Outils Unix pour le développement de programmes C

Frédéric Pétrot

8 novembre 1999

Table des matières

Chapitre 1

Cours « outils et méthodes de développement sous Unix »

1.1 Présentation et objectifs du cours

Ce cours présente une « méthodologie », mot bien pompeux en la circonstance, de programmation autour des outils de développement disponibles sous *Unix* et du langage **C**. La trame du cours repose sur mon expérience pratique en tant que développeur au sein d'une structure d'une trentaine de programmeurs de différents niveaux contribuant à un programme de CAO pour VLSI (circuits intégrés numériques à très grande intégration). Les sources du projet dépassent allègrement le millions de lignes de **C**, sans parler de la documentation, etc.

L'idée ici est de présenter des méthodes qui permettent d'écrire du code de façon rapide et efficace, en minimisant le temps de développement et les bugs.

Notons que l'on se place à un relativement bas niveau : le langage C et l'environnement *Unix*. On recherche donc la performance d'implantation, et on s'attache à résoudre des problèmes ardus. On recherche également le coté pratique des choses : point de théorie ici, mais plutôt la présentation de résultats pratique qui s'appuient sur des cours théoriques fait par ailleurs. Nous allons essentiellement voir des outils, leurs objectifs et la façon de les utiliser.

Le cours s'appuie sur des TPs qui permettent la mise en œuvre des différents méthodes et outils vu en cours. La théorie est absente, mais cela n'empêche pas d'avoir une approche un peu formelle sur ces problèmes pratiques : c'est ce que l'on se propose d'aborder en cours et au début des TPs.

Notre fil conducteur durant les TPs sera la réalisation d'un petit compilateur du langage C qui s'approcherais en réalité plus d'un langage de commande tel `tcsh` que du vrai langage C.

Juste un mot avant de passer au plan du cours : je ne suis pour rien dans la définition et le développement des outils présentés ici. Ils sont le fruit du travail des concepteurs d'*Unix* et de ses clones, et c'est eux qu'ils faut remercier pour leur ardeur! Je me contenterais de vous donner des « pointeurs » vers l'information qui me semble pertinente dans cet environnement. La version des outils retenue est celle du projet **Gnu**, et les sources, incluant la documentation, sont disponibles sur le serveur `http://www.gnu.org`. les options peuvent donc varier si l'on est sur un système propriétaire.

Notons par ailleurs que les outils présentés ici ne sont pas nouveaux : ils datent pour les plus anciens des années 70, et pour les plus récents du milieu des années 80.

Le plan d'ensemble du cours est le suivant:

– rappel de ce en quoi consiste la compilation d'un programme et les fichiers intervenants. Qualificatifs `extern` et `static`, portée ;
– présentation de règles de codage en C sous *Unix* ;
– présentation de constructions en générale peu connues mais très utiles du langage C et de sa bibliothèque standard. Options à connaître du compilateur C ;
– de la bonne utilisation de son éditeur : `ctags` et `vi` (ou `emacs` pour les inconditionnels), `cxref` sur les systèmes qui le possède, plus bien sur `indent` pour avoir un style homogène sur tout un projet ;

– outil d'aide à l'optimisation du code : `gprof`, `gcov` ;
– construction d'un projet : architecture générale multi-plateforme, l'outil `make`, les archives (`ar`, `ranlib`, `nm`) ;
– aide à l'écriture des compilateurs : les outils `lex` et `yacc` ;
– mise au point, *debug* : l'outil `gdb` ;
– gestion des sources, gestion des versions de projets, etc : les outils `diff`, `patch`, `rcs` et `cvs`.
– distribution de version : `shar`, `uuencode`, `tar` ;
– un peu de documentation : `nroff` et les macros de base pour les manuels en ligne.

Une fois l'ensemble de ces outils, et les méthodes qu'ils supportent, vus, on peut espérer que le développement sous *Unix* n'aura plus de secrets pour vous !

1.2 Rappel des étapes de la compilation C

Un programme *exécutable* est obtenu par l'édition de liens entre des fichiers *objets*, suffixés `.o`. Ces fichiers objets peuvent avoir été regroupés (en gros concaténés) sous forme de bibliothèques, suffixées `.a` pour des simples raisons d'ergonomie : il est plus facile de spécifier l'édition de liens avec une bibliothèque plutôt qu'avec 36 `.o`. Il est également plus facile de fournir une bibliothèque (par exemple `libc.a` plutôt que tous ses constituants : 642 fichiers objets font partie de la `libc.a` sous Linux 2.0.33).

Les fichiers objets ou assimilés sont des fichiers binaires dépendant de l'architecture de la machine. En fait, c'est du langage machine dans lequel seules les références aux fonctions appellés ou variables globales utilisées n'appartenant pas au fichier courant n'ont pas été résolues. Le rôle de l'éditeur de lien `ld` est précisément de résoudre ces références externes.

Les fichiers objets sont obtenus par assemblage du code assembleur issu de la compilation grâce à `as` : c'est la transformation des mnémoniques assembleur en binaire exécutable – aux références externes près – directement sur le matériel.

Les fichiers assembleurs sont obtenus par analyse syntaxique et sémantique du langage d'entrée : le C pour ce qui nous concerne, mais en fait n'importe quel autre langage compilé.

L'entrée du compilateur C est cependant un unique fichier : il faut avoir inclus les fichiers de déclarations (`.h`) et substitué toutes les macros pour qu'il puisse faire aisément son travail. C'est le rôle du preprocesseur `cpp`.

En résumé, notre source passe dans `cpp`, puis dans `cc`, puis dans `as` pour devenir un `.o`. Il est ensuite lié grâce à `ld` à la bibliothèque standard du C, (`/usr/lib/libc.a`), et ceci produit l'exécutable `a.out`.

Les étapes visibles par l'utilisateur sont la compilation en objet à partir du source avec directives de preprocesseur, et l'édition de liens qui prend l'ensemble des objets pour créer l'exécutable.

Ce qu'il faut absolument retenir de tous ça :

– le code exécutable des fonctions (standard ou *ad-hoc* si l'on en crée ou l'on en achète) est précompilé dans une bibliothèque *connue uniquement lors de l'édition de liens*, soit la toute dernière phase de la compilation. Si l'on appelle une fonction qui n'existe pas, (faute de frappe dans le nom, etc), on aura un message du style:

```
/tmp/cca160881.o: In function ‘main’:
/tmp/cca160881.o(.text+0x41): undefined reference to ‘exti’
```

– les fichiers d'entête ont un rôle (primordial) de déclaration et de vérification sémantique : ils permettent de s'assurer par exemple qu'une fonction est appellée avec les bons arguments. Ils ne sont par contre pour *rien* dans la définition des fonctions et des variables.

1.3 Règles de développement

Je présente ici un ensemble de règles de développement du logiciel qui participent à une intervention plus façile sur des sources existantes. Ne perdons pas de vue que dans la « vraie vie » du programmeur, personne ne commence un projet à partir de rien pour le mener au bout seul.

Le développement est donc un travail d'équipe, et les règles que je présente ici ont pour but de minimiser les discussions entre personnes pour garantir *a priori* un bon échange d'information entre les différents partie d'un programme. Ces règles sont séparés en deux grands groupes : le style de codage et la conception modulaire.

1.4 Style de codage

Les buts recherchés sont :

– un style de programmation (une syntaxe, pas la forme elle même), homogène ;
– une facilité de compréhension, sans pour autant oublier que l'on s'adresse à des gens qui sont des programeurs ;
– la portabilité sur différentes architectures ;
– la possibilité de maintenance par différents programmeurs.

1.4.1 Convention sur les noms de fichiers

Nous adopterons celles d'*Unix* :

`.c`	fichier source C avec des directives de preprocessing
`.h`	fichier d'entête
`Makefile`	base de règles pour la compilation de projets

Nous prendrons le suffixe par défaut pour les objets générés, que nous supposerons `.o`, et les bibliothèques, que nous suffixerons `.a`. Nous ne suffixerons pas l'exécutable.

1.4.2 Organisation et contenu du source C

On entent par « source C » un fichier **C** et son entête associé.

1. le fichier **C** est séparé en section que l'on placera, si possible, dans l'ordre suivant :
   (a) inclusion des fichiers d'entête. Les entêtes standards précedent les entêtes internes au projet ;
   (b) définition des variables globales externes (`extern`) ;
   (c) définitions des types, macros et types énumérés locaux au fichier : `#define`, `enum` ;
   (d) définitions des variables globales locales au fichier (`static`) ;
   (e) déclaration des fonctions locales au fichier (`static`) ;
   (f) finalement, définition des fonctions, statiques ou externes, dans un ordre logique !
2. la longueur de la ligne sera inférieure à 80 caractères, afin d'être éditable sur des terminaux simples ;
3. une fonction fait, en standard, de deux a trois pages maximun. Il est clair que c'est une règle générale, et qu'elle souffre des exceptions justifiées ;
4. à chaque fichier source est associé un fichier d'entête. le rôle de ce fichier est de *rendre visible* les fonctions externes et les variables globales externes qui y sont définies, à l'exclusion de toute autre chose ;
5. fichier d'entête :
   (a) on inclura dans le fichier **.c** le fichier `.h` qui lui est associé. Ceci garanti la cohérence entre ce qui est défini dans le fichier `.c` et ce qui est déclaré pour être utilisé par le monde extérieur, `.h` ;
   (b) le `.h` contient la définition des structures, macros et types énumérés que l'on souhaite rendre visible ;
   (c) toutes les variables et fonctions déclarées dans le fichier sont typées `extern` ;
   (d) il n'y a pas la moindre affectation de variable globale !

(e) on évitera de multiples inclusions par :

```
#ifndef _FICHIER_H_
#define _FICHIER_H_
...
#endif
```

(f) on évitera d'inclure un fichier `.h` dans un autre fichier `.h` ;

(g) occasionellement, on pourra avoir plusieur `.h` par `.c` : en effet, des types, variables, etc, doivent parfois être visibles par d'autres fichiers *localement*, pour le développement, sans pour autant devoir être visibles par l'utilisateur final. On appelle ces fichiers des entêtes privés, et suivant la norme adoptée pour le développement de X11, on les suffixera par `P.h`.

6. il faut minimiser l'usage des variables globales ;
7. les constantes (types enumérés) ou macros sont en majuscule :

```
#define TRUE 1

enum {ONE = 1, TWO, THREE, FOUR, FIVE};
```

8. ne pas oublier de parenthéser les arguments des macros :

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

9. utiliser le modèle ANSI pour les déclarations de prototypes et les définitions de fonctions ;
10. choisir un style pour le nommage des variables et s'y tenir ! Trois styles sevissent actuellement : `UneVariable`, `une_variable`, ou encore `entierUneVariable`.
    Pour ce qui me concerne, je *préconise* :
    – `UneJolieVariableLocale` pour les variables locales ;
    – `LaBelleFonction()` pour les fonctions. La présence des parenthèses lève les ambiguïtés ;
    – `unpetitparamètre` pour les paramètres de fonctions, ce qui permet aisément de différencier une variable locale d'un paramètre ;
    – `unmondechamp` pour les noms de champs des structures. La présence de `->` permet facilement de les différencier des paramètres.

Remarques :

1. l'indentation n'est pas un problème : on utilisera dans tous les cas un outil de reformatage (type `indent`), avant de distribuer ses sources ;
2. l'optimisation réelle ne dépend que très peu de la manière dont les sources sont écrites syntaziquement : on privilégiera donc la lisibilité ;
3. penser à la portabilité du code : tous les types n'ont pas les mêmes longueurs sur toutes les plateformes, voir même varient en fonction du compilateur sur une plateforme donnée ! Un 386 avec un compilateur microsoft a des `int` sur 16 bits, alors qu'avec un compilateur *Unix*, les `int` sont sur 32 bits.

1.5 Quelques constructions et fonctions C peu usités mais Ô combien utiles

1.6 Les fonctions à nombre variable d'arguments

Ce sont les fonctions de type `printf`.

Imaginons que l'on cherche à écrire une fonction qui prend un nombre quelconque de chaînes de caractères en paramètres, sachant que la dernière est nulle, et qui retourne un tableau de pointeurs sur ces chaînes.

une telle fonction se déclare ainsi :

```
char **VarArgToTable(char *s, ...)
{
}
```

Il faut inclure <stdarg.h>, qui défini un nouveau type : le type pointeur sur argument va_list. On doit donc déclarer une variable de ce type, et la faire pointer sur le *dernier* argument nommé, grâce à va_start, qui prend comme premier argument la variable de type va_list et comme second argument le dernier paramètre nommé de la fonction. On pourra ensuite examiner les différents arguments grâce à va_arg, qui prend en premier argument la va_list et en second argument le type attendu du paramètre « invisible ». On doit donc forcément connaître ce type ! On achéve la fonction par un va_end avec comme argument la variable de type va_list.

La fonction ainsi écrite se résume à :

```
#include <stdio.h>
#include <stdarg.h>
#include <error.h>

char **VarArgToTable(char *s, ...)
{
char    **Table;                         /* Pointer to the table to be returned */
va_list *Parameters, *SavedParameters; /* Pointer to parameter */
int     NumberOfParameters;            /* Number of paramters */
int     Index;                         /* Index */

   /* All args are NULL:
    * so we return an array with a single element that points to
    * NULL. */

   if (!s) {
      Table = (char **)malloc(sizeof(char *));
      if (!Table)
         Error(ERROR, NULL_MALLOC, "%s: line %d\n", __FUNCTION__, __LINE__);
      *Table = NULL;
      return Table;
   }

   SavedParameters = Parameters = va_start(s);

   /* Counts the number of arguments */
   for(NumberOfParameters = 0;
       va_arg(Parameters, char *) != NULL;
       NumberOfParameters++);

   /* Allocate the table */
   Table = (char **)malloc((NumberOfParameters + 1) * sizeof(char *));
   if (!Table)
      Error(ERROR, NULL_MALLOC, "%s: line %d\n", __FUNCTION__, __LINE__);

   /* Fill the table */
   Table[0] = s;
   for(Index = 1; Index < NumberOfParameters; Index++)
      Table[Index] = va_arg(SavedParameters, char *);

   return Table;
}
```

Il est aussi utile de connaître les fonctions de la famille vprintf qui sont identiques à printf, mais qui prennent une va_list au lieu des arguments standards. Elles sont très utiles pour l'écriture des fonctions de gestion des erreurs centralisées.

Imaginons la fonction Error qui prend comme premier argument un niveau (ERROR ou WARNING), un code d'erreur défini pour l'application, puis des arguments à la printf.

Dans error.h :

```
enum {ERROR, WARNING, INFORMATION};
enum { , ..., ..., ..., NULL_MALLOC, ...};
```

Dans error.c :

```
#include <sdtio.h>
#include <stdlib.h>
#include "error.h"

static char *ErrorString[] = {"Error", "Warning", "Info"};

void Error(int level, int type, char *fmt, ...)
{
va_list Parameters;


   (void)fprintf(stderr, "%s: ", ErrorString[level]);
   /* Print out the rest of the user message */
   va_start(Parameters, fmt);
   (void)vfprintf(stderr, fmt, Parameters);
   va_end(Parameters);

   /* And some meaningfull informations, if needed */
   switch (type) {
      case ... :
         ...
         break;

      case NULL_MALLOC:
         (void)fprintf(stderr, "No enough virtual memory for operation\n");
         break;
      ...
   }

   if (level == ERROR)
      (void)abort;
}
```

1.7 Goto or not goto ?

L'usage des gotos est discrédité dans la plupart des ouvrages de programmation, et en effet, c'est une source d'ennuis infinie si l'on s'en sert abusivement. Il est un usage des gotos que j'utilise : pour sortir de boucles imbriquées lorsqu'on detecte une condition d'arrêt au n^{ime} niveau d'imbrication. C'est un break ou continue généralisé, en quelque sorte.

L'exemple ci-dessous illustre la recherche d'une valeur dans une matrice et retourne les indices de ligne et colonne *dès* qu'ils ont été déterminés :

```
int *matrix;
int lines, columns;
int value;
int p, q;

...

   for (p = 0; p < lines; p++)
      for (q = 0; q < columns; q++)
         if (*(matrix + p * columns + q) == value)
            goto end; /* Break would simply break out of the colums loop ! */
end:
   ...
```

La relative lisibilité du code tient au fait que l'étiquette se trouve tout près du `goto`.

1.8 Options du compilateur C

Les options les plus utiles du compilateur C sont :

`-c`	n'exécute pas la phase d'édition de liens.
`-o` *arg*	renomme le fichier de sortie en *arg* au lieu du par défaut `a.out`.
`-g`	insère des informations nécessaire aux outils de mise au point au niveau source.
`-p`	insère des compteurs permettant d'évaluer le temps passé à exécuter chaque fonction.
`-Wall`	demande au compilateur de faire le plus de vérifications statiques possibles.
`-Werror`	considère un avertissement (*warning*) comme une erreur.
`-O`*n*	effectue les optimisations de niveau *n*, ou *n* est compris entre 1 et 4 généralement.
`-I`*path*	recherche les fichiers d'entête dans le reperoire pointé par *path* avant de les rechercher dans les répertoires standards `/usr/include`, `/usr/local/include`.
`-L`*path*	recherche les bibliothèques dans le reperoire pointé par *path* avant de les rechercher dans les répertoires standards `/usr/lib`, `/usr/local/lib`.
`-D`*arg*`=`*val*	identique à l'écriture d'une directive `#define arg val` dans le code.

1.9 Utilitaires variés pour l'aide au développement

En dehors des incontournables `find`, `grep` et autres `diff`, *Unix* propose des outils spécifiquement dédié au développement.

1.10 Les éditeurs

Je n'en parlerais que pour parler des deux « mondes » qui s'opposent : les amateurs de `vi`, dont je fais partie, et ceux d'`emacs`, dont je ne fais pas partie. Notons que je n'ai pas d'*a priori* concernant les éditeurs, si ce n'est qu'ils doivent permettre de développer du code rapidement. Et je ne connais que `vi` et `emacs` qui rendent ce service ! Choisisez un éditeur, et apprennez et maîtrisez ses fonctionnalités.

Pour les amateurs de `vi`, je conseille le refonte du domaine publique `vim`, qui inclue notamment la coloration des mots clefs.

1.11 ctags

Lorsque l'on travaille sur un grand nombre de fichiers sources, éventuellement dans des répertoires différent, il est pénible de se rappeller l'endroit ou telle où telle fonction est définie, telle où telle variable déclarée, etc.

Le but de `ctags` est précisément de construire une base de données contenant les références aux différents constituant du source C. Une fois la base de données construite, on peut, sous son éditeur favori ('a condition qu'il s'appelle `vi` où `emacs`), aller directement en quelques touches dans le fichier ou la fonction ou la variable est déclarée, empiler de tels appels, puis revenir à son point de départ. C'est extrêmement utile pour consulter les paramètres d'une fonction que l'on vient d'écrire et qui n'est pas encore documentée.

Supposons que l'on ne travaille que dans un repertoire `sce` :

```
sce %-) ls
Makefile   hash.c     hash.h     hash.o     main.c     main.o
```

Pour créer le fichier de *tags*, on fera tout simplement :

```
sce %-) ctags *.[ch]
sce %-) ls
Makefile   hash.c     hash.h     hash.o     main.c     main.o     tags
```

On obtient donc un fichier `tags` qui contient des lignes à l'allure suivante :

```
additem hash.c  /^static entry *additem(entry *head, $/;"  f
main    main.c  /^int main()$/;"  f
```

En fait, on se moque du format, l'important est que notre éditeur préféré s'en accomode sans nous en informer !

Il suffira alors (sous `vi` que je connais) de faire :

```
:se tags=./tags "Use the current directory tags file
```

Puis de se placer quelque part sur le nom de fonction, de variable, de type, ..., et de faire]̂. Ceci positionne sur la définition de l'objet, et l'on peut se déplacer et réitérer l'opération. Ceci « empile » les appels, et l'on remonte de la pile grâce à T̂. On peut aller à un *tag* donné au démarage grâce à :

```
sce %-) vi -t main
```

Ou encore y aller en cours d'édition grâce à :

```
:ta main
```

`Ctags` est aussi utilisable sur des sources dans de multiples répertoires, ou il prend alors un intérêt tout particulier : inutile de spécifier des déplacements dans l'arborescence pour aller chercher un fichier, il suffit d'utiliser les *tags*.

Il faut alors centraliser la base de données, et *ajouter* les *tags* au fichier, histoire de ne pas l'écraser bêtement à chaque nouveau sous-répertoire. On procède alors ainsi :

```
sce %-) find -name \*.\[ch\] -exec ctags -a -f $HOME/projet/sce/tags {} \;
```

N'hésitez pas à faire `man ctags`, notamment pour les amateurs d'`emacs`.

1.12 cxref

Cet outil n'est malheureusement pas disponible sur toutes les architectures, mais il permet de construire le graphe des appels entres les différentes fonctions d'un programme. Vous pouvez toujours voir s'il existe sur votre système, ...

1.13 indent

Une des guerres de religions les plus brutale concerne l'indentation. Elle est à peu près aussi intense qu'entre les amateurs d'`emacs` ou de `vi`, ...

`Indent` permet de mettre tout le monde d'accord en permettant une réindentation du source suivant tout un tas de paramètres. Il faut consulter la documentation pour les détails, mais deux styles d'indentation sont largement répandus :

– le style Kernighan & Ritchie, qui reprend le format du célèbre ouvrage « Le langage C », s'obtient avec `indent -kr f.c` :

```
if (a == 3) {
    if (b == 4) {
        c = 5;
    } else {
        d = 6;
    }
}
```

– le style imposé dans les projets **Gnu** s'obtient avec `indent -gnu f.c` :

```
if (a == 3)
  {
    if (b == 4)
      {
```

```
            c = 5;
         }
      else
         {
            d = 6;
         }
   }
```

J'utilise personellement un style proche du Kernighan & Ritchie, mais sans tabulation et avec une indentation de 3 :

```
indent -kr -ts 80 -i 3 a.c
```

On voit qu'il est possible de choisir un style par défaut, puis de le modifier pour qu'il soit exactement comme on le désire en modifiant quelques paramètres.

Maintenant, le mieux est de consulter `man indent` pour choisir les options qui vous plaisent.

1.14 Gprof & gcov

Un des objectifs de tout bon développeur est de fournir des programmes performant. Des complexités théoriques imposent des limites qui sont inévitables, mais des implantations soignés peuvent permettre de faire descendre les constantes intervenant dans les expressions de complexité de manière drastique. On a déja vu des facteurs 5 ou plus entre une bonne et une mauvaise implémentation d'un algorithme linéaire !

Aussi est-il utile de traquer les fonctions les plus gourmandes. C'est ce que `gprof` permet de faire en fournissant une approximation du temps passé dans chaque fonction, le nombre de fois qu'une fonction est appellée, etc. Ceci permet souvent de comprendre et de corriger si possible une inefficacité d'implementation.

L'utilisation de `gprof` est fort simple :

1. il faut compiler l'ensemble des sources et faire l'édition de liens avec l'option `-p` ;
2. on exécute ensuite son programme comme si de rien n'était (les performances sont à peine dégradées). Si l'exécution se passe bien, c.-à-.d sortie normale avec `exit` ou `return` de `main`, alors on se retrouve avec un fichier de statistiques `gmon.out` dans le répertoire courant ;
3. on lance ensuite `gprof` avec comme premier paramètre le nom de l'exécutable (par défaut `a.out`), et comme second le fichier de statistique. Le résultat est donnée sur la sortie standard, que l'on peut rediriger vers un fichier pour une étude approfondie.

   ```
   sce %-) gprof ht > perf
   sce %-) vim perf
   ```

Le fichier donne deux résultats complémentaires permettant l'analyse fine des performances.

Le premier indique, par ordre de pourcentage de temps décroissant, le temps nécessaire à l'exécution d'une fonction.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 31.61      0.49     0.49                             internal_mcount
 10.97      0.66     0.17    25143     6.76    12.27  hash_install
  8.39      0.79     0.13                             .rem
  6.45      0.89     0.10                             _sbrk_unlocked
  5.81      0.98     0.09    25144     3.58     5.23  _doprnt
  5.81      1.07     0.09    25144     3.58     4.36  hash_lookup
  2.58      1.11     0.04   207914     0.19     0.19  _libc_threads_interface
  2.58      1.15     0.04   104987     0.38     0.38  _malloc_unlocked
  2.58      1.19     0.04   103356     0.39     0.39  _smalloc
  2.58      1.23     0.04    75429     0.53     0.53  strlen
```

Les colonnes donnent, dans l'ordre :

1. le pourcentage du temps passé dans la fonction elle même, sans tenir compte des fonctions qu'elle appelle. La somme de cette colonne est donc, approximativement, 100% ;
2. la somme des temps des fonctions listée au dessus incluant la fonction elle même. Cette entrée n'a pas grand sens ;
3. le nombre de secondes passées dans la fonction *sans* prendre en compte le temps des fonctions qu'elle appelle ;
4. le nombre de fois ou la fonction est appellée ;
5. le nombre de micro-secondes par appel passées dans la fonction *sans* prendre en compte le temps des fonctions qu'elle appelle ;
6. le nombre de micro-secondes par appel passées dans la fonction en prenant en compte le temps des fonctions qu'elle appelle ;
7. et finalement le nom de la fonction.

Le listing est un peu pollué par les fonctions des libraries standard qui commencent par des *underscore*s. On peut éviter cet affichage grâce à l'option `-e` *`function`*, mais c'est assez longuet à spécifier !

Le second représente le graphe des appels de fonction, (*call graph* en anglais) et a l'allure suivante :

```
index % time    self  children    called     name
-----------------------------------------------
                               11486             InstallKey [8]
                0.08    0.05   11681/11681       main [3]
[8]      9.2    0.08    0.05   11681+11486   InstallKey [8]
                0.00    0.02   23165/46537       malloc [13]
                0.01    0.00   42682/258540      strcmp [10]
                0.00    0.01   11493/34853       free [14]
                0.00    0.00       7/11690       calloc [19]
                0.00    0.00       7/8           tablesize [48]
```

Nous sommes en train de regarder la partie du graphe d'appel des fonctions qui concerne la fonction `InstallKey`. Les fonctions qui l'appelle sont listées au dessus : `main` dans ce cas. Les fonctions qui sont appellées par elle sont listées en dessous. Les colonnes ont la signification suivante pour la fonction concernée : la ligne noté `[8]` dans notre cas.

1. index de la fonction dans le graphe ;
2. pourcentage du temps d'exécution ;
3. temps passé dans la fonction et ses descendantes ;
4. temps propagé par les fonctions appellées de celle ci ;
5. nombre de fois ou la fonction a été appellée. Le premier nombre n'inclus pas les appels récursifs, qui sont notés après un `+` ;

Pour les parents de la fonction, la signification est la suivante :

1. temps propagé par la fonction dans son appelant ;
2. temps propagé par les descendantes de la fonction dans son appelant ;
3. nombre de fois ou cet appelant a appellé la fonction sur nombre de fois ou la fonction a été appellée. Dans notre cas, la fonction `InstallKey` n'a été appellée que par `main`.

Pour les descendants de la fonction, ces informations ont la signification symétrique.

Un autre type d'outil fort intéressant permet de connaître le nombre de fois que chaque bloc de base du code a été appellé. Il s'agit des outils effectuant du « test de couverture ». Je rappel qu'un bloc de base, ou encore *basic block* en anglais, est une suite d'instructions séquentielles sans branchement.

Cette comptabilité n'est pas du même « domaine » que `gprof`, car le temps de traitement effectif n'intervient pas. Elle est en quelque sorte la representante expérimentale de la compléxité d'un algorithme.

`Gcov` permet de savoir précisément quelles lignes de codes sont effectivement exécutées, combien de fois une ligne de code donnée est exécutée, et en conjonction avec `gprof` combien de temps est nécessaire au déroulement d'une partie du code. Une fois ces informations connues, il est possible de regarder si l'on peut optimiser les parties les plus couteuses.

L'utilisation de `gcov` est également aisée, le malheur étant qu'elle n'est pas souvent documentée (sur les systèmes auquel j'ai accès). Aussi est-elle résumée ici.

Tout d'abord, et même si cela parait bizzare pour un outil d'aide à l'optimisation, le code devrait être compilé sans optimisation. Il est également souhaitable de placer peu d'expressions sur la même ligne, car l'information fournie par `gcov` est donnée ligne par ligne. Les macros complexes cacherons forcément de l'information, aussi est-il préférable d'utiliser des fonctions (et éventuellement de les qualifier d'*inline*) pour pouvoir acquérir un maximun d'informations. Les adeptes du « one liner » n'ont qu'à se retenir pour cette fois !

`gcov` génère un fichier `source.c.gcov` à partir d'un fichier `source.c` dans lequel chaque ligne est précédée du nombre de fois ou elle a été exécutée. Avec l'aide de `gprof`, on peut déterminer le temps absolu pris par un bout de code spécifique qui n'est pas une fonction.

Attention, `gcov` ne fonctionne qu'avec les programmes compilés avec `gcc`.

On appelle `gcov` ainsi :

```
gcov [-b] [-l] [-f] [-o répertoire] source
```

- `-b` demande d'écrire les fréquences de prise de branchement, et émet un résumé sur la sortie standard ;
- `-l` demande la création de nom à ralonge pour les fichiers d'entête, si du code exécutable s'y trouve (ce que je réprouve !) ;
- `-f` émet des statistique par fonction et non simplement pour le fichier ;
- `-o répertoire` indique le répertoire dans lequel se trouve les fichiers résultant de l'exécution du programme.

Pour pouvoir utiliser `gcov`, les sources du programme doivent être compilées avec les options `-fprofile-arcs -ftest-coverage`. Ces options sont nécessaires afin de demander au compilateur d'ajouter de l'information pour `gcov`, typiquement du code servant à générer les fichiers pour `gcov` lors de l'exécution.

En effet, l'exécution génère des informations de « profile ». Pour chaque fichier compilé avec `-fprofile-arcs`, un fichier d'extension `.da` est émis dans le répertoire des sources.

Il faut ensuite lancer `gcov` à proprement parler, en lui donnant comme argument la liste des fichiers sources ayant été compilé avec `-fprofile-arcs`.

`gcov` reécrit le programme source en préfixant chaque ligne par le nombre de fois ou elle a été exécutée. Les résultats sont clairement dépendant des données, ce qui signifie que des exécutions avec d'autres données donneront des résultats différents.

Pour un branchement, `gcov` écrit une pourcentatge représentant le nombre de fois ou la branche a été prise divisé par le nombre de fois ou le test a été exécuté. Si la branche n'a jamais été exécutée, le message *"never executed"* est émis.

Pour un appel de fonction, un pourcentage indiquant le nombre de fois que l'on est revenue de l'appel sur le nombre d'appel est affiché. C'est souvent 100%, sauf si l'on utilise des `exit` ou des `longjmp`.

Les traces d'exécutions (contenues dans les fichiers `.da`) s'accumullent : ceci permet de mener des campagnes de statistiques sur un grand nombre de données, afin d'obtenir des informations plus fiables. Il suffit d'effacer le fichier `.da` pour remettre les compteurs à zéro. De même, si `gcov` se rend compte que le `.da` est obsolète par rapport au programme en cours d'exécution, il l'écrase sans autre forme de procès.

Un petit exemple d'utilisation de `gcov` (un peu trivial, je m'en excuse). Il permet de vérifier que `random` retourne équitablement des nombres pairs et des nombres impaires.

Le source :

```
#include <stdlib.h>
int f(i)

   return random()%i;


main()

int i;
```

```
   for (i = 1; i < getpid(); i++)  /* Pas très fin, ... */
      /* vérifions que random n'est pas biaisé */
      if (f(i)%2)
         printf("odd");
      else
         printf("even");

   exit(0);
```

Les commandes :

```
$ gcc -fprofile-arcs -ftest-coverage a.c
$ ./a.out
$ gcov  -b -f a.c
 87.50% of 8 source lines executed in function main
100.00% of 5 branches executed in function main
100.00% of 5 branches taken at least once in function main
 83.33% of 6 calls executed in function main
100.00% of 2 source lines executed in function f
100.00% of 1 branches executed in function f
100.00% of 1 branches taken at least once in function f
100.00% of 2 calls executed in function f
No executable source lines in function _GLOBAL_.I.mainGCOV
No branches in function _GLOBAL_.I.mainGCOV
100.00% of 1 calls executed in function _GLOBAL_.I.mainGCOV
 90.00% of 10 source lines executed in file a.c
100.00% of 6 branches executed in file a.c
100.00% of 6 branches taken at least once in file a.c
 88.89% of 9 calls executed in file a.c
Creating a.c.gcov.
```

Le résultat dans `a.c.gcov`.

```
                #include <stdlib.h>
                main()

           1    int i;

        8136       for (i = 1; i < getpid(); i++)
call 0 returns = 100%
branch 1 taken = 100%
branch 2 taken = 100%
branch 3 taken = 100% le test de boucle est toujours pris -- sauf 1 fois
        8135          if (f(i)%2)
call 0 returns = 100%             on revient toujours de f(i)
branch 1 taken = 50%              et on prend cette partie de la branche 1 fois sur 2
        4033             printf("odd");
call 0 returns = 100%
branch 1 taken = 100%             on va a la fin a tous les coups
                      else
        4102             printf("even");
call 0 returns = 100%             on va a la fin a tous les coups
        8135
           1       exit(0);
call 0 returns = 0%               on ne sort jamais de la fonction~!
      ######
call 0 never executed

                int f(i)

        8135            return random()%i;
call 0 returns = 100%
branch 1 taken = 100%
        8135
```

```
call 0 returns = 0%
call 1 returns = 100%
```

1.15 Architecture logicielle et organisation des sources

1.15.1 Quelques mots sur l'architecture logicielle

Un point primordial dans tout projet de programmation est le choix d'une architecture logicielle adaptée.

Cette architecture dépend fondamentalement du projet, et requiert une connaissance globale de ce qui doit être fait : c'est le chef de projet qui va « partitionner » le projet en sous-projets aussi indépendants que possible. L'interdépendance entre les sous-projets doit être clairement définie, et les interfaces logicielles précisées.

Ces interfaces peuvent être de trois types :

1. des fichiers, textuels ou binaires, contenant l'information qui est générée par une partie et lue par une autre. C'est une façon de procéder assez classique, et qui, si les formats sont standard, procure une grande souplesse. On entent par fichier également les `pipes` ou les fifos nommées : ce qui importe ici, c'est que l'on défini une syntaxe et une sémantique d'échange.

   Son coût est la performance :
   - il faut écrire sur disque puis relire, ce qui nécessite des ressources importantes et à faible bande passante ;
   - si la syntaxe est compliquée, il faut utiliser des algorithmes *ad-hoc*, donc lents (même s'ils sont de complexité linéaire. Exemple: jpeg).
2. des structures de données. En passant par l'intermédiaire de structures de données, on permet un échange d'information aussi rapide que possible. De plus, si la structure de données est « protégée » par des fonctions d'accès, on a une certaine protection contre les erreurs de sémantique.

   Cependant, cette solution a également un coût :
   - forte dépendance sur la structure de données : si le vendeur ou l'équipe qui la gère décide de changements visibles, il est assez dur de s'adapter, et nécessite souvent de rester sur une ancienne version du produit ;
   - le programmeur averti peut utiliser des champs pour un usage inavouable, les detournant de leur usage primitif.
3. des fonctions. C'est la méthode *a priori* la plus efficace, dans le sens ou une partie du programme peut acceder directement aux fonctionnalités d'une autre partie.

   Néanmoins cette méthode fait reposer du développement sur un autre développement en cours, (si l'on s'attaque à un projet vierge), ce qui est extrêment dangeureux. En effet, des innéficacités dans le design initial apparaîssent toujours, ce qui nécessite de modifier des interfaces. Hors les appels de fonctions sont toujours distribués aléatoirement dans le code, ce qui rend les modifications fastidieuses et sujettes à erreurs !

Dans la réalité, les trois méthodes sont quasiment toujours utilisées concurramment dans un même projet. Ce qui décide d'en utiliser une plutôt qu'une autre, c'est la « distance » qui sépare une partie du projet d'un autre. De plus, il est toujours possible de changer *a posteriori* ce qui constituait la lecture d'un fichier en une traduction de structure de données, alors que l'inverse est nettement plus complexe.

Ce qui permet de faire un « bon » partitionnement, c'est d'une part la connaissance pointue du problème, et d'autre part l'expérience, qui ne vient qu'avec la pratique.

On pourra également se référer a des méthodes formelles de spécifications, l'orientation objet, etc, qui donnent de précieuses indications.

1.15.2 Organisation des sources

L'organisation des sources est le reflet des choix d'architecture. On ne peut donc pas en parler en détail sans prendre un exemple particulier, ce que nous ferons en TD.

Mais il y a quelques généralités que l'on peut avancer :

– un produit doit être compilé sur plusieurs plateformes (matérielles, ou plusieurs systèmes d'exploitation sur un même matériel) ;
– les sources doivent être uniques !
– un produit doit être modulaire, pour répondre à des offres commerciales variées ;

Afin de supporter une compilation sur plusieurs architectures, on doit d'abord déterminer le type du système :

```
uname -s -r | sed -e "s/ /-/g"
```

Sur un PC sous Linux, le résultat est typiquement `Linux-2.0.34`. Sur un Sun, le résultat sera par exemple : `SunOS-5.1.1`

Ensuite, on peut organiser les répertoires de travail comme suit, en suposant que nous avons les 2 plateformes précédentes à disposition :

```
projet -+
        |
        +-- benchmarks/
        |
        +-- Linux-2.0.34/----+-- bin/
        |                    |
        |                    +-- include/
        |                    |
        |                    +-- lib/
        |                    |
        |                    +-- obj/
        |
        |
        +-- SunOS-5.1.1/-----+-- bin/
        |                    |
        |                    +-- include/
        |                    |
        |                    +-- lib/
        |                    |
        |                    +-- obj/
        |
        +-- etc/
        |
        +-- bin/
        |
        +-- include/
        |
        +-- lib/
        |
        +-- sce/
```

Le répertoire `sce` contient les sources du programme en développement. les répertoires `bin`, `lib` et `obj` contiennent respectivement les binaires, les bibliothèques et les fichiers objets pour l'architecture sur laquelle se fait le développement. Les répertoires `obj`, qui se trouvent sous une architecture spécifique, contiennent des liens symboliques sur les sources. L'intérêt d'une telle approche est que :

1. toutes modifications effectuées dans `sce` sont directement répercutées dans `obj`, et *vice-versa*. C'est particulièrement utile si l'on a un programme qui ne marche pas sur une architecture spécifique, et que l'on a donc envie de mettre au point sous `obj` ;
2. le gain de place est considérable comparé à une copie des lors que le projet est assez gros.

Il est clair que l'ajout de fichiers nécessite la mise à jour de ces sous-répertoires. Elle peut être automatisé grâce à différentes approches, comme par exemple les *Makefiles*. Mais c'est une autre histoire, ...

Chapitre 2

Make: un programme pour gérer la cohérence de la production de fichiers

2.1 Introduction

À l'aide d'un fichier de description et d'un ensemble de règles, l'utilitaire `make` crée une suite de commandes qui seront exécutées par le *shell* d'Unix. Habituellement, ces commandes agissent sur des fichiers qui composent un projet de développement logiciel. Mais il peut en être autrement et ce programme peut être utilisé dans des contextes très divers, comme la production et la validation de circuits *VLSI*, ou la gestion électronique de documents.

Un aspect complexe de la production automatique des programmes est de garantir que chaque fichier produit est bien issu du ou des fichiers sources les plus récents. Une approche naïve consiste à regénérer la totalité des objets produits à chaque modification de l'un des fichiers source. C'est inacceptable, car cela entraîne des temps de régénération qui croissent très rapidement avec le nombre de fichiers, surtout lorsque des outils complexes rentrent en jeux pour leur production.

Au départ, `make` a été écrit pour gérer la compilation de programmes sources en exécutables, car un executable doit être recompilé si l'un de ses fichiers source est modifié.

Par exemple, un exécutable est constitué de fichiers objets et de bibliothèques de fonctions, qui eux même doivent être générés à partir de sources en langage C, ou en assembleur.

Ce sont des relations temporelles qui permettent à `make` de savoir s'il faut ou non regénérer un fichier. C'est la date de création des fichiers qui est utilisée par `make` pour vérifier la nécessité de la production d'un fichier.

2.2 Brève présentation

Le système Unix doit sa réputation à l'excellent environnement de programmation qu'il procure. `Make` fait partie des utilitaires qui contribuent à cette réputation.

Malgré l'accroissement de la taille des logiciels, `make` reste un outil indispensable et facilement utilisable, du moins pour celui qui le maîtrise.

Même si l'apparition d'outils graphiques avec des fonctionnalités identiques se généralise, ce ne sont que des *méta-make* utilisant les concepts de `make`.

Le prermier intérêt de `make` est de gagner du temps en ne recompilant que ce qui est strictement nécessaire. Mais l'intérêt de `make` dépasse cela, notamment de par l'extension des projets d'équipe. En explicitant les relations de dépendance entre les différentes parties de projets complexes, géré à plusieurs, `make` aide à coordonner les travaux de nombreux développeurs.

2.2.1 Quelle est l'entrée de `make`?

La commande

```
> make prg
```

indique que l'on désire « faire » la dernière version de `prg`. Si `prg` est un programme exécutable, cette commande indique que l'on va faire toutes les commandes de compilation et d'édition de lien nécessaire pour le construire.

Les actions de `make` sont basées sur une base de données textuelle, le `Makefile`.

Deux fichiers standards sont recherchés par `make` dans le répertoire courant lors de son appel :

1. `makefile`
2. `Makefile`
   On préfere généralement cette seconde forme car lors d'un `ls` du répertoire, le `Makefile` apparaît dans la tête de liste.

On peut spécifier un nom particulier à la place des noms par défaut sur la ligne de commande grâce à l'option `-f`, permettant ainsi d'avoir plusieurs bases de données dans le même repertoire, par exemple :

```
make -f test.mk
```

L'usage veut que l'on suffixe par `.mk` les fichiers d'entrée de `make`.

2.3 Comment sont organisés les `Makefile`s

Les deux notions fondamentales sur lesquelles est basé `make` sont les *cibles* et les *dépendances* (ou *prérequis)*. (Dans la littérature anglo-saxonne, on trouve les *targets* et les *dépendancies* ou *prerequisits*). Des règles de production sont utilisées pour générer, si nécessaire, la *cible* à partir des *dépendances*.

Une *cible* est l'objet qui va être produit à partir d'une ou plusieurs *dépendances*, par l'utilisation d'un outil approprié, si et seulement si les *dépendances* sont plus récentes. De manière générale, les *cibles* et les *dépendances* sont des fichiers. Et récursivement, les *dépendances* sont elles-mêmes les *cibles* d'autres règles de productions, et ce, jusqu'à être des fichiers source.

Ceci permet de garantir la cohérence entre les objets générés et leurs sources en un temps de regénération minimal.

2.3.1 Exemple de `Makefile`

Imaginons que nous travaillons sur un programme qui contient trois sources **C** : `main.c`, `pwet.c` et `util.c`, et utilise la librairie mathématique `libm.a`.

Pour compiler manuellement ce programme, on est obligé de faire les commandes suivantes :

```
> cc -c main.c
> cc -c pwet.c
> cc -c util.c
> cc -o pwet main.o pwet.o util.o /usr/lib/libm.a
```

Bien sûr, il est aussi possible de compiler l'ensemble des programmes en une seule ligne, mais ce ne serait pas très économique : chaque fichier peut être construit séparément, pas forcément par la même personne, et une *vraie* application peut prendre des heures à compiler, donc il faut utiliser les fichiers objets préexistant quand ils existent et qu'ils sont à jour.

Afin d'éviter d'avoir à taper ces commandes, on peut utiliser un fichier de type *batch*, mais ça ne résout pas le problème de la compilation " intelligente ". Pour cela, on a recours à `make` et à ses fichiers de description.

Un `Makefile` à l'allure suivante :

```
pwet : main.o pwet.o util.o
        cc -o pwet main.o pwet.o util.o

main.o : main.c util.h
        cc -c main.c
```

```
pwet.o : pwet.c util.h
        cc -c pwet.c

util.o : util.c util.h
        cc -c util.c
```

Tout fichier généré par un programme, c'est-à-dire tous sauf les fichiers sources, doit être la *cible* d'une règle.

La *cible* est placée en début de ligne et " : " la sépare de ses *dépendances*, qui sont placées à droite.

Ainsi, la dernière *règle* de l'exemple ci-dessus indique que `util.o` va être construite en utilisant les fichiers `util.c` et `util.h`. Si la date de dernière modification de `util.c` ou `util.h` est postérieure à celle de `util.o`, alors `make` lance un *shell* pour exécuter les commandes fournies sur les lignes suivantes. Toute commande est précédée d'une **tabulation**, sinon `make` émet un message assez incompréhensible du style :

```
make: Fatal error in reader: Makefile, line 3: Unexpected end of line seen
```

Ou encore ::

```
Makefile:2: *** missing separator.  Stop.
```

Par défaut, `make` commence l'analyse du fichier par la première règle qui y apparaît. Dans notre cas, il s'agit de la *cible* `pwet`. Cette cible est un exécutable obtenu en faisant l'édition de lien des trois objets dont elle dépend.

2.3.2 Comment `make` utilise ce `Makefile`

Après avoir lu l'ensemble du `Makefile`, `make` va interpréter la première règle. Mais avant d'exécuter les commandes liées à la règle, `make` doit s'assurer que toutes les *cibles* dont elle dépend sont à jour, c'est-à-dire que `main.o`, `pwet.o` et `util.o` existent et ont une date postérieure à celles de leurs *dépendances*. Cette dépendance en temps est connue grâce au système d'exploitation qui tient à jour des dates de dernières modifications pour les fichiers. On peut voir ces dates en tapant `ls -l`. Si ce n'est pas le cas, alors `make` va recontruire la ou les *cibles* concernées, qui vont être analysées de manière identique, et ce récursivement. La récursion s'achève lorsque l'on atteint un fichier existant qui n'est pas une *cible*. S'il n'y a pas de fichier sur disque, alors on a droit au message :

```
make: Fatal error: Don't know how to make cible 'pwet.c'

make: *** No rule to make cible 'pwet.c'.  Stop.
```

Par exemple, si le fichier `pwet.c` est modifié et que `make` est lancé, `make` va appeler le compilateur **C** pour générer `pwet.o`, puis faire l'édition de liens de `pwet`. Si c'est `util.h` qui est changé, alors `make` va recompiler `main.c`, `pwet.c` et `util.c`.

`Make` affiche à l'écran les commandes qu'il exécute, ce qui permet de s'assurer que tout se passe comme prévu.

Construire une application, c'est enchaîner une suite de commandes dans un ordre bien précis. Usuellement, on demande à `make` de construire le dernier élément de cette chaîne. Alors `make` remonte la chaîne de dépendances afin de déterminer les commandes qui doivent être exécutées. Finalement, on redescent les maillons un par un en construisant ce qui est nécessaire jusqu'à ce que l'objet final soit construit.

2.4 `Make`, sa vie, son œuvre

Quatre type d'élements sont présents dans les `Makefiles`: les règles, les définitions de variables, les inclusions, et les commentaires.

- une définition de variable est l'association à un identificateur d'une ligne de texte qui lui sera substituée dans le texte ensuite. C'est précisément le même fonctionnement que la clause `#define` en **C**, ce qui explique le nom de *macro* souvent utilisé pour les variables.
- les règles décrivent quand et comment reconstruire un ou plusieurs fichiers, les *cibles*, à partir d'un ou plusieurs fichiers, les *dépendances*.

– une inclusion permet de lire un fichier dans un `Makefile`, afin de définir des variables essentiellement.
– les commentaires sont précédés d'un `#`, et s'achèvent à la fin de la ligne.

Une ligne trop longue peut être coupée à tout moment par l'insertion d'un `\`, et ce quel que soit son type. Attention, si une ligne contient un `#` et s'achève sur un `\`, la ligne qui suit est également considérée comme un commentaire. De même, vérifier qu'il n'y a pas de blancs derrière le `\`, ce qui mène à :

```
sh:  : not found
```

2.4.1 Définition de variables

Si l'on met dans un fichier `Makefile`

```
nom = de diou
```

alors l'utilisation ensuite de `$(nom)` ou `${nom}` sera interprétée comme `de diou`.

On trouve typiquement dans les `Makefiles` des règles du genre :

```
pwet : main.o pwet.o util.o
	cc -o pwet main.o pwet.o util.o
	mv pwet ../bin
```

Si cette écriture est encore acceptable pour trois fichiers, elle n'est plus guère intéressante dès que le nombre de *dépendances* est important. Les objets étant utilisés à la fois pour exprimer les *dépendances* de la règle et pour lédition de liens, il raisonnable d'avoir un moyen qui évite d'en recopier la liste. C'est en partie à cela que sont destinées les variables. À l'aide de variables, on écrirait cette règle ainsi :

```
TARGET_BIN = ../bin
OBJ        = main.o pwet.o util.o

pwet : $(OBJ)
	cc -o pwet $(OBJ)
	mv pwet $(TARGET_BIN)
```

Lors de l'exécution, `make` va afficher :

```
cc -o pwet main.o pwet.o util.o
mv pwet ../bin
```

De manière générale, les variables sont utilisées pour créer des listes de noms de fichiers, des options à fournir aux producteurs de fichiers (les compilateurs par exemple), des noms d'outils à utiliser, *etc.* Examples :

```
CC = /labo/gnu/bin/gcc
OPT = -O
DBG =    # mettre -g pour avoir les possibilités de debug
```

Le grand intérêt de leur utilisation est de permettre de modifier un `Makefile` sans introduire d'incohérences : on est absolument sûr que ce sont les mêmes données qui sont utilisées partout.

Les noms de variables sont constituées de caractères différents de " `#` ", " `:` " et " `=` ". Il est recommandé de ne pas utiliser d'autres caractères que les lettres, les chiffres et "`_`", car l'interprétation peut varier suivant les implémentations.

L'usage veut que l'on utilise des majuscules dans les noms de variables, car cela permet de les faire ressortir du texte.

On fait référence à une variable grâce à `$(VAR)`, ou encore `${VAR}`, mais cette seconde forme est moins usuelle [1]. Elle a néanmoins l'avantage d'éviter la confusion avec l'opérateur de construction des librairie. Le caractère `$` a un sens particulier, et il faut utiliser `$$` pour l'obtenir. C'est parfois utile pour l'utilisation dans des *shells* lancés par `make` pour exécuter les commandes.

Les variables peuvent être utilisées partout, et sont des strictes substitutions de texte. Un esprit perverti pourrait ecrire une règle comme :

```
PWET = c

main.o : main.$(PWET) util.h
	$(PWET)$(PWET) -$(PWET) main.$(PWET)
```

1. Pourquoi ? Je l'ignore !

car `make` verra après substitution de `$(PWET)` la ligne `cc -c main.c`.

Les variables peuvent être affectées dans d'autres variables, par exemple :

```
ALC_INC      = -DMUT_H='"mut315.h" -I/labo/include
LOCAL_CFLAGS = $(CFLAGS) $(ALC_INC)
CC           = gcc -Wall -f-writable-string $(LOCAL_CFLAGS)
```

Par convention, les variables sont placées au début du `Makefile`, afin d'être facilement visualisées et changées. Il faut prendre garde de ne pas mélanger l'utilisation de variables avec l'utilisation de chaînes statiques. Il est fortement déconseillé de faire :

```
CC    = gcc -Wall

main.o : main.c util.h
        $(CC) $(CFLAGS) -c main.c

util.o : util.c util.h
        cc -c util.c
```

L'ordre de déclaration des macros n'a pas d'importance, tant qu'il n'y a pas de déclaration récursive. Le processus d'expansion des macros étant la substitution de texte, la récursion n'a pas de sens. L'autre restriction est qu'une macro doit être définie avant la ligne de règles de production qui l'utilise. L'utilisation d'une macros non définie s'évalue en la chaîne vide.

2.4.2 Macros par défaut

Un certain nombre de macros sont fournies par défaut par `make`. L'avantage de ces macros apparaît plus clairement dans l'utilisation des " règles de suffixe ", qui sont abordées plus tard.

`SHELL` indique sous quel *shell* les commandes doivent être exécutées. En ce qui nous concerne, cette variable sera presque toujours fixée à `sh`, car c'est le plus répandu et le plus portable des *shells*. Comme les différents *shells* ne sont pas compatibles, cette variable doit être manipulée avec précaution.

`MAKE` qui indique **le** `make` utilisé dans le `Makefile` en cours d'exécution. Sa valeur est malheureusement variable suivant les `make`s, mais c'est néanmoins cela qu'il faut utiliser lors de l'invocation récursive de `Makefile`s.

`CC` indique le compilateur **C** préféré sur le systeme. Lors d'un développement, on peut être amené à utiliser différents compilateurs **C**, et à la différence des *shells*, ils sont compatibles.

`CFLAGS` indique les options par défaut du compilateur **C**.

Et il y en a bien d'autre, voir pour celles qui sont définies par défaut pour **Alliance** et qu'il est nécessaire d'utiliser dans ce cadre le document *Alliance Tools Developement Conventions*.

Par convention, les noms de macros sont en majuscule. Nous suivrons cette convention autant que possible.

On peut visualiser l'ensemble des macros prédéfinies par défaut en lançant `make` avec l'option `-p`.

2.4.3 Définition de macros sur la ligne de commande et dans le *shell*

On peut definir des macros directement sur la ligne de commande de `make`. Par exemple :

```
make prg LIBS="-lm -lX11 -lfred"
```

va positionner la variable `${LIBS}`. Attention, il ne doit pas y avoir de blancs autour du signe `=`, et s'il y a plusieurs arguments, alors il faut mettre des guillemets afin que le `shell` les considères comme un seul et unique argument.

`Make` hérite directement des variables du *shell*. La variable `$HOME` indique le chemin d'accès de la racine du compte Unix courant. On peut utiliser cette variable dans un `Makefile` sans autre forme de procès :

```
$ echo $HOME
/users/cao/fred
```

```
$ cat Makefile
prg: main.o
     cc -o ${HOME}/bin/pwet main.o -lm
$ make
cc -o /users/cao/fred/bin/pwet main.o
```

2.4.4 Ordre de priorité des macros

Dans l'ordre du moins prioritaire, vers le plus prioritaire :

1. définitions par défaut de `make` ;
2. variables courantes du *shell* ;
3. définitions présentes dans le `Makefile` ;
4. macros définies sur la ligne de commande.

Une option de `make`, l'option `-e`, permet de d'inverser les priorités 3 et 2. Ainsi, les variables courantes du *shell* sont plus prioritaires que les définitions dans le fichier d'entrée de `make`.

Les règles de priorité pour la redéfinition des variables permettent une grande souplesse. Si la variable `MACHINE` est définie dans le *shell* d'où est lancé le `make`, alors une référence à `$(MACHINE)` dans le `Makefile` prendra la valeur de la variable du *shell*. Si le `Makefile` contient une ligne du type `MACHINE = TI57`, alors c'est cette valeur qui sera utilisée dans la suite du `Makefile`. Si lors de l'invocation de `make` la ligne de commande est du style `make MACHINE=ZX80`, alors c'est cette valeur qui sera utilisée dans la suite du `Makefile`.

2.4.5 Substitution de chaîne dans les macros

Imaginons que nous avons la macro :

```
SOURCES = main.c util.c pwet.c
```

Alors, `${SOURCES:.c=.o}` vaut `main.o util.o pwet.o`. Dans ces conditions, il est aisé de maintenir de façon cohérente un groupe d'objets à partir de sources.

La substitution ne peut avoir lieu qu'à la fin d'une macro ou immédiatement avant un espace.

2.4.6 Macro internes pour les cibles et les dépendances

Certaines variables, dites *dynamiques*, sont positionnées par `make` lors de l'évaluation de chaque règle. En voici le sens :

`$@` est le nom de la *cible* visée par la règle. Elle est assez usité, car en général la cible est le nom du programme que l'on veut créer.

```
pwet : main.o pwet.o util.o
       ${CC} -c $@ main.o pwet.o util.o
```

`@` est un nom de variable comme les autres, mais ne peut être utilisé que dans les commandes, et pas dans les règles de dépandance. Dans certaines version de `make`, Il est possible d'utiliser `$$@` dans les dépendances. Exemple :

```
CMDS = cat dd echo date diff ar ranlib
${CMDS} : $$@.c
          ${CC} -o $@ $?
```

`$?` est la liste des *dépendances* qui sont plus récentes que la *cible*, chaque élément séparés du suivant par un blanc.

`$%` est le nom d'un membre d'une librairie en cours de création.

`$*` est le nom hormis le suffixe, de la *cible* sélectionée, **uniquement** pour les règles implicites.

`$<` est le nom hormis le suffixe, de la *dépendance* sélectionée, **uniquement** pour les règles implicites. (Voir *Écriture des règles* pour les règles implicites.)

Des variantes de ces variables peuvent être obtenues en utilisant les constructions "`$(`*C*`D)`", afin d'obtenir le chemin d'accès et "`$(`*C*`F)`" pour obtenir uniquement le nom de fichier d'une variable dynamique, ou *C* est soit "`@`", "`?`", "`%`", "`*`" ou "`<`".

Certaines version de `make` proposent plus de variables dynamiques, des moyens d'affectation plus puissants, des conditions d'affectation ou de production, mais ils n'y a pas de consensus syntaxique, ce qui proscrit leur utilisation dans les `Makefiles` d'**Alliance**.

2.4.7 Règles sur les suffixes

Les conventions et leurs bienfaits

Sur un système Unix, un fichier **C** a toujours l'extension `.c`, de même, un fichier assembleur a l'extension `.s`. Et le résultat d'une compilation donnera un `.o`. Ces conventions permettent à `make` d'exécuter un certain nombre d'actions en fonctions de règles sur les suffixes.

```
OBJS = main.o pwet.o util.o
LIB  = -lm
pwet : ${OBJS}
       cc -o $@ ${OBJS} ${LIB}
```

Cet exemple est un `Makefile` qui fonctionne sans avoir besoin d'aucune autre règle. En construisant `pwet`, `make` vérifie que `main.o` est à jour. Il fait cela en regardant dans le répertoire courant s'il n'existe pas un fichier qui, en utilisant une règle sur les suffixes, permettrait de construire `main.o`. S'il trouve un `main.c` qui est plus récent que `main.o`, alors il applique la règle de compilation **C** par défaut. Sinon, il chercher un `main.f`, et appelle le compilateur **Fortran**, etc. On pourrait utiliser ce `Makefile` pour construire n'importe quel `.o` à partir de sa source **C**.

Le but des règles sur les suffixes est de généraliser les appels aux commandes de construction des programmes. Voici les règles de l'exemple précédent :

```
.SUFFIXES : .o .c .s
.c.o :
     ${CC} ${CFLAGS} -c $<
.s.o :
     ${AS} ${ASFLAGS} -o $@ $<
```

La ligne `.SUFFIXES:` est une forme de règle de dépendance, mais avec une différence. Les extensions sur cette ligne sont les extensions que `make` va considérer. Si un prérequis s'achève avec l'un de ses suffixes, mais n'a pas de règle associée dans le `Makefile`, `make` va rechercher une règle sur les suffixes qui soit applicable.

Deux macros internes sont destinées à être utilisées dans les règles implicites et là uniquement : `$<` qui prend le nom du prérequis employé pour reconstruire la cible, et `$*`, qui représente le nom du prérequis sans extension.

On se rappellera avec bonheur l'option `-p` qui affiche, lors de l'appel de `make`, l'ensemble des règles implicites définies.

```
make -fp - </dev/null | more
```

Les `make` modernes supportent une autre manière, plus claire, de définir des règles implicites. On utilise le caractère `%` comme substitut au nom de fichier, et ensuite on écrit la règle de manière identique à une règle explicite, en utilisant les variables automatiques. C'est assez pratique car on peut faire apparaître des sous répertoire dans l'expression de la cible ou de la première dépendance.

```
obj/%.o : %.c
    ${CC} ${CFLAGS} -o $@ -c $<
```

Priorité dans les règles sur suffixes

L'ordre de recherche des dépendances possibles lors de l'évaluation d'une règle implicite est dictée par l'ordre d'apparition des suffixes sur la ligne `.SUFFIXES:`. Cela peut créer des problèmes si l'on à pas songé à exprimer toutes les règles, et même écraser des sources...

Écrire ses propres règles implicites

Ce n'est pas une bonne idée de réecrire les règles sur suffixes standard, car si quelqu'un récupère un tel `Makefile`, il faudra qu'il le lise avec ces considérations en tête, ce qui n'est pas forcément simple. Cependant, dans bien des cas on peut utiliser `make` dans un cadre particulier. Prenons l'exemple d'un document écrit en LaTeX. Le compilateur de document s'appelle `latex` (suffixe `.tex` qui crée un `.dvi`), l'outil de gestion de la bibliographie s'appelle `bibtex` (suffixe `.bib` qui crée un `.bbl`). Transformer la sortie de LaTeX en PostScript, (`.ps`), se fait grâce à `dvips`, et visualiser le document se fait à l'aide de `ghostview`. Afin d'automatiser la production de document LaTeX, on peut écrire :

```
.SUFFIXES:
```

```
.SUFFIXES: .tex .dvi .bib .bbl .ps

.tex.dvi :
	latex $<
.bib.bbl :
	bibtex $*
	touch $*.tex
.ps.dvi :
	dvips $< -t a4 -o
	ghostview -a4 $@
```

Cas particulier : les bibliothèques

Afin de pouvoir spécifier des fichiers individuels dans la construction des bibliothèques, `make` propose des extensions à sa syntaxe des cibles et dépendances par l'emploi de parenthèses. Un nom de fichier comme :

```
libMut316.a(mbk_util.o)
```

ou une macro comme

```
${LIBUTIL}(mbk_util.o)
```

font référence à une bibliothèque qui contient un fichier objet. Dans certaines version de `make`, il est même possible d'indiquer le nom d'une fonction particulière dans la bibliothèque :

```
libMut316.a((_namealloc))
```

Les conventions sur la manière dont le compilateur génère les noms de fonctions sont documentées sur le système (on peut le vérifier à l'aide de `nm`). La règle par défaut pour la construction des bibliothèques est la suivante :

```
.c.a :
	${CC} -c ${CFLAGS} $<
	ar rv $@ $*.o
	rm -f $*.o
```

Par convention, c'est l'extension `.a` qui est usité pour les bibliothèques, mais ce n'est pas une nécessité, et c'est la présence des parenthèses qui va activer la règle sur suffixes `.c.a`. Notons également que par défaut c'est le fichier `.c` qui est utilisé, et non un objet `.o`.

Exemple :

```
prg : main.o libpwet.a(pwet.o) libpwet.a(utli.o)
	$(CC) -o $@ main.o libpwet.a
```

Imaginons que `util.c` a changé depuis la dernière fois que `libpwet.a` fût construite.

```
> make
cc -c util.c
ar rv libpwet.a util.o
r - util.o
rn -f util.o
cc -o prg main.o libpwet.a
```

En construisant `libpwet.a`, `make` commence par positionner ses macros internes, avec la différence que `$@` prend la valeur *à gauche* des parenthèses, et non l'ensemble. Une autre macro interne, `$%`, prend la valeur de ce qui est placé entre les parenthèses.

Que ce passe t-il si l'on désire créer des éléments d'une bibliothèques avec des commandes différentes ? Par exemple lorsque certains fichiers sont au point et d'autre ont encore besoin d'être debuggé ? On utilise des règles avec des deux deux points " `::` ". Voyons cela :

```
libpwet.a :: util.c # tout a fait au point
	${CC} -O -c ${CFLAGS} util.c
```

```
	ar rv $@ util.o
	rm -f util.o

libpwet.a :: pwet.c # encore plein de bugs
	${CC} -g -DDEBUG -c ${CFLAGS} pwet.c
	ar rv $@ pwet.o
	rm -f pwet.o
```

2.4.8 Les commandes

Chaque commande exécutée par `make` se comporte comme si elle était lancée d'un *shell*. D'ailleurs, elle l'est! Ainsi, on peut utiliser n'importe quelle commande, de façon quasi identique au *shell* de sous `make`, mais il faut en modifier un peu la syntaxe, à cause des intéractions avec les macros.

Remarques utiles

Notons au passage l'utilisation de `*`, (ou de tout autre caractère du *shell* qui permet de créer une liste de fichiers), qui indique que tous les fichiers d'extension `.2` du répertoire courrant doivent être envisagés comme *dépendances*.

```
all : mbk.ps mbk.m

mbk.ps : *.2
	groff -Tps -man -fP *.2 > mbk.ps

./catman/mbk.m : *.2
	sed -e "s/nnn/315/g" $? | nroff -e > ./catman/mbk.m
```

Ceci va construire à la fois les cibles `mbk.ps` et `mbk.m`. Il y a une subtilité : `mbk.ps` contiendra à la fin l'ensemble des fichiers du répertoire formatés, alors que `mbk.m` ne contiendra que les fichiers plus récent que la précédente version de ce même fichier.

`Make` traite *chaque ligne* de commande en l'exécutant dans sons propre *shell*. Ainsi, sous `make`, la commande :

```
cd tmp
rm -f *
```

va créer un *shell* dont le répertoire de départ est le répertoire courant et effectuer le `cd`, puis la ligne étant finie, le *shell* s'achève. Un autre *shell* est crée pour la ligne suivante, dont le répertoire de départ est le répertoire courant et effectuer le `rm -f *`, ce qui n'est sans doute pas le comportement attendu...

En dehors de problèmes que cela peut occasionner, le lancement d'un *shell* est une opération coûteuse, ce qui peut grandement déteriorer les performances d'un `Makefile`.

Solutions pour exécuter des commandes dans un même *shell* :

```
cd tmp; rm -f *
```

Ou encore :

```
cd tmp; \
rm -f *
```

Attention, il ne faut pas de blancs après le \! On préférera cette seconde solution. De même, on utilisera toujours le *Bourne Shell* dans un `Makefile` pour des raisons de portabilité.

Erreurs

Lorsqu'une commande échoue, elle retourne une valeur diff'érente de zéro au *shell*, qui va la transmette à `make`. Dans ce cas, `make` stoppe.

Il y a des cas ou cela n'est pas souhaitable, par exemple lorsque l'on essaye d'effacer un fichier objet qui n'existe pas. Il suffit d'ajouter un
- devant la commande pour que `make` ignore sa valeur de retour :

```
cleanlib : libpwet.a
    -rm libpwet.a pwet.o util.o
```

Si un des trois fichier n'exsite pas, cela signifie simplement qu'il n'avait pas été construit, mais ce n'est pas une raison pour arreter le `Makefile`.

On peut simuler la présence de `-` devant toutes les commandes avec la présence d'un `.IGNORE`: dans le `Makefile`, ou d'appeler `make` avec l'option `-i`. L'option `-k` est plus subtile : si une cible ne peut être construite, alors `make` recherche les autres cibles indépendantes de celles la pour les construires.

2.4.9 Gestion de projet

Quand les projets évoluent, que plusieurs personnes sont impliqués, les `Makefiles` grossisent et deviennent plus complexes. On ne peut plus généraliser l'utilisation des règles implicites, et l'utilisation de la récursion devient très usuelle.

Quelques difficultés de `make` dans ces conditions sont :

– construire sur plusieurs répertoires. Il n'est pas évident de faire travailler `make` sur un ensemble de répertoires ;
– différences dans les options de compilation : `make` ne reposant que sur les dates des fichiers, si l'on modifie l'option de compilation de tel ou tel fichier, il n'y aura pas de recompilation automatique ;
– dépendances cachées, dans les fichiers d'entête. Les `Makefiles` doivent contenir explictement les dépendances sur les fichiers d'entêtes.

Afin de pouvoir contourner ces limitations, des techniques particulières sont mises en œuvre.

Fausses cibles

Il y a des cibles qui ne sont pas des fichiers, mais qui sont simplement des noms simples pour effectuer, en général, un ensemble d'actions.

```
all : cat diff grep more

cat : cat.c cat.h
    ${CC} -o $@ cat.c

diff : diff.c
    ${CC} -o $@ diff.c

grep : grep.c
    ${CC} -o $@ grep.c

more : more.c
    ${CC} -o $@ more.c
```

Pour tout construire, on appellera `make all`, sans pour autant qu'un fichier `all` soit créé.

Un autre grand classique est le “ nettoyage ” :

```
clean :
      -rm -f *.o
```

L'intérêt essentiel des fausses cibles, et des fausses dépendances, est de détourner `make` de son chemin habituel, et de forcer une action. Quelques applications de cette technique seront présentées dans la suite. Si `make` utilise les dates sur les vrais fichiers pour calculer les reconstructions, il utilise des règles prédéfinies et arbitraires pour les régles utilisant des non-fichiers.

- une cible n'existant pas sous forme de fichier n'est jamais à jour. Ainsi, les commandes qui lui sont associées sont toujours exécutées.
- un prérequis qui n'est pas un fichier est toujours une date plus récente que sa cible. Ainsi une cible avec un faux prérequis est toujours reconstruite.

L'utilisation d'un nom n'existant pas sur le disque pour faire un faux prérequis impose que ce nom soit défini comme cible quelque part dans le `Makefile` même si il n'active aucune commande. Si ce n'est pas le cas, `make` va s'arreter en affichant :

```
Make: Don't know how to make fichier
```

Il est assez classique d'utiliser des cibles qui sont effectivement des fichiers, mais qui ne sont là que pour indiquer qu'une action qui d'habitude ne créée pas de fichier a eu lieu. Par exemple, l'effacement de la table des symboles :

```
stripexec: cat more diff
    strip $?
    touch $@
```

`Make` sait ainsi s'il faut ou non *strippé* les fichiers en argument. Si l'on lance `make stripexec` et qu'il n'est pas nécessaire d'effacer les tables de symboles, alors `make` va répondre :

```
‘stripexec is up to date.
```

Exécution récursive de `Make`

Dans les gros projets, les sources sont conservées dans des répertoires séparés, souvent gérées par différents individus.

Dans ces circonstances, `make` ne peut être utilisé avec un unique `Makefile`. Ce n'est d'ailleurs pas souhaitable, car chaque partie du projet doit certainement être compilé de façon différente. Les fichiers sont organisés hiérarchiquement grâce à la structure en arbre du système d'exploitation.

Assez généralement les sous-répertoires de la hiérarchie servent à construire des libraries de fonctions, mais ce n'est pas une obligation. Prenons l'exemple d'un outil qui lit un fichier et travaille sur une structure de données : le lecteur de fichier se trouve dans le sous-répertoire `read`, les structures de données destinées à reçevoir ce qui est lu doit être mis dans `struct`, et finalement l'application est dans `appl`. Le `Makefile` dans `root` peut avoir l'allure suivante :

```
READ   = ../read/libRead.a
STRUCT = ../struct/libStruct.a
OBJ    = appl.o

prg : $(READ) $(STRUCT) $(OBJ)
    $(CC) -o $@  $(OBJ) $(READ) $(STRUCT)

$(READ) :
    cd ../read ; make

$(STRUCT) :
    cd ../read ; make
```

En fait, au lieu d'appeler `make` directement, on utilise `$(MAKE)` afin d'être sûr, quelque soit le répertoire d'appel, d'appeler le `make` qui a servi la première fois. De plus, `$(MAKE)` passe les arguments de la ligne de commande récursivement, hormis les options `-f`, donc c'est un nom par défaut de `make` qui doit être utilisé dans les sous-répertoires, et `-p` et `-d` qui ne sont utiles qu'au *debug*. Chaque invocation de `make` recommence avec les valeurs par défaut des variables. Donc les macros qui doivent être passées aux diverses niveaux de `Makefile`s doivent être explicitememt indiquées lors de l'appel sur la ligne de commande :

```
world:
    $(MAKE) cat more diff "CFLAGS=$(CFLAGS) -DDEBUG" "VERSION='2.3'"\
            "INCLUDE=$(INCLUDE)" "LIBS=$(LIBS)"
```

Il existe la macro `VPATH` qui sert à indiquer l'ensemble des répertoires dans lesquels rechercher les fichiers de dépendance. Les règles par défaut ne font pas usage de `VPATH`. Il faut être complètement explicite dans les descriptions. Les répertoires sont parcourus dans l'ordre fourni dans la macro, et l'on s'arrete sur la première occurence du fichier nommé. Il est fortement recommander de n'utiliser `VPATH` que pour la recherche des sources.

Imposer une reconstruction

La méthode bestiale pour assurer la cohérence des objets par rapport aux sources consiste à effacer tous les `.o` avant de refaire une cible.

```
test : clean
     ${MAKE} read.o "INCLUDE=/labo/include"

clean:
     -rm read.o test
```

Mais on peut être plus rusé, mais cela nécessite de bien saisir les mécanismes des règles concernant les fausses cibles :

```
faire:
     ${MAKE} read.o INCLUDE=/labo/include

refaire:
     ${MAKE} read.o INCLUDE=/labo/include IMPOSE=impose_faire

impose_faire:

read.o : read.c ${IMPOSE}
```

Il faut définir une macro, `IMPOSE`, et une fausse target, `impose_faire`. Le cœur du mécanisme repose sur l'interprétation des dépendances par `make`. Par définition, une dépendance qui n'est pas un fichier est toujours à jour, ce qui impose la reconstruction de sa cible. Ainsi, si le vrai fichier `read.o` dépend du faux fichier `impose_faire`, alors `make` va refaire `read.o`. Afin de choisir si on doit reconstruire ou non `read.o`. C'est là tout l'intérêt de la macro `IMPOSE`. Normallement, elle est nulle, donc `read.o` ne dépend que de `read.c`. Mais si `IMPOSE` est défini, alors tout sera reconstruit.

```
make refaire
```

ou encore :

```
make IMPOSE=impose_faire
```

2.4.10 Versions différentes à l'aide des règles sur suffixes

On peut différencier des fichiers objets issus de la même source mais avec des options de compilation différentes à l'aide des règles implicites. Ce n'est possible que parce que le l'éditeur de liens n'a pas de conventions strictes sur le nommage des fichiers qu'il accepte. Utilisons par exemple l'extension `.g` pour les objets contenant des informations de debug.

```
OBJS       = main.o pwet.o util.o
OBJS_DEBUG = main.g pwet.g util.g
.SUFFIXES : .g

pwet : $(OBJS)
      ${CC} -c $@ $(OBJS)

test : $(OBJS_DEBUG)
      ${CC} -c $@ $(OBJS_DEBUG)
```

2.4.11 Fichiers d'entête

Comme l'on utlise des fichiers d'entête, l'utilisation des règles implicite perd pas mal d'intérêt pour la compilation **C**. Il faut expliciter les dépendances entre les différents `.h` et les différents `.c`.

`Make` peut nous aider à générer cette liste de dépendances vis-à-vis des `.h` et des `.c`. C'est l'option `-M` qui permet de faire cela. Notons que cette option est rarement utilisée, car il est impératif d'écrire le `Makefile` en même temps que l'on développe l'application, et dans ces conditions on connaît forcément les liens entre les différents fichiers. Elle peut être utilisée pour vérifier qu'il n'y a pas d'erreurs, par exemple.

2.4.12 Inclusion de définitions

Afin de partager un ensemble de définitions, souvent imposés dans le cadre d'un large développement d'outils devant fonctionner ensembles, il est possible d'inclure des fichiers contenant des 'efinitions de macros dans les `Makefiles` grâce à la directive `include` *fichier*.

Par convention, on place les inclusions au début du `Makefile`, et l'on suffixe des fichiers de définition par `.mk`.

2.4.13 Cibles spéciales

`.DEFAULT:` les commandes associées à cette cible sont exécutées si `make` ne peut pas trouver le moyen de construire aucune cible ;

`.IGNORE:` ignore les codes d'erreur ;

`.PRECIOUS:` les fichiers indiqués ici ne sont pas effacés si `make` vient à s'arreter, par une interruption ou une règle qui échoue ;

`.SILENT:` n'affiche pas les commandes lors de leur exécution ;

`.SUFFIXES:` les dépendances associés avec cette cible sont les suffixes que `make` va connaître et auxquels peuvent être associés les règles implicites. S'il n'y a pas de suffixes associés, alors tous les suffixes précédemment définis sont effacés.

Chapitre 3

Tables de Hachâge: correction

Ce chapitre présente les sources permettant de créer la bibliothèque de hachâge que nous avons étudiés en TD. Ces sources sont dans le domaine public, mais j'en suis tout de même l'auteur, ... Veuillez avoir une pensée pour moi si vous les utilisez !

Vous pouvez récupérer les sources en cliquant ici!

3.1 Makefile

```
# This file belongs to the Hashing library serving a course basis
# Developed by Frederic Petrot: Frederic.Petrot@lip6.fr
# This file is subject to the GNU copyleft %-)
#
# Compiles the hashing stuff and builds a small example

OBJ    = main.o
OBJL   = error.o hash.o
LIBH   = libh.a
EXEC   = ht
CC     = gcc
CFLAGS = -Wall -O8

$(EXEC): $(OBJ) $(LIBH)
$(CC) $(CFLAGS) -o $@ $(OBJ) -L. $(LIBH)

$(LIBH) : $(OBJL)
$(AR) rv $@ $?
$(RM) $?

clean:
$(RM) -f $(OBJ)

realclean: clean
$(RM) -f core a.out $(EXEC) *~ *.bak *.BAK
```

3.2 error.h

```
/*
 * Sample error reporting routine, ...
 * Nothing really bright, but are useful things always bright ?
 *
 * Author: Frédéric Pétrot
 * Date  : 01/09/98
 * File subject to the GNU copyleft
 */
```

```
enum {ERROR, WARNING, INFORMATION};
extern void ReportError(int, char *, char *, ...);
```

3.3 error.c

```
/*
 * Sample error reporting routine, ...
 * Nothing really bright, but are useful things always bright ?
 *
 * Author: Frédéric Pétrot
 * Date  : 01/09/98
 * File subject to the GNU copyleft
 */
#include <stdio.h>
#include <stdarg.h>

#include "error.h"

/*\
 *                       File scope variables
\*/

static char *Severity[] = {"Error", "Warning", "Information"};

void ReportError(int level, char *function, char *format, ...)
{
va_list ap;

   va_start(ap, format);

   level %= sizeof(Severity) / sizeof(*Severity);

   fprintf(stderr, "%s in function %s: ", function, Severity[level]);
   vfprintf(stderr, format, ap);
   va_end(ap);

   if (level == ERROR)
      exit(1);
}
```

3.4 hash.h

```
/*\
 * A few hash table management function.
 * The interface is quite simple, and relies on a few functions.
 * It could be extended to mimick the hsearch (3) interface to compare
 * the efficienty of both implementations.
 *
 * Author: Frédéric Pétrot
 * Date  : 01/09/1998
 * Credits:
 *
 * Thanks to :
 *   Ludovic Jacomme for many implementation hints, and a few
 *   nice ideas like putting the access functions within the hashtable,
 *   resize using prime number table, and so on.
 *
 *   Donald E. Knuth for his nice Stanford Graphbase hash function
```

```
 *
 * This file is subject to the GNU copyleft
\*/

#ifndef _HASH_H_
#define _HASH_H_

/* Maximun Chain:
 * This is the maximun length of the external chaining as defined by
 * default.
 */

#define MAX_CHAIN_LENGTH 5

/* This is working structure that has nothing to do with the final
 * user, IMHO
 * However, C doesn't have a clean way to do so, thus it appears here.
 */

typedef struct entry {
   struct entry *next; /* next element for external chaining */
   void         *key;  /* search key */
   void         *data; /* meaningfull data */
} entry;

typedef struct htable {
   /* Variables */
   entry  **table;                  /* elements themselves                  */
   unsigned long slots;             /* number of slots of the table         */
   int           resize;            /* number of resize                     */
   unsigned long length;            /* attained length of external chaining */
   unsigned long entries;           /* number of entries in the table       */
   int           overflow;          /* indicates that an overflow occured   */
   unsigned long threshold;         /* threshold indicating when to resize  */

   /* Functions */
   int           (*equal)           /* function for key comparison          */
 (void *, void *);
   unsigned long (*code)(char *);   /* function for hash code computation   */
   unsigned long (*stretch)
                  (unsigned long); /* function for new size computation    */
   void          (*display)(char *,
                    void *);       /* function for display data            */
   void          (*remove)(char *,
                          void *); /* function to free size computation    */
} htable;

typedef htable * hashtable;

extern hashtable  CreateHashTable(unsigned long,
                                  unsigned long,
                                  unsigned long (*)(char *),
                                  unsigned long (*)(unsigned long),
                                  void          (*)(char *, void *),
                                  void          (*)(char *, void *));
extern void       DestroyHashTable(hashtable);
extern int        InstallKey(hashtable, char *, void *);
extern int        LookupKey(hashtable, char *, void **);
extern void       DumpHashTable(hashtable);
extern int        RemoveKey(hashtable, char *);

#endif  /* _HASH_H_ */
```

3.5 hash.c

```
/* vim:set syntax keyword Type entry
 *
 * This algoritmic part of this code is shamelessly borrowed from:
 * The Stanford Graphbase, Donald E. Knuth, Addison Wesley 1994
 * It defines a hash function that is much (about twice) more
 * efficient than the other implementation also included here.
 *
 * Thanks to Ludovic Jacomme for many fruitfull discussions on this
 * matter.
 * Author: Frédéric Pétrot
 * This file is subject to the GNU copyleft
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "error.h"
#include "hash.h"

/*\
 *                            Local types
\*/


/*\
 *                           Local defines
\*/
#define HASH_MULT 314159
#define HASH_PRIME 516595003

/*\
 *                       File scope variables
\*/
static int primes[] = {
           11,
           23,
           47,
           97,
          197,
          397,
          797,
         1597,
         3203,
         6421,
        12853,
        25717,
        51437,
       102877,
       205759,
       411527,
       823117,
      1646237
#if HUGE_MEMORY
      3292489,
      6584983,
     13169977,
```

```
     26339969,
     52679969
#endif

};

/* A typical chicken and egg problem! */
#define PRIME_TABLE_SIZE (sizeof(primes) / sizeof(primes[0]))

/*\
 *                          Local functions:
\*/

/* Item management:
   we could make some sort of block allocation to improve the
   performance on large set of data, but I just won't do it right now */
static entry *AddItem(entry *head, char *key, void *data)
{
entry *Item;

   Item = (entry *)malloc(sizeof(entry));
   if (!Item)
      ReportError(ERROR, __FUNCTION__,
                  "Not enough memory to allocate a hash table items");

   Item->key  = key;
   Item->data = data;
   Item->next = head;

   return Item;
}

static void FreeItem(entry *item)
{
   free(item);
}

static int SearchItem(entry *head, void *key, int (*equal)(void *, void *))
{
entry *Item;

   for (Item = head; Item; Item = Item->NEXT)
      if (equal(key, Item->key))
 return Item;
   return NULL;
}

static int CountItem(entry *head, void *key, int (*equal)(void *, void *))
{
entry *Item;

   for (Item = head; Item; Item = Item->NEXT)
      if (equal(key, Item->key))
 return Item;
   return NULL;
}

/* Comparison function */
int streq(char *s, char *t)
{
   return !strcmp(s, t);
}
```

```
/* Hashing function:
   This one is pretty good as far as I've experimented.
   Thanks to Don Knuth for this (among other things). */

static unsigned long hashcode(char *key)
{
long Code;

   for (Code = 0; *key; key++) {
      Code += (Code ^ (Code >> 1)) + HASH_MULT * (unsigned char)*key;
      while (Code >= HASH_PRIME)
         Code -= HASH_PRIME;
   }
   return Code;
}

/* New size:
 * For efficiency reasons, a hash table is better off if its size is a
 * prime number.
 * Thus we compute a new prime that's approximatively twice the size
 * of the previous one using a table. */

static unsigned long tablesize(unsigned long n)
{
int i;

   for (i = 1; i < PRIME_TABLE_SIZE; i++)
      if (primes[i] > n)
         return primes[i];

   return (n << 1) + 1; /* an easilly computed odd integer */
}

static void displaydata(char *s, void *d)
{
   printf("(%s 0x%p) ", s, d);
}

static void removedata(char *s, void *d)
{
   /* Do nothing, since we do not know what's in the fields */
}

/*\
 *                       External Functions
\*/

htable *CreateHashTable(unsigned long size,
                        unsigned long threshold,
                        int           (*equal)(void *, void *),
                        unsigned long (*code)(void *),
                        unsigned long (*stretch)(unsigned long),
                        void (*display)(void *, void *),
                        void (*remove)(void *, void *))
{
htable *HashTable;

   HashTable = (htable *)calloc(1, sizeof(htable));
   if (!HashTable)
      ReportError(ERROR, __FUNCTION__,
                  "Not enough memory to allocate hash table structure");
```

```
   /* Initialize the hash table mathods:
    * Code:    computes the hash code
    * Stretch: computes the growing factor of the hash table when resizing
    * Display: displays the content of the data pointer
    * Remove:  deletes the content of the data pointer
    */

   HashTable->code      = equal ? equal : streq;
   HashTable->code      = code ? code : hashcode;
   HashTable->stretch   = stretch ? stretch : tablesize;
   HashTable->display   = display ? display : displaydata;
   HashTable->remove    = remove ? remove : removedata;

   HashTable->threshold = threshold > 0 ? threshold : MAX_CHAIN_LENGTH;
   HashTable->slots     = HashTable->stretch(size);

   HashTable->table     = (entry **)calloc(HashTable->slots, sizeof(entry *));

   if (!HashTable->table)
      ReportError(ERROR, __FUNCTION__,
                  "Not enough memory to allocate '%d' hash table slots",
                  HashTable->slots);

   return HashTable;
}

int InstallKey(htable *hash, char *key, void *data)
{
unsigned long Code;
entry         *ScanItem;
int           Index;

   Code = hash->code(key) % hash->slots;
   for (ScanItem = hash->table[Code], Index = 0;
        ScanItem != NULL;
        ScanItem = ScanItem->next, Index++)
      if (hash->equal(key, ScanItem->key) == 0)
         return 1;

   /* Keep track on the longest chain in case we overflow */
   hash->length = Index > hash->length ? Index : hash->length;

   /* Resize:
    * We must reenter all data into a newly created table. */
   if (Index > hash->threshold && !hash->overflow) {
   entry **OldTable, **NewTable;
   int    OldSize, NewSize;

      OldTable = hash->table;
      OldSize  = hash->slots;
      NewSize  = hash->stretch(hash->slots);

      if (NewSize == primes[PRIME_TABLE_SIZE - 1])
         NewTable = NULL;
      else
         NewTable = (entry **)calloc(NewSize, sizeof(entry *));

      if (NewTable == NULL)
         hash->overflow = 1;
      else {
         hash->resize++;
```

```
         hash->table  = NewTable;
         hash->slots  = NewSize;

         /* Install the old entries */
         for (Index = 0; Index < OldSize; Index++)
            for (ScanItem = OldTable[Index];
                 ScanItem;
                 ScanItem = ScanItem->next)
               (void)InstallKey(hash, ScanItem->key, ScanItem->data);

         /* Delete the old items, but not the value they point to */
         for (Index = 0; Index < OldSize; Index++)
            for (ScanItem = OldTable[Index];
                 ScanItem;
                ) {
            entry *Item;

               Item     = ScanItem;
               ScanItem = ScanItem->next;
               FreeItem(Item);
            }

         free(OldTable);
      }
   }
   hash->table[Code] = AddItem(hash->table[Code], key, data);
   hash->entries++;
   return 0;
}

int LookupKey(htable *hash, char *key, void **data)
{
unsigned long Code;
entry         *ScanItem;

   Code = hash->code(key) % hash->slots;
   for (ScanItem = hash->table[Code];
        ScanItem != NULL;
        ScanItem = ScanItem->next)
      if (hash->equal(key, ScanItem->key) == 0) {
         *data = ScanItem->data;
         return 1;
      }

   return 0;
}

int RemoveKey(htable *hash, char *key)
{
unsigned long Code;
entry         *ScanItem, *PreviousItem;
int           Length;

   Code = hash->code(key) % hash->slots;

   for (ScanItem  = hash->table[Code], Length = 0;
        ScanItem != NULL;
        ScanItem  = ScanItem->next, Length++);

   for (PreviousItem = NULL, ScanItem = hash->table[Code];
        ScanItem != NULL;
        PreviousItem = ScanItem, ScanItem = ScanItem->next
```

```
     ) {
      if (hash->equal(key, ScanItem->key) == 0) {
         hash->remove(ScanItem->key, ScanItem->data);
         if (PreviousItem == NULL) {
            hash->table[Code] = ScanItem->next;
            FreeItem(ScanItem);
         } else {
            ScanItem = ScanItem->next;
            FreeItem(PreviousItem->next);
            PreviousItem->next = ScanItem;
         }
         hash->entries--;
         if (hash->length == Length)
            hash->length--;
         return 1;
      }
   }
   return 0;
}

void DumpHashTable(htable *hash)
{
int Index;
entry *ScanItem;

   for (Index = 0; Index < hash->slots; Index++) {
      for (ScanItem = hash->table[Index];
           ScanItem;
           ScanItem = ScanItem->next)
         hash->display(ScanItem->key, ScanItem->data);
      puts("");
   }
   printf("Slots: %ld, Entries: %ld, Longuest chain %ld, Resize %d\n",
           hash->slots, hash->entries, hash->length, hash->resize);
}

void DestroyHashTable(htable *hash)
{
int Index;
entry *ScanItem, *Item;

   for (Index = 0; Index < hash->slots; Index++)
      for (ScanItem = hash->table[Index];
           ScanItem;
          ) {
         Item = ScanItem;
         ScanItem = ScanItem->next;
         hash->remove(Item->key, Item->data);
         FreeItem(Item);
      }
   free(hash->table);
   free(hash);
}
```

3.6 main.c

```
/*\
 * Sample use of the hash functions
 *
 * Author: Frédéric Pétrot
 * Date  : 01/09/1998
```

```
 * Credits: Thanks to the Linux hsearch (3) man page for the nice list
 * of names %-)
 *
 * This file is subject to the GNU copyleft
\*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "hash.h"

char *data[] = {
   "alpha", "bravo", "charley", "delta", "echo", "foxtrot",
   "golf", "hotel", "india", "juliette", "kilo", "lima", "mike",
   "november", "oscar", "papa", "quebec", "romeo", "sierra",
   "tango", "uniform", "victor", "whiskey", "x-ray", "yankee", "zulu",
   "lpha", "ravo", "harley", "elta", "cho", "oxtrot",
   "olf", "otel", "ndia", "uliette", "ilo", "ima", "ike",
   "ovember", "scar", "apa", "uebec", "omeo", "ierra",
   "ango", "niform", "ictor", "hiskey", "-ray", "ankee", "ulu",
   "pha", "avo", "arley", "lta", "ho", "xtrot",
   "lf", "tel", "dia", "liette", "lo", "ma", "ke",
   "vember", "car", "pa", "ebec", "meo", "erra",
   "ngo", "iform", "ctor", "iskey", "ray", "nkee", "lu",
   "ha", "vo", "rley", "ta", "o", "trot",
   "f", "el", "ia", "iette", "o", "a", "e",
   "ember", "ar", "a", "bec", "eo", "rra",
   "go", "form", "tor", "skey", "ay", "kee", "u"
};


/*\
 * A simple hash function, just to try the mecanism out!
\*/

unsigned long hashcode(char *key)
{
long Code = 0;

   while (*key)
      Code =  Code << 1 ^ *key++;
   return  Code >= 0 ? Code : -Code;
}

/*\
 * Display function for the current purpose
\*/

static void display(char *s, void *d)
{
   printf("(%s %d) ", s, *(int *)d);
}

/*\
 * Remove function for the strdup allocated strings
\*/

static void delete(char *s, void *d)
{
   free(d);
}

/*\
```

```
 * Main, and all that kind of stuff
\*/

static char buffer[BUFSIZ];

int main()
{
hashtable h;
int *i;
char *s, *t;
int p;

  h = CreateHashTable(10, 4, hashcode, NULL, display, delete);

  for (p = 0; p < sizeof(data)/sizeof(data[0]); p++) {
     if (!LookupKey(h, data[p], (void **)&i)) {
        i = (int *)calloc(1, sizeof(int));
        *i = 0;
        InstallKey(h, data[p], i);
     } else
        (*i)++;
  }

#if 0
  for (p = 0; p < (sizeof(data)/sizeof(data[0])); p += random()%2) {
     RemoveKey(h, data[p]);
#endif
     RemoveKey(h, "-ray");
     RemoveKey(h, "x-ray");
     RemoveKey(h, "victor");
#if 0
  }
#endif

  for (p = 0; p < (sizeof(data)/sizeof(data[0])); p++) {
     if (!LookupKey(h, data[p], (void **)&i))
        printf("Key %s not found\n", data[p]);
     else
        printf("Key %s has value %d\n", data[p], *i);
  }

  DumpHashTable(h);

  DestroyHashTable(h);
  return 0;
}
```

Chapitre 4

Lex: de l'extraction des lexèmes

4.1 Reconnaissance des unités syntaxiques

La compilation d'un langage passe par deux phases distinctes : l'analyse lexicale et l'analyse syntaxique. La vocation de l'outil `lex` est de simplifier l'écriture de cette première phase.

L'analyse lexicale consiste à reconnaître dans un texte un certain nombre de motifs prédéfinis, que l'on appelle *lexèmes* ou encore *unités syntaxiques* en français, ou encore *tokens* dans la langue de Sheakspeare. Ces motifs sont soit des constantes, au sens de " des chaînes de caractères fixes ", soit des expressions rationnelles.

On va donc chercher à découper le fichier d'entrée en une suite de *tokens*

L'idée est d'effectuer des actions, écrites en **C**, chaque fois que l'on a reconnu dans le texte un motif. Ces actions consistent essentiellement à " caractériser " le motif, et à renvoyer cette information à l'analyseur lexical.

D'un point de vue pratique, la sortie de `lex` est un fichier en language **C** qu'il convient de compiler et de *linker* à l'application.

L'utilisateur ne connaît que quelques points d'entrée de ce fichier : le nom de la fonction d'analyse lexicale, `int yylex()` qui renvoie un *token*, le fichier dans lequel est lu le texte d'entrée, `FILE *yyin` qui est positionné par défaut à `stdin`, un *buffer* qui contient la valeur associée au *token*, `char yytext[]`, et sa longueur `int yyleng`.

L'utilisateur appelle `yylex()` autant de fois que nécessaire pour " vider " le fichier d'entrée : si la lecture de l'entrée, `yyin`, s'arrête parce qu'un `return` a eu lieu dans une des actions, le prochain appel à `yylex()` va redémarrer d'ou la fonction s'était arrétée précédement dans le fichier.

La principale utilisation de `lex` est liée à `yacc`, l'analyseur syntaxique. C'est lors de l'étude de `yacc` que nous verrons comment ces outils communiquent.

4.2 Fichier d'entrée de `lex`

Le fichier d'entrée de `lex` est structuré en trois parties, séparées par une ligne ne contenant que `%%`.

La première partie constitue les *déclarations*. Elle peut éventuellement contenir du texte entre `%{` et `%}` (ces caractères doivent être sur des lignes par eux même), qui permet de déclarer des variables qui seront utlisées ensuite, où d'inclure des fichiers d'entête, *etc.* Par exemple :

```
%{
#include <stdio.h>
#include "y.tab.h"
static int lineno;
%}
```

Ce code est recopié tel quel dans le fichier **C** généré par `lex`. Ensuite viennent les définitions de macros permettant de simplifier l'écriture des règles et les déclarations des *conditions.*

Les définitions de macros sont de la forme *nom définition.* Le nom est un mot commencant par une lettre ou un *underscore* suivi par zéro ou plus lettres, *underscores*, chiffres où tiret. La définition commence au premier caractère non-blanc qui suit le nom, et s'achève à la fin de la ligne. On peut ensuite faire référence à l'expression rationnelle ainsi décrite grâce à `{nom}` qui sera expansée en `(définition)`, où les parenthèses servent à garantir que la priorité des opérateurs est respectée. Par exemple si l'on défini `DIGIT [0-9]`, on peut après décrire une règle du style `{DIGIT}+"."{DIGIT}*`, qui sera comprise comme `([0-9])+"."([0-9])*`.

La seconde partie défini les *règles*. Cette partie est constituée de lignes du genre *motif action*, ou le motif doit commencer la ligne sans blancs, et l'action doit commencer impérativement sur la même ligne. Les actions sont des morceaux de code écrit en **C**. Les premières lignes de cette partie peuvent être des lignes de code **C** *qui ne commencent pas dans la première colonne*. Ce lignes de codes seront recopiées au début de la fonction générée par `lex`, et permettent notamment de déclarer des variables locales.

Finallement, la dernière partie est disponible pour l'utilisateur, et elle est copiée telle quelle dans le fichier généré.

4.3 Expression rationnelles

Les expressions rationnelles permettant de décrire les motifs dans `lex` sont présentées cidessous. Les motifs à gauche reconnaîssent les expressions indiquées à droite :

`a`
: le caractère `'a'` en tant que tel ;

`.`
: n'importe quel caractère hormis le retour charriot `('\n')` ;

`[abc]`
: soit le caractère `'a'`, soit le caractère `'b'`, soit le caractère `'c'`. Les `[]` définissent une *classe de caractères*.
On peut définir des classes contenant des intervalles, comme par exemple `[a-z]` qui est la classe des lettres minuscules, ou encore `[a-zA-Z0-9]`, qui est la classe des lettres ou des chiffres.
On peut aussi définir des classes complémentaires, comme `[^a-zA-Z]` qui indique tous les caractères sauf les lettres ;

`\x`
: si `x` est la lettre `'a'`, `'b'`, `'f'`, `'n'`, `'r'`, `'t'` ou `'v'`, alors `\x` correspond à l'interprétation **C**-ANSI du caractère. Si `x` n'est pas une de ces lettres, alors le caractère `'x'` en tant que tel. Par exemple, nous avons vu que `'.'` a un sens particulier dans l'écriture des expressions rationnelles, on est donc obligé d'utiliser `\.` lorsque l'on chercher à reconnaître une `'.'` dans un texte. Si `x` est de la forme `\147`, alors c'est que l'on recherche le caractère de code octal 147. Si `x` est de la forme `\xF3`, alors c'est que l'on recherche le caractère de code hexdécimal F3.

`"[a-z]"`
: la chaîne de caractères `"[a-z]"`. En effet, les guillements ôtent aux opérateurs des expressions rationnelles leur sens particulier. On peut préciser que l'on désire reconnaître un guillement en utilisant `\"` C'est équivalent à utiliser plusieurs `'\'`, et simplifie donc l'écriture. On peut également utiliser les apostrophes pour le même usage ;

`r*`
: ou `r` est une expression rationnelle construite à l'aide des opérateurs ici décrit, recherche zéro ou plus occurence de `r`. La recherche d'une lettre suivie de zéro ou plusieurs chiffres s'écrirait ainsi : `[a-zA-Z][0-9]*`

`r+`
: recherche une ou plus occurence de `r`. Un nombre se recherche typiquement par `[0-9]+` ;

`r?`
: recherche zéro ou une occurence de `r`.

`r{n,m}`
: ou `n` et `m` sont des entiers, rechercher l'expression rationnelle répétée de `n` à `m` fois. Par exemple `[01]{32}` reconnaît des nombres binaires sur 32 bits, `[01]{16,32}` recherche des nombres binaires sur 16, 17, ..., 32 bits, et `[01]{8,}` recherche les nombres de 8 bits et plus ;

`{macro}`
: est l'expansion en expression rationnelle de la macro de nom `macro` prédéfinie. En effet, lorsque l'on utilise des expressions complexes, ou que l'on utilise souvent une expression, il est intéressant de leur associer un nom symbolique facile à utiliser et à retenir. Par exemple, on peut définir `NOMBRE [0-9]+` et subséquemment faire référence à `{NOMBRE}` ;

`(r)`
: recherche la même chose que `r`, les parenthèses étant utilisées pour modifier les règles de priorités ;

`rs`
: l'expression rationnelle `r` suivie de l'expression rationnelle `s`. On donne le nom de concaténation à cet opérateur (noté multiplicativement) ;

`r|s`
: soit ce qui est reconnu par `r`, soit ce qui est reconnu par `s` ;

`r/s`
: ce qui est reconnu par `r` si et seulement si `r` est suivi de `s`. `s` ne fait pas partie de ce qui est reconnu dans le texte ;

`^r`
: reconnaît l'expression `r` si et seulement si elle est placée en début de ligne ;

`r$`
: reconnaît l'expression `r` si et seulement si elle est placée en fin de ligne ;

`<c>r`
: reconnaît l'expression `r` si et seulement si `lex` est dans la “ condition ” `c`. Nous discuterons des conditions plus en détail après.

Attention, dans la défnition d'une classe de caractère, toutes les opérateurs d'expressions rationnelles perdent leur signification, excepté `'\'`, `'-'`, `']'` et `'^'` si ce dernier apparaît comme premier caractère de la classe.

Une règle de type `r/s` ne peut pas être telle que `s` soit exprimée comme `p/q`.

4.4 Analyse du texte d'entrée

Lorsque le programme de reconnaîssance qui a été généré à partir des spécifications d'expressions rationnelles est exécuté, il va analyser son entrée (prise en général dans un fichier texte), afin de rechercher les chaînes de caractères qui correspondent à une ou plusieurs expressions rationnelles.

S'il n'y a qu'une expression qui reconnaît la chaîne en entrée, alors c'est cette expression qui est retenue.

S'il y a plusieurs possibilités, alors c'est celle qui reconnaît le plus de texte vis-à-vis de l'entrée qui est retenue. Notons que dans le cas d'une expression `r/s`, la longueur du texte reconnue par `s` est prise en compte, bine que réintroduite dans l'entrée. Si plusieurs expressions reconnaîssent la même quantité de texte, alors c'est l'ordre de déclaration dans le fichier `lex` qui impose la priorité : les règles apparaîssant au début du fichier sont plus prioritaires que celles qui arrivent après.

Une fois que la “ bonne ” règle est retenue, le texte correspondant à l'expression rationnelle, (le motif en un mot), est disponible par la variable globale `char yytext[YYLMAX]`, et sa longueur dans `int yyleng`. L'action associée à la règle est exécutée, et ce qui reste du texte est réintroduit en entrée du programme.

Un motif décrit par une expression rationnelle va reconnaître un ensemble d'élements de l'entrée, comme par exemple des nombres. Or ces nombres, par définition peuvent prendre une infinité de valeurs différentes. Il faut donc distinguer deux choses : le motif (ou la classe de caractère) reconnu, ici le fait que ce soit un nombre, et la valeur associée, 3, 12 ou 37549. La classe de caractère est identifiée par un nom symbolique que l'on appelle *token*, et la valeur lue dans le texte est la valeur associée (et non du) *token*.

L'analyseur n'effectue qu'une action par *token* reconnu, celle de la règle retenue.

4.5 Lecture supplémentaire, rejection, ...

Il est possible de modifier dynamiquement le fichier d'entrée de `lex`, grâce à `input()` qui lit le prochain caractère de l'entrée en court-circuitant le mécanimsme normal de `lex`, et `unput()` qui réintroduit dans le flux d'entrée un caractère sous le nez de `lex`.

Par exemple, lisons un commentaire **C** :

```
"/*"  { /* l'action doit commencer sur la meme ligne! */
      register int c;

         while(1) {
            while ((c = input()) != '*' && c != EOF);
            if (c == EOF) {
               fprintf(stderr, "Fin de fichier dans un commentaire!\n");
               break;
            }
            while ((c = input()) == '*');
            if (c == '/')
               break; /* fin du commentaire */
         }
      }
```

Maintenant, lisons un texte en minuscules et reintroduisons le dans le flux d'entrée en majuscules :

```
[a-z]+   {
            int i;

            for (i = 1; i <= yyleng; i++)
               unput(toupper(yytext[yyleng - i]));
         }
```

Notons que la texte doit être introduit dans l'ordre inverse du sens naturel, car `unput()` ajoute en tête de texte.

Il est possible de modifier le mécanisme de lecture de `lex` grâce à `REJECT`. Cette action demande à `lex` de considérer le " meilleur motif suivant " pour l'analyse, tout en ayant déjà effectuer les actions du meilleur motif. On peut enchaîner les `REJECT`. Par exemple, comptons le nombre de mots d'un texte et le nombre de fois que `fred` apparaît dedans :

```
   int wc, fc; /* Nb de mots, nb de fred */
%%
fred      {fc++; REJECT;}
[^ \t\n]+ {wc++;}
```

Certes c'est un exemple trivial ou l'on aurait pu faire le `wc++` à la place de `REJECT`, mais il illustre bien le mécanisme.

On peut conserver le texte précédemment reconnu et ajouter celui qui correspond au prochain motif dans `yytext` grâce à la fonction `yymore()`. Cette fonction indique qu'il ne faut pas remplacer le contenu de `yytext` par ce qui a été reconnu dans le fichier d'entrée, mais l'ajouter à la fin du précedent texte. Exemple :

```
%%
allo         {yymore();}
" la terre"  {printf("%s\n", yytext);}
```

n'affichera rien après que l'on lu `allo`, puis affichera `allo la terre` une fois la seconde partie de la phase lue.

L'utilisation de ces mécanismes (essentiellement le `REJECT`) peut avoir un impact considérable sur les performances de l'analyseur. On tentera donc de s'en passer lorsque c'est possible.

4.6 Conditions

`Lex` propose un mécanisme qui permet d'activer des règles de façon conditionnelle. Toute règle dont le motif est préfixé par <COND> ne sera active, c.-à-d. interviendra dans l'analyse du texte d'entrée, que si l'analyseur se trouve dans la condition `COND`. Par convention, le nom des conditions est écrit en lettres majuscules. Par exemple :

```
<CHAINE>[^"]* {/* Lit le contenu d'une chaine de caractères */}
```

On peut spécifier plusieurs conditions avec `<c1,c2,c3>`.

Les conditions sont déclarées dans la première partie du fichier, grâce à un `%s` en début de ligne.

On rentre dans une condition à l'aide de la construction `BEGIN COND`, et une condition par défaut existe permettant de revenir dans l'état initial : `BEGIN INTIAL`. Lorsqu'une condition est activée, l'ensemble des règles sans condition reste partie intégrale de l'analyseur : il regarde les règles incluant la condition et les règles sans condition avant de prendre une décision[1].

En réalité, les noms des conditions sont des `#define` du **C**, on peut donc les affecter dans des variables de type `int`. Illustrons ces mécanismes sur un petit cas d'école.

Prenons l'exemple d'un analyseur qui lit les commentaires dans un fichier **C** tout en incrémentant correctement le nombre de lignes lues, sachant que le commentaire peut être détecté soit dans le cas `INITIAL`, soit dans le cas `AUTRE`.

```
%{
int lineno = 1; /* variable globale */
%}
.....
%s CMT
%%
                   int condition; /* Pas de {} autour! */
"/*"               {condition = INITIAL; BEGIN CMT;}
<AUTRE>"/*"        {condition = AUTRE; BEGIN CMT;}
<CMT>[^*\n]*       /* absorbe tout sauf '*' et '\n' */
<CMT>"*"+[^*/\n]*  /* absorbe les '*' non suivies de '/'s */
<CMT>\n            {lineno++;}      /* tout ca pour ca! */
<CMT>"*"+"/"       {BEGIN condition;}
....
```

Il y a bien sur d'autres manières de faire la même chose, mais nous avons ici tenté de lire autant de texte que possible à chaque fois, ce qui nous donne de meilleurs performances.

4.7 Principes d'utilisation

`Lex` prend en entrée un fichier de description et génère en sortie un fichier **C** qui s'appelle toujours `lex.yy.c`. Une librairie permet de faire très rapidement de petits *scanners* (analyseurs lexicaux de fichiers) car elle contient un `main` standard et la fonction `int yywrap()` qui doit retourner 1 lorsque l'on arrive en fin de fichier.

```
> ls
scanner.lex
> lex scanner.l
> ls
scanner.lex      lex.yy.c
> cc -o scan lex.yy.c -ll
> scan < file
.....
```

1. L'outil `flex` permet de déclarer des conditions exclusives, `%x`, ou seules les règles précédées par une condition sont exécutées.

Chapitre 5

Yacc: outil d'aide à l'analyse des grammaires

5.1 Introduction

L'analyse grammaticale est une activité des plus courantes dans le monde de l'informatique, car beaucoup d'outils prennent en entrée des fichiers. Les grammaires peuvent être simples, comme dans le cas d'un assembleur par exemple, une instruction par ligne, ou très complexes et récursives, comme dans des expressions du langage **C**.

L'outil `yacc`[**?**], et l'implémentation particulière que nous utilisons, `bison`, est capable d'analyser une classe de grammaire dite *LALR(1)*, c'est-à-dire *Look Ahead Left to right scanning of the input constructing a Rightmost derivation in reverse with 1 look ahead token*, en bref toutes les grammaires que nous serons amener à rencontrer. Pour ceux qui aiment théoriser, la lecture de [**?**] est une obligation.

Dans la pratique, `yacc` peut être utilisé pour analyser les langages les plus courants, **C**, **pascal**, **fortran**, *etc*, sans connaître la théorie de la méthode (ça ne fait pas de mal, mais on peut s'en passer).

Il faut dès à présent noter que l'analyse des grammaires simples est beaucoup plus efficace lorsqu'écrite en **C**, et n'est pas pour autant plus difficile à implémenter ainsi. On préférera donc reserver `yacc` à des tâches plus ardues, l'analyse de grammaires récursives, très compliquée à faire *à la main*.

5.2 Écriture des règles

Les deux constituants de l'analyse syntaxique sont les éléments terminaux, *tokens*, et les règles d'association, dites élements non-terminaux ou encore *grouping*.

Dans le cas du langage **C** par exemple, les *tokens* sont les identificateurs, les constantes, les mots clefs, *etc.* Ainsi la décomposition d'une fonction en *tokens* est:

`int`	mot-clef int
`carré(x)`	identificateur '(' identificateur ')'
`int x;`	mot-clef int identificateur ';'
`{`	'{'
`return x * x;`	mot-clef return identificateur '*' identificateur
`}`	'}'

La manière dont la syntaxe est définie en **C** inclue les expressions, les déclarations, les définitions de fonctions. À Chaque non-terminal est associé une règle qui permet de dériver son analyse de règles plus simples, qui s'achèvent sur des *tokens*.

Ainsi, une expression telle que `x * x` est exprimée formellement comme:

> Une expression est consituée d'une expression, de l'opérateur multiplier, et d'une expression.

`yacc` lit une séquence de *tokens* et les regroupes en utilisant les règles de grammaire. Si la séquence est valide, alors elle est reduite dans le non-terminal qui la représente. Finalement, la totalité de l'entrée est réduite dans le symbole de départ. Par défaut, ce symbole est le premier non-terminal qui apparait dans le fichier d'entrée. La clause `%start` permet d'en indiquer un autre.

Les règles se répartissent grossièrement en deux catégories :

– les règles linéaires. ce sont des règles qui sont définies en fonctions d'autres règles et/ou de *tokens*. Ces régles peuvent avoir une ou plusieurs alternatives.
– les règles récurrentes. ce sont des règles qui sont définies en fonctions d'elles mêmes, d'autres règles et/ou de *tokens*. Comme toute récurrence, il doit exister une alternative assurant la terminaison de la récurrence : ces règles ont donc au moins deux alternatives. Les règles peuvent aussi être récurrentes à travers d'autres règles, mais on se ramène alors au cas précécent.

J'illustre ci-dessous des 2 types de règles, ainsi que les

5.3 Quelle est l'entrée et la sortie de `yacc`?

Les actions de `yacc` sont basées sur un fichier source, suffixé usuellement par `.y`, qui défini les règles d'analyse syntaxique et les actions qui y sont associés. La sortie de `yacc` est un fichier de nom `y.tab.c` dans le `yacc` usuel, et *file*`.tab.c` sous `bison`. Un comportement équivalent à `yacc` est obtenu avec `bison -y`.

Si l'analyseur lexical, (voir le document *lex : un programme pour la production d'analyseurs lexicaux*), n'est pas dans le même fichier que la sortie de `yacc`, il faut faire en sorte de partager les *tokens*. `yacc` les fournis dans le fichier `y.tab.h`

5.4 Organisation des fichiers d'entrée de `yacc`

Les spécifications d'entrée de `yacc` ont la forme suivante:

%{
Déclarations **C**
%}
Déclarations pour yacc
%%
Règles de grammaire
%%
Code **C** *éventuel*

Les déclarations **C** permet de définir des macros, des variables, d'inclure des fichier, *etc.*

Les déclarations pour `yacc` definissent les noms des symboles terminaux et non-terminaux, ainsi que leur type, leur précédence,

Les règles de grammaire indiquent la façon de construire un non-terminal à partir de ses éléments, terminaux ou non. Il doit toujours y avoir au moins une règle de grammaire.

Le code **C** éventuel contient n'importe quel type de code **C**. Dans les programmes les plus simples, l'analyseur lexical et les fonctions associées aux règles de grammaire peuvent être définies là. Cette section est copiée telle quelle à la fin de l'analyseur généré.

Nous allons voir dans la suite la manière dont sont éxprimées les règles syntaxique et comment leurs sont associées les valeurs sémantique. Mais avant d'en arriver là, nous allons voir comment il faut procéder pour générer les analyseurs.

5.5 Comment utiliser un fichier `yacc`

L'outil `yacc` est un générateur d'analyseur syntaxique. Sa sortie est un fichier **C** qu'il faut compiler, puis linker.

`yacc` utilise des variables globales et des fonctions préfixées par `yy` ou `YY`, il faut donc éviter d'utiliser un tel préfixe pour les fonctions utilisateur.

La production d'un objet à *linker* à partir d'un source `yacc` se fera typiquement ainsi:

```
y.tab.o y.tab.h : grammar.yac
        $(YACC) $(YFLAGS) -d grammar.yac; \
        $(SED) -e "s/yy/grammar/g" -e "S/YY/GRAMMAR/g" y.tab.c > 1; \
        $(MV) 1 y.tab.c; \
        $(SED) -e "s/yy/grammar/g" -e "S/YY/GRAMMAR/g" y.tab.h > 1; \
        $(MV) 1 y.tab.h
        $(CC) $(CFLAGS) -c y.tab.c
```

L'interface avec le langage **C** est définie avec précision, et ce qu'il en est dit ici est *a priori* valable pour toutes les implémentation de `yacc`.

Le résultat de `yacc` est en fait une fonction **C** qui par convention s'appelle `yyparse()`. C'est l'appel à cette fonction qui entraîne l'analyse grammaticale d'un fichier. Cette fonction se chager de lire les *tokens*, d'exécuter les actions, et finalement de rendre la main lors de la fin de fichier ou d'échouer lors d'erreurs de syntaxes. `yyparse()` renvoie `0` si tout c'est bien passé, `1` si il y a des erreurs de syntaxe. Lors de l'exécution d'une action, `YYACCEPT` stoppe la lecture et renvoie `0`, et `YYABORT` stoppe la lecture et renvoie `1`.

Afin de pouvoir lire les *tokens* un à un, `yacc` utilise la fonction `yylex` qui à chaque appel retourne la valeur du *token* suivant dans le fichier lu. Par exemple:

```
yylex()
{
   ...
   if (isspace(c))
      continue;
   ...
   if (c == '+' || c == '-' || c == '*' || c == '\\')
      return c;
   ...
   return ID;
}
```

Suivant la complexité de l'analyse, cette fonction, qui est du ressort de l'utilisateur, peut être écrite en **C**, ou générée grâce a `lex`.

Si les *tokens* sont nécessaire à l'analyse grammaticale, leur valeur sémantique est nécessaire à l'exploitation dans les actions. Ces valeurs peuvent être de différents types, *int*, `double`, ou même des structures. Pour les déclarer, on utilise:

```
%union {
   int       i;
   double    d;
   char    **s;
}
```

Et il faut fournir cette information, lorsqu'elle est pertinente, dans cette union à l'interieur de `yylex()`. La variable `yylval` définie du type de la pile de `yacc`, `YYSTYPE`, est prévue à cet effet.

```
yylex()
{
   ...
   yylval.i= trois;
   return INT;
   ...
}
```

La fonction `yyerror` est utilisée par `yacc` pour emmettre les messages d'erreur. Elle est définie par l'utilisateur sous la forme, minimale ici:

```
yyerror(s)
char *s;
{
   (void)fprintf(stderr, "%s\n", s);
}
```

Il ne faut pas mettre d'`exit()` en fin de `yyerror()` si l'on désire faire de la récupération dérreurs, ce qui est souhaitable pour les langages complexes.

5.6 Un exemple de fichier yacc

```
%{
#include <stdio.h>
%}
%token NUM
%left AND NAND OR NOR
%left NOT
%start input
%%
input : /* empty */
      | input line
      ;

line  : bexpr
      { printf("%d\n", $1);}
      ;
bexpr : NUM
      { $$ = $1;}
      | bexpr AND bexpr
      { $$ = $1 & $3;}
      | bexpr OR bexpr
      { $$ = $1 | $3;}
      | NOT bexpr
      { $$ = !$2;}
      | '(' bexpr ')'
      { $$ = $2;}
      ;
%%
main()
{
   return yyparse();
}

yylex()
{
static int c;
static char key[20];
int i = 0;

   while (isspace(c = getchar()));

   if (c == EOF)
      return 0;

   if (isalpha(c)) {
      do {
         key[i++] = (char)c;
      } while (isalpha((c = getchar())));
      key[i] = '\0';
      if (!strcmp(key, "and"))
         return AND;
      if (!strcmp(key, "or"))
         return OR;
      if (!strcmp(key, "not"))
         return NOT;
      (void)puts("Unknown operator");
      exit(12);
   }
   switch (c) {
   case '0' :
   case '1' :
```

```
      yylval = c;
      return NUM;
   case '(' :
   case ')' :
      return c;
   }
   (void)puts("Unknown character sequence");
   exit(11);
}

yyerror(s)
char *s;
{
   puts(s);
}
```

Cet exemple simple illustre à lui seul beaucoup d'aspects de l'écriture de grammaire pour `yacc`, ainsi que la manière d'utiliser un *parser* généré avec du code **C**.

La première partie du fichier, entre `%{ %}`, contient des définitions **C**, typiquement des `includes`, des déclarations de structures, des prototypes de fonctions, *etc.*

La seconde, entre `%} %%`, contient les déclarations `yacc`. Ces déclarations sont des types:

`%token`
: qui défini une classe d'appartenance à un *token*. Par exemple, `0` et `1` appartiennent à la classe `NUM`.

`%right`

`%left` definissent l'ordre d'associativité des *tokens*. Dans le cas présent, `x and y and z` est interprété comme `(x and y) and z`, car on associe d'abord les *tokens* de gauche. Leur présence est indispensable au bon déroulement de la compilation du *parser*.

`%nonassoc` indique que l'on doit faire une erreur de syntaxe lorsque l'on trouve deux fois le même opérateur à la file. L'ordre de déclaration des *tokens* défini la priorité, de la plus basse vers la plus haute, de l'associativité par défaut.

`%type` permet de définir le type des symboles non terminaux dans le cas d'utilisation de la clause `%union`. En effet, `yacc` ne peut pas connaître le type à appliquer à `$$` dans ce cas. La syntaxe de la définition est `%type <`*`type`*`>` *non-terminal*

`%start` précise la règle qui est l'entrée de la grammaire. Dans le cas précis, si l'on ne voulait analyser qu'une ligne, alors il suffirait de faire `%start line`.

Ensuite vient la définition des règles d'analyse proprement dites. La partie gauche défini le non-terminal, et la partie droite les règles d'association. Les règles sont généralement récursives, et dépendent soit de terminaux, soit de non-terminaux.

```
input : /* empty */
      | input line
      ;
```

Lorsqu'une règle

5.7 Les variables accessibles dans `yacc`

L'utilisation des valeurs fournies par `yylex` nécessite une syntaxe particulière, et estrangifiante.

`$$` est la "variable" qui contient la valeur associée à la règle courante.

`$n` est la "variable" qui contient la valeur de la nième composante de la règle courante.

`YYBACKUP(token, value)` cette action remet un *token* sur la pile. Elle n'est acceptable que pour les règles à une réduction, et sans *tokens* de prévision.

`YYRECOVERING` est une expression qui vaut `1` si l'on est en mode de récupération d'erreur, `0` sinon.

`yychar` contient la valeur du *token* de prévision courant.

`yyclearin` efface le *token* de prévision courant. C'est utile dans les régles d'erreurs.

`yyerrok` recommence à générer des messages d'erreurs dès la fin du recouvrement d'erreur.

`yylval` contient la valeur sémantique présente sur la pile. Le type de cette variable est lié à la déclaration `%union`.

5.8 RCS: outil de gestion des révisions de programmes

5.9 Introduction

Le système *RCS*, *Revision Control System*, gère les multiples versions des fichiers. *RCS* automatise l'archivage, la récupération, le commentaire des changements, l'identification des auteurs et la fusion des révisions. *RCS* est utile pour les textes qui sont fréquement modifiés, tel que les programmes, la documentation, les graphiques, ...

Les fonctions de *RCS* sont :

- Archiver et recharger les multiples versions d'un texte. Les anciennes versions sont sauvegarder de façon à utiliser peu de place sur le disque. Les changements ne détruisent plus les originaux, et les anciennes versions restent accessibles. Les versions peuvent être récupérées en fonction d'interval de numéro de révisions, de noms symboliques, d'auteurs, de dates, et d'états.
- Gérer un historique[1] complet des changements. *RCS* prend note automatiquement des modifications. En sus des modifications elles-même, *RCS* conserve leur auteur, la date de la prise en compte des changements (*check-in*), et un court message d'information résumant les modifications. Ces informations permettent aisément de connaître les modifications apportées à un fichier sans avoir à comparer les sources. La prise en compte des changements demande ces informations, ce qui permet de palier partiellement l'absence de commentaires dans le code, or d'aller traquer un collègue.
- Résoudre les conflits d'accès à une ressource partagée. Lorsque plusieurs personnes veulent modifier le même fichier, *RCS* les en informe et évite ainsi au travail de l'un de mettre en péril le travail de l'autre.
- Maintenir un “ arbre ”de révisions. Chaque module peut évoluer indépendamment des autres.
- Fusionner les révisions, et résoudre les conflits. Des développements séparés dans un même fichier peuvent être parfaitement légaux. Néanmoins, si *RCS* détecte des changements qui se recouvrent, alors il en averti l'utilisateur.
- Créer des version du projet diffusable et de configuration. Les révisions peuvent être typées comme “ diffusée ”, “ stable ”, “ expérimentale ”, etc. Ainsi des ensembles de modules peuvent être liés simplement pour faire un projet stable.
- Minimiser la place requise pour la sauvegarde des modifications. Seules les différences sont conservées, ce qui permet à un projet de vivre plusieurs années sans grossir trop.

5.10 Brève présentation

Le système *RCS* est composé d'un ensemble d'outils qui sont appelés indépendamment les uns des autres. À chaque outil correspond une fonctionnalité bien précise.

`ci` fait le *check-in* d'un fichier, c.-à-d. qu'il sauvegarde une nouvelle révision dans un fichier *RCS* existant, ou qu'il le crée si c'est le premier `ci`.

`co` fait le *check-out*, c.-à-d. qu'il récupère la dernière version archivée dans le fichier *RCS*. Un `co` ne peut avoir lieu qu'après un `ci`.

`rcsmerge` fusionne les modifications existant entre deux révisions d'un fichier *RCS* dans le fichier de travail correspondant.

`rcsfreeze` donne un nom symbolique à un ensemble de fichiers *RCS* qui constituent une version diffusable.

1. Exposé chronologique.

`rcsdiff` affiche les différences entre le fichier de travail et la dernière version qui a été archivée grâce à *RCS*.

`rlog` donne l'ensemble des informations ne concernant pas le source lui même d'un les fichiers *RCS*.

`ident` affiche " un certain nombre d'informations " contenue dans un exécutable dont les sources sont maintenues avec *RCS*.

`rcs` permet de configurer le mode de fonctionnement des autres outils, et de faire des opérations interdites à `ci` et `co`, notamment le relâchement de sémaphore.

5.11 Premier pas avec *RCS*

Imaginons que nous ayons un fichier `pwet.c` que nous désirons mettre sous le contrôle de *RCS* afin d'en gérer les versions succéssives. Il faut commencer par créer dans le répertoire `RCS` dans le répertoire de travail, puis d'appeller `ci` :

```
> mkdir RCS
> ci pwet.c
```

Cette commande crée un fichier *RCS* dans le répertoire `RCS`, y archive `pwet.c` en lui donnant le numéro de révision 1.1, et efface `pwet.c`. *RCS* demande aussi une description qui doit être une indication de ce qui est fait dans `pwet.c`. Les `ci` suivant ne demanderons plus qu'un résumé des changements effectués entre deux versions. Les fichiers dans le répertoire `RCS` sont dits *fichiers RCS* alors que les fichiers sources sont dits *fichiers de travail.*

Le fichier `pwet.c` ayant été effacé, il faut le récupérer pour le compiler ou l'éditer. La commande :

```
> co pwet.c
```

extrait la plus récente révision du fichier RCS dans `pwet.c`. Mais alors vous n'avez pas les droits d'écriture sur le fichier. Il faut faire :

```
> co -l pwet.c
```

pour indiquer que vous n'autoriser plus les autres à modifier ce fichier (à l'aide d'un *lock*). Maintenant, après avoir fait quelques modifications, vous voulez voir les différences entre votre fichier et sa version précédente :

```
> rcsdiff pwet.c
```

Finalement, il faut archiver la nouvelle version :

```
> ci pwet.c
```

Le numéro de révision est automatiquement incrémenté.

Si `ci` répond :

```
> ci error: no lock set by fred
```

cela signifie que vous essayer d'archiver un fichier dont vous n'avez pas demandé l'exclusivité. Bien sûr, il est trop tard pour faire un `co -l`, car cela effacerait vos modifications. En conséquence de quoi, il faut appeller :

```
> rcs -l pwet.c
```

Ceci va vous rendre propriétaire de la dernière version du fichier RCS, si et seulement si quelqu'un d'autre n'en a pas pris posséssion avant, auquel cas il vous faudra négocier avec cette personne.

5.12 Convention sur les noms de fichiers et stratégie de nommage des révisions

Les fichiers marchent par paires : à un fichier de travail correspond un fichier *RCS*. Si le fichier de travail s'appelle `pwet.c`, alors le fichier *RCS*. s'appellera `pwet.c,v`. Ce fichier est recherché dans le répertoire `./RCS` par défaut, puis dans le répertoire courant.

5.13 Attributs des fichiers RCS

La commande `rcs` permet de créer de nouveaux fichiers *RCS* ou de modifier les attributs de ceux qui existent. Un fichier *RCS* contient les multiples versions (ou révisions) du source, des informations commentant les modifications, un cours résumé du contenu du fichiers, une liste d'utilisateurs habilités à le lire et/ou le modifier, et quelques attributs spécifiques.

N'ont le droit d'acceder à un fichier grâce à `rcs` que les utilisateurs dont le nom figure dans la liste, le propriétaire, au sens Unix, du fichier, ou le super-utilisateur.

Pour créer un fichier *RCS* sans y déposer de révision, on utilisera l'option `-i`. C'est en général la personne qui gère le projet qui prend cette responsabilité, et qui affecte également, avec l'option `-a` la liste des utilisateurs admis.

```
> rcs -i -ambk,beh,abl mbk_util.c
```

Les utilisateurs *mbk*, *beh* et *abl* ont le droit de faire toutes les commandes du système sur le fichier `mbk_util.c`.

Si l'on désire ôter ses droits à un utilisateur, on emploie l'option `-e` :

```
> rcs -eabl mbk_util.c
```

Si `-e` est utilisé sans argument, l'ensemble de la liste est détruite.

Les options `-l` et `-u` permettent de vérouiller ou déverouiller une révision, (les options prennent alors comme argument le numéro de la révision), sans pour autant modifier l'état des fichiers de travail. C'est particulièrement utile en cas d'omission de vérouillage, ou lorsqu'un utilisateur a vérouillé un fichier et est parti sans relâcher son verrou.

Le mode de vérouillage est un attribut général du fichier *RCS* qui peut être mis à stricte `-L`, c.-à-d. que même le propriétaire, au sens Unix, du fichier est obligé d'indiquer qu'il vérouille (c'est le mode par défaut), ou lâche, auquel cas cette opération n'est pas exigée. Ce cas se présente lorsque l'on gère des fichiers qui ne sont pas partagés avec *RCS*.

On peut également attribuer un nom symbolique à une révision. C'est nécessaire lorsque l'on se prépare à faire une diffusion (*release*). On utilise alors l'option `-n`. Avec `-n`*nom*`:` on donne à la dernière version le nom *nom*. Attention, l'omission du `:` indique que l'on désire effacer le nom *nom* correspondant à une précédente révision! On peut spécifier une révision spéciale avec `-n`*nom*`:`*revision*. Par exemple :

```
> rcs -nRelease3.0: RCS/*
```

va associer le nom `Release 3.0` à l'ensemble des révisions les plus récentes qui ont été sauvegardées.

Une dernière option qu'il faut connaître est l'option `-c` qui spécifie, pour un fichier donnée, le ou les caractères qui définissent le commentaire pour les formats de fichiers qui ne possèdent pas de commentaires multi-lignes, comme par exemple les fichiers *shell* par exemple :

```
> rcs -c# getlog.sh
```

5.14 Archivage

La commande `ci` archive les nouvelles révisions dans les fichiers *RCS*. Le *login* de la personne qui appelle `ci` doit être dans la liste d'accès. Pour qu'un utilisateur puisse ajouter une révision dans une branche, il faut que ce soit lui qui ait vérouillé cette branche. Sinon il devra créer une autre branche.

```
> ci xx
```

`Ci` vérifie qu'il y a eu modification entre la révision qui doit être archivée et la révision courante. Si ce n'est pas le cas, alors il n'y a pas modification du fichier *RCS*. On peut vouloir forcer un changement de révision de l'archive : utiliser alors l'option `-f`.

Pour chaque révision, `ci` exige un message d'information. Son but est de résumer les changements apportés au fichier, et doit être terminé par un caractère de fin de fichier (`^D`). Si l'on applique `ci` à plusieurs fichiers :

```
> ci pwet.c pwet.h Makefile
```

`ci` demande s'il doit utiliser le même message d'information pour tous les fichiers.

Si un fichier *RCS* n'existe pas, `ci` le crée et l'archive (sous le numéro de révision 1.1). La liste d'accès est vide, et un texte spécifiant le pourquoi de l'archivage est demandé.

Le numéro de version, *rev*, peut être spécifié sur la ligne de commande par de nombreuses options. Si *rev* est un numéro de revision, il doit être plus élevé que le précédent sur cette branche, ou doit commencer une nouvelle branche.

Si *rev* est une branche plutôt 'qu'un numéro de révision, la révision immédiatement supérieur à la précédente est ajoutée à cette branche. Si *rev* indique une branche non existante, alors cette branche est créée avec le numéro de révision initiale *rev*.1. Si *rev* est omis, alors `ci` essaye de déterminer le numéro de révision qui va bien à partir du dernier vérouillage de l'utilisateur qui l'appelle. Si l'utilisateur à vérouillé la révision finale d'une branche, alors la révision est ajoutée sur cette branche en incrémentant ce numéro de révision. Si l'utilisateur a vérouillé une révision non-terminale, alors une nouvelle sous-branche est ajoutée sur cette révision. Example : la dernière révision vérouillée par l'utilisateur `fred` est 4.3.2, mais la dernière version vérouillée est 4.3.5, alors si `fred` fait un `ci`, le numéro révision sera 4.3.2.1.

Voyons maintenant les différentes options de `ci`. Par défaut, l'option `-r`*rev*`]` est employée : cela archive une révision, sous le numéro de version *rev* s'il est spécifié et légal, dévérouille la révision correspondante, et efface le fichier source de travail.

L'option `-l` a le même comportement que `-r` sauf qu'il vérouille immédiatement la révision sauvée et la rend disponible pour travailler dessus. Ceci permet d'archiver une révision tout en continuant à travailler dessus.

L'option `-u` a le même effet que `-l` sauf que le fichier correspondant à la révision qui vient d'être archivée n'est pas vérouillée. Ceci implique que le fichier de travail n'est disponible que pour être compilé ou consulté, mais en aucun cas modifié immédiatement.

Il est possible d'imposer l'archivage d'une révision même si aucun changement n'a été fait avec l'option `-f`. Cela peut s'avérer utile dans le cas ou l'on désire maintenir un ensemble de fichier avec le même numéro de révision.

Donner un nom symbolique à une révision se fait grâce à `-n` ou `-N`, la première utilisation échoue si le nom passé en paramètre a déjà été usité, l'autre remplace purement et simplement un nom préexistant.

L'état de la révision est imposable grâce à l'option `-s`, qui prend comme argument classique `Exp` pour expérimentale, `Stab` pour stable, et `Rel` pour diffusée.

5.15 Récupération

`Co` récupère une révision dans un fichier *RCS* et le met dans un fichier de travail. La récupération peut se faire avec ou sans vérouillage, sachant que le vérouillage n'est utile que si l'on cherche à modifier le fichier. Si l'on veut vérouiller le fichier, il faut être dans la liste des utilisateurs légaux du fichier. A contrario, tout le monde a le droit de récupérer sans vérouiller. Une révision est choisie en fonction de son numéro de révision ou de branche, date/temps d'archivage, auteur ou état. Quand ces options sont utilisées en combinaison, `co` extrait la plus récente qui les satisfait toutes.

Si aucune option n'est spécifiée, alors c'est la dernière version qui est récupérée.

Les options les plus utiles sont `-r` qui récupère la version dont le numéro est inférieur ou égal à son argument. On peut vérouiller à l'aide de `-l`, et dévérouiller avec `-u`.

Des mots-clefs sont insérés dans le fichier, et leur génération est ajustable. Par défaut, `-kkv`, produit la forme par défaut : `$Revision$` deviendra `$Revision: 5.7 $`. Le nom de l'utilisateur n'apparaît que s'il y a vérouillage. Si l'on veut forcer un nom, il faut utiliser `-kkvl`. L'option `-kk` *a contrario* ne fait pas de subsitution, ce qui permet de faire des différences qui ne portent que sur le source et non sur ce qui est utile à la gestion *RCS* du fichier.

5.16 Mots-clefs

Les chaînes de la forme `$Keyword$` ou `$Keyword : ...$` qui apparaîssent n'importe ou dans du texte sont remplacées par des chaînes de la forme `$Keyword : valeur$`. Ces chaînes peuvent être mises dans des variables statiques inusités ou dans des commentaires. C'est `co` qui met à jour le contenu de ces chaînes.

Les mots-clefs sont

`$Author$` le nom de *login* de la personne qui a archivé la révision ;

`$Date$` la date et l'heure de l'archivage ;

`$Header$` un entête standard contenant le chemin d'accès du fichier *RCS*, le numéro de révision, la date, l'auteur, l'état et le " vérouilleur " ;

`$Id$` identique à `$Header$` hormis que le chemin d'accès du fichier *RCS* est omis ;

`$Locker$` le nom de *login* du " vérouilleur ", vide si le fichier n'est pas vérouillé ;

`$Log$` le message de *log* qui est fourni par l'utilisateur lors de l'archivage, précédé d'un entête standard. Les *logs* sont accumulés, ce qui permet de maintenir un historique des changements dans le fichier le plus récent;

`$RCSfile$` Le nom du fichier sans chemin ;

`$Revision$` Le numéro donné à la révision courante ;

`$Source$` Le nom du fichier avec son chemin d'accès complet ;

`$State$` L'état du fichier tel qu'indiqué avec l'option `-s` de `rcs` ou `ci`.

5.17 Positionnement des attributs des fichiers *RCS*

Quand j'aurais le temps, désolé, ...

5.18 Les autres utilitaires

Pas mieux.