the word list are represented by *is_hyf* and non-hyphen positions by *no_hyf*.

Now we change this values according to the hyphens found by current patterns. So we change *no_hyf* into *err_hyf* and *is_hyf* into *found_hyf*.

Moreover we collect statistics about the number of good, bad and missed hyphens.

```
⟨ Pass: change dots 84 ⟩ ≡
public:
  void change_dots(THword &w)
  {
    for (Tindex i = w.size() − hyf_max; i ≥ hyf_min; i−−) {
      if (w.hval[i] % 2 ≡ 1) {
        if (w.dots[i] ≡ THword :: no_hyf) w.dots[i] = THword :: err_hyf;
        else if (w.dots[i] ≡ THword :: is_hyf) w.dots[i] = THword :: found_hyf;
      }
      if (w.dots[i] ≡ THword :: found_hyf) good_count += w.dotw[i];
      else if (w.dots[i] ≡ THword :: err_hyf) bad_count += w.dotw[i];
      else if (w.dots[i] ≡ THword :: is_hyf) miss_count += w.dotw[i];
    }
  }
```

This code is used in section 80.

85. For each dot position of current word, we first check if we need to consider it. It might be knocked out or we don't care about. For example, when considering hyphenating patterns, there's no need to count hyphens already found.

If a relevant dot is found, we increment the counts in the candidate store (which inserts first if necessary).
(Why does cweave think *fpos* is a type? We must tell explicitly it is not.)

format *fpos* *dpos*

```
⟨ Pass: do word 85 ⟩ ≡
protected:
  void do_word(THword &w)
  {
    for (Tindex dpos = w.size() − dot_max; dpos ≥ dot_min; dpos −−) {
      ⟨ Pass: (do word) check this dot position and continue if do not care 86 ⟩
      Tindex spos = dpos − pat_dot;     /* compute the subword range */
      Tindex fpos = spos + pat_len;
      spos ++;
      vector⟨Tin_alph⟩ subw;     /* compute the subword */
      for (Tindex i = spos; i ≤ fpos; i++) subw.push_back(w[i]);
      if (w.dots[dpos] ≡ good_dot) {
        candidates.increment_counts(subw, w.dotw[dpos], 0);     /* good */
      }
      else {
        candidates.increment_counts(subw, 0, w.dotw[dpos]);     /* bad */
      }
    }
  }
```

This code is used in section 80.

86. If the dot position is knocked out or a “do not care”, we skip this position.

```
⟨ Pass: (do word) check this dot position and continue if do not care 86 ⟩ ≡
  if (w.no_more[dpos]) continue;
  if ((w.dots[dpos] ≠ good_dot) ∧ (w.dots[dpos] ≠ bad_dot)) continue;
```

This code is used in section 85.

87. Printing pass statistics.

```
⟨ Pass: print pass statistics 87 ⟩ ≡
public:
  void print_pass_statistics(void)
  {
    cout ≪ endl;
    cout ≪ good_count ≪ "␣good␣" ≪ bad_count ≪ "␣bad␣" ≪ miss_count ≪ "␣missed" ≪ endl;
    if (good_count + miss_count > 0) {
      cout ≪ 100.0 ∗ good_count/float(good_count + miss_count) ≪ "␣%␣";
      cout ≪ 100.0 ∗ bad_count/float(good_count + miss_count) ≪ "␣%␣";
      cout ≪ 100.0 ∗ miss_count/float(good_count + miss_count) ≪ "␣%␣" ≪ endl;
    }
  }
```

This code is used in section 80.

88. Go through the input file, process the words. At the end print the pass statistics.

```
⟨ Pass: do dictionary 88 ⟩ ≡
protected:
  void do_dictionary(void)
  {
    THword w;
    while (word_input.get(w)) {      /* process words until end of file */
      if (w.size() ≥ hyf_len) {      /* don't hyphenate short words */
        hyphenate(w);
        change_dots(w);
      }
      if (w.size() ≥ dot_len) do_word(w);      /* process if reasonable */
    }
#ifdef DEBUG      /* test code */
    TOutputs_of_a_pattern o;
    vector⟨Tin_alph⟩ word;
    patterns.init_walk_through();
    cout ≪ "Patterns␣in␣the␣pattern␣manipulator:" ≪ endl;
    while (patterns.get_next_pattern(word, o)) {
      cout ≪ "Word␣";
      for (vector⟨Tin_alph⟩::iterator i = word.begin(); i ≠ word.end(); i++) cout ≪ *i ≪ "␣";
      cout ≪ "...␣has␣";
      for (typename TOutputs_of_a_pattern::const_iterator i = o.begin(); i ≠ o.end(); i++)
        cout ≪ "(" ≪ i→first ≪ "," ≪ i→second ≪ ")␣";
      cout ≪ endl;
    }
    vector⟨Tin_alph⟩ vect;
    Tcount_type a;
    Tcount_type b;
    candidates.init_walk_through();
    cout ≪ "Candidates:" ≪ endl;
    while (candidates.get_next_pattern(vect, a, b)) {
      for (vector⟨Tin_alph⟩::iterator i = vect.begin(); i ≠ vect.end(); i++) cout ≪ *i ≪ "␣";
      cout ≪ "with␣g,b:␣" ≪ a ≪ "," ≪ b ≪ endl;
    }      /* end of test code */
#endif
    print_pass_statistics();
    cout ≪ "Count␣data␣structure␣statistics:" ≪ endl;
    candidates.print_statistics();
  }
```

This code is used in section 80.

89. At the end of a pass, we traverse the *candidates* selecting good and bad patterns and inserting them into the *patterns* structure.

The *g* and *b* values of the *w* word contain the number of times this pattern helps or hinders the cause. We here determine if the pattern should be selected, or if it is hopeless, or if we cannot decide yet. In the latter case, we return *more_to_come* value *true*. This means there may still be good patterns extending current type of patterns. The *more_to_come* value is returned to the caller. Before we finish this, we print some statistics to the user.

Also note that hopeless patterns are inserted into the *patterns* structure with the fake hyphenation value, so as they will be able to be deleted later.

```
⟨ Pass: collect candidates 89 ⟩ ≡
protected:
  bool collect_candidates(Tcount_type &level_pattern_count)
  {
    cout << "Collecting␣candidates" << endl;
    bool more_to_come = false;
    vector⟨Tin_alph⟩ w;      /* word */
    Tcount_type g, b;      /* good and bad from pattern */
    Tcount_type good_pat_count = 0;
    Tcount_type bad_pat_count = 0;      /* numbers of patterns added at the end of a pass */
    candidates.init_walk_through();
    while (candidates.get_next_pattern(w, g, b)) {
      if (good_wt * g < thresh) {      /* hopeless pattern */
        patterns.insert_pattern(w, pat_dot, hopeless_hyph_val);
        bad_pat_count++;
      }
      else {      /* now we test good_wt * g − bad_wt * b ≥ thresh but we may not deal with negative
            numbers (it took me four hours to find) */
        if (good_wt * g ≥ thresh + bad_wt * b) {      /* good pattern */
          patterns.insert_pattern(w, pat_dot, hyph_level, 1);      /* we may erase the previous outputs */
          good_pat_count++;
          good_count += g;
          bad_count += b;
        }
        else more_to_come = true;
      }
    }
    cout << good_pat_count << "␣good␣and␣" << bad_pat_count << "␣bad␣patterns␣added";
    if (more_to_come ≡ true) cout << "␣(more␣to␣come)";
    cout << endl;
    cout << "finding␣" << good_count << "␣good␣and␣" << bad_count << "␣bad␣hyphens" << endl;
    if (good_pat_count > 0) {
      cout << "efficiency␣=␣" << float(good_count)/(float(good_pat_count) +
          float(bad_count)/(float(thresh)/float(good_wt))) << endl;
    }
    cout << "Pattern␣data␣structure␣statistics:" << endl;
    patterns.print_statistics();
    level_pattern_count += good_pat_count;
    return more_to_come;
  }
```

This code is used in section 80.

90. Make the pass and return the *more_to_come* value.

```
⟨ Pass: do all 90 ⟩ ≡
public:
  bool do_all(Tcount_type &level_pattern_count)
  {
    cout ≪ endl ≪ "Generating␣a␣pass␣with␣pat_len␣=␣" ≪ pat_len ≪ ",␣pat_dot␣=␣" ≪ pat_dot ≪
        endl;
    do_dictionary( );
    return collect_candidates(level_pattern_count);
  }
```

This code is used in section 80.

91. Level. In the level, patterns with given hyphenating information on various positions are collected. We read the parameters and generate passes.

```
⟨ Level (head) 91 ⟩ ≡
  template⟨class Tindex, class Tin_alph, class Tval_type, class Twt_type, class
      Tcount_type, class THword, class TTranslate, class TCandidate_count_structure, class
      TCompetitive_multi_out_pat_manip, class TWord_input_file, class TPass⟩
  class Level {
    ⟨ Level: data 92 ⟩
    ⟨ Level: constructor and destructor 93 ⟩
    ⟨ Level: do all 94 ⟩
  };
```

This code is used in section 79.

92. We have to know the current hyphenation value *hyph_level*, the *hopeless_hyph_val* is a fake value used to determine bad patterns we have to collect but we delete them later.

Left and right hyphen minimal positions are the bounds of the hyphen positions measured from the beginning and end of a word where we ignore hyphenating.

The *patterns* is the global pattern storage. The *level_pattern_count* is used for statistics.

Variables *pat_start* and *pat_finish* are the minimal and maximal lengths of pattern candidates in this level and *good_wt*, *bad_wt*, and *thresh* are used for choosing patterns when collecting candidates.

```
⟨ Level: data 92 ⟩ ≡
protected:
  TTranslate &translate;
  const char *word_input_file_name;
  const Tval_type hyph_level;
  const Tval_type hopeless_hyph_val;
  const Tindex left_hyphen_min, right_hyphen_min;
  TCompetitive_multi_out_pat_manip &patterns;
  Tcount_type level_pattern_count;
  Tindex pat_start, pat_finish;
  Tcount_type good_wt, bad_wt, thresh;
```

This code is used in section 91.

93. We read the values of the pattern length range and the parameters for choosing candidates. Of course we initialize the default values.

```
⟨ Level: constructor and destructor 93 ⟩ ≡
public:
  Level(TTranslate &tra, const char *i_d_f_n, const Tval_type &l, const Tval_type &h, const
          Tindex &lhm, const Tindex &rhm, TCompetitive_multi_out_pat_manip &p)
  : translate(tra), word_input_file_name(i_d_f_n), hyph_level(l), hopeless_hyph_val(h), left_hyphen_min(lhm),
        right_hyphen_min(rhm), patterns(p), level_pattern_count(0) {
    do {     /* read the pattern length range */
      cout << "pat_start,␣pat_finish:␣";
      cin >> pat_start;
      cin >> pat_finish;
      if (pat_start < 1 ∨ pat_start > pat_finish) {
        cout << "Specify␣two␣integers␣satisfying␣1<=pat_start<=pat_finish␣";
        pat_start = 0;
      }
    } while (pat_start < 1);
    do {     /* read the weights */
      cout << "good␣weight,␣bad␣weight,␣threshold:␣";
      cin >> good_wt;
      cin >> bad_wt;
      cin >> thresh;
      if (good_wt < 1 ∨ bad_wt < 1 ∨ thresh < 1) {
        cout << "Specify␣three␣integers:␣good␣weight,␣bad␣weight,␣threshold>=1␣";
        good_wt = 0;
      }
    } while (good_wt < 1);
  }
```

This code is used in section 91.

94. In a given level, the patterns of given length are generated with dot positions in an “organ-pipe” fashion. For example, for *pat_len* ≡ 4 we choose patterns for different positions in the order 2, 1, 3, 0, 4. For all positions passes are generated.

The array *more_this_level* remembers which positions are permanently “knocked out”, this is, if there are no possible good patterns at that dot position, we do not have to consider longer patterns at this level containing that position.

At the end of a level the bad patterns are deleted.

```
⟨ Level: do all 94 ⟩ ≡
public:
  void do_all(void)
  {
    cout ≪ endl ≪ endl ≪ "Generating␣level␣" ≪ hyph_level ≪ endl;
    Growing_array ⟨Tindex, char⟩ more_this_level(true);
    for (Tindex pat_len = pat_start; pat_len ≤ pat_finish; pat_len++) {      /* for all pattern lengths */
      Tindex pat_dot = pat_len/2;
      Tindex dot1 = pat_dot * 2;
      do {      /* for all positions */
        pat_dot = dot1 − pat_dot;
        dot1 = pat_len * 2 − dot1 − 1;
        if (more_this_level[pat_dot]) {      /* which are not knocked out */
          TPass pass(translate, word_input_file_name, hyph_level, hopeless_hyph_val, left_hyphen_min,
              right_hyphen_min, pat_len, pat_dot, good_wt, bad_wt, thresh, patterns);
          more_this_level[pat_dot] = pass.do_all(level_pattern_count);
        }
      } while (pat_dot ≠ pat_len);
      for (Tindex k = more_this_level.size( ); k ≥ 1; k−−)
        if (more_this_level[k − 1] ≠ true) more_this_level[k] = false;
    }      /* OK, we have generated */
    Tindex old_p_c = patterns.get_pat_count( );
    patterns.delete_values(hopeless_hyph_val);      /* now delete bad patterns */
    cout ≪ old_p_c − patterns.get_pat_count( ) ≪ "␣bad␣patterns␣deleted" ≪ endl;
    patterns.delete_hanging( );
    cout ≪ "total␣of␣" ≪ level_pattern_count ≪ "␣patterns␣at␣level␣" ≪ hyph_level ≪ endl;
  }
```

This code is used in section 91.

95. Generator. This is where we start the real work. We read the values of *hyph_start* and *hyph_finish*, create the set of needed symbols, create the pattern store and generate levels.

```
⟨ Generator (head) 95 ⟩ ≡
  template⟨class Tindex, class Tin_alph, class Tval_type, class Twt_type, class
      Tcount_type, class THword, class TTranslate, class TCandidate_count_structure, class
      TCompetitive_multi_out_pat_manip, class TOutputs_of_a_pattern, class
      TWord_input_file, class TWord_output_file, class TPattern_input_file, class
      TPattern_output_file, class TPass, class TLevel⟩
  class Generator {
    ⟨ Generator: data 96 ⟩
    ⟨ Generator: constructor 97 ⟩
    ⟨ Generator: read patterns 98 ⟩
    ⟨ Generator: output patterns 99 ⟩
    ⟨ Generator: hyphenate word list 100 ⟩
    ⟨ Generator: do all 101 ⟩
  };
```

This code is used in section 79.

96. The generator needs to know the hyphenation level range, multiple output pattern store, and values *left_hyphen_min* and *right_hyphen_min* telling the bounds where hyphenation should be ignored.

```
⟨ Generator: data 96 ⟩ ≡
protected:
  TTranslate translate;
  const char *name;
  const char *word_input_file_name;
  const char *pattern_input_file_name;
  const char *pattern_output_file_name;
  TCompetitive_multi_out_pat_manip patterns;
  Tval_type hyph_start, hyph_finish;
  Tindex left_hyphen_min, right_hyphen_min;
```

This code is used in section 95.

97. In the constructor we create the translate service, pattern structure, set the values, and read the hyphenation level range.

```
⟨ Generator: constructor 97 ⟩ ≡
public:
  Generator(const char *dic, const char *pat, const char *out, const char *tra)
  : translate(tra), word_input_file_name(dic), pattern_input_file_name(pat), pattern_output_file_name(out),
        patterns(translate.get_max_in_alph()), left_hyphen_min(translate.get_left_hyphen_min()),
        right_hyphen_min(translate.get_right_hyphen_min()) {
    do {      /* read the level range */
      cout ≪ "hyph_start,␣hyph_finish:␣";
      cin ≫ hyph_start;
      cin ≫ hyph_finish;
      if ((hyph_start < 1) ∨ (hyph_finish < 1)) {
        hyph_start = 0;
        cout ≪ "Specify␣two␣integers␣satisfying␣1<=hyph_start,␣hyph_finish␣" ≪ endl;
      }
    } while (hyph_start < 1);
  }
```

This code is used in section 95.

98. Before starting a run, set of patterns may be read in.

```
⟨ Generator: read patterns 98 ⟩ ≡
  void read_patterns(void)
  {
    vector⟨Tin_alph⟩ v;
    TOutputs_of_a_pattern o;
    TPattern_input_file file(translate, pattern_input_file_name);      /* open the file */
    while (file.get(v, o)) {      /* read the file */
      if (v.size() > 0) {      /* avoid strange things */
        for (typename TOutputs_of_a_pattern::iterator i = o.begin(); i ≠ o.end(); i++) {
            /* go through the outputs */
          if (i→second ≥ hyph_start) {
            throw Patlib_error("!␣The␣patterns␣to␣be␣read␣in␣contain␣"
            "hyphenation␣value␣bigger␣than␣hyph_start.");
          }
          patterns.insert_pattern(v, i→first, i→second);
        }
      }
    }
  }
```

This code is used in section 95.

99. At the very end of work we output the patterns.

```
⟨ Generator: output patterns 99 ⟩ ≡
  void output_patterns(void)
  {
    TPattern_output_file file(translate, pattern_output_file_name);
    vector⟨Tin_alph⟩ v;
    TOutputs_of_a_pattern o;
    patterns.init_walk_through();
    while (patterns.get_next_pattern(v, o)) {
      file.put(v, o);
    }
  }
```

This code is used in section 95.

100. At the very end of a pass the word list may be hyphenated with the patterns just made. So we prepare the input and output file. The pass we use to hyphenate must have *left_hyphen_min* and *right_hyphen_min* set to correct values.

The only problem is with the level number. Think about a special kind of usage to hyphenate a word list with a set of patterns. This can be simulated setting the level range like 6, 1, causing the levels itselves to be skipped. Therefore we take normally as the level number the last level we did. In the special case where the *hyph_finish* is less than *hyph_start* we increase the value to the bigger one. (This feature has been added for Petr Sojka.)

The rest of values is clean or makes no effect to the *hyphenate* and *change_dots* process. In the end, the statistics of hyphens found, wrong, and missed is printed. The output file is named “pattmp.n”, where “n” is the last level number. If bigger than 999, sorry MS DOS users :-).

```
⟨ Generator: hyphenate word list 100 ⟩ ≡
public:
  void hyphenate_word_list(void)
  {
    string s;
    cout ≪ "hyphenate␣word␣list␣<y/n>?␣";
    cin ≫ s;
    if (¬(s ≡ "y" ∨ s ≡ "Y")) return;
    Tval_type level_value = hyph_finish;
    if (hyph_start > hyph_finish) level_value = hyph_start;
    Tval_type fake_level_value = 2 ∗ ((level_value/2) + 1);
    char file_name[100];      /∗ Is there a less stupid way to do this? ∗/
    sprintf(file_name, "pattmp.%d", level_value);
    cout ≪ "Writing␣file␣" ≪ file_name ≪ endl;
    THword w;
    TWord_output_file o_f(translate, file_name);
    TWord_input_file i_f(translate, word_input_file_name);
    TPass pass(translate, word_input_file_name, level_value, fake_level_value, left_hyphen_min,
        right_hyphen_min, 1, 1, 1, 1, 1, patterns);
    while (i_f.get(w)) {      /∗ go through the file ∗/
      if (w.size() > 2) {
        pass.hyphenate(w);
        pass.change_dots(w);
      }
      o_f.put(w);
    }
    pass.print_pass_statistics();
  }
```

This code is used in section 95.

101. The *hopeless_fake_number* is a value which is even and greater than any hyphenating value used. So we compute it and create levels. In the end the outputs are output and the word list is hyphenated.

```
⟨ Generator: do all 101 ⟩ ≡
public:
  void do_all(void)
  {
    read_patterns( );    /* read in patterns */
    cout ≪ patterns.get_pat_count( ) ≪ "␣pattern␣lines␣read␣in" ≪ endl;
    Tval_type hopeless_fake_number = 2 * ((hyph_finish/2) + 1);
    for (Tval_type l = hyph_start; l ≤ hyph_finish; l++) {
      TLevel level(translate, word_input_file_name, l, hopeless_fake_number, left_hyphen_min,
          right_hyphen_min, patterns);
      level.do_all( );
    }
    output_patterns( );
    hyphenate_word_list( );
    cout ≪ endl;
  }
```

This code is used in section 95.

PATLIB