Interprétation et compilation
Licence Informatique troisième année

Jean Méhat
jm@univ-paris8.fr
Université de Paris 8 Vincennes Saint Denis

5 avril 2013

Table des matières

Chapitre 1

Introduction

Ce chapitre présente brièvement le sujet principal du cours (les compilateurs), avec leur structure ordinaire à laquelle nous ferons référence dans la suite du cours, puis expose l'organisation de ce support.

1.1 Pré-requis pour suivre le cours

Le cours suppose que l'étudiant est familier de la programmation et de l'environnement Unix. La plupart des exemples en langage de haut niveau sont pris dans le langage C, dans lequel l'étudiant est supposé pouvoir programmer.

Il n'y a pas de rappel sur le hash-coding (voir le cours sur les structures de données et les algorithmes.).

1.2 L'intérêt de l'étude de la compilation

Ça permet de voir en détail le fonctionnement d'un langage de programmation, y compris dans des aspects obscurs et peu étudiés.

Pour écrire du bon code, il faut avoir une idée de la façon dont il est traduit. L'étude de la compilation le permet.

Ça nous amène à étudier des algorithmes d'intérêt général et des objets abstraits comme les automates finis et les expressions régulières qui sont utiles dans d'autres contextes.

Ça nous conduit à apprendre à nous servir d'outils (comme Yacc et Bison, des générateurs d'analyseurs syntaxiques) qui sont utiles dans de nombreux contextes.

Ça nous donne un bon prétexte pour regarder un peu la programmation en assembleur.

1.3 Ce qu'est un compilateur

Un compilateur est un *programme* qui est chargé de *traduire* un *programme* écrit dans un langage dans un autre langage. Le langage du programme de départ est appelé le langage *source*, le langage du programme résultat le langage *cible* (*source language* et *target language* en anglais).

Le plus souvent, le langage source est un langage dit de *haut niveau*, avec des structures de contrôle et de données complexes alors que le langage cible est du langage machine, directement exécutable par un processeur. Il y a cependant des exceptions ; par exemple, certains compilateurs traduisent des programmes d'un langage de haut niveau vers un autre.

1.3.1 Compilateurs *versus* interprètes

Un interprète est un *programme* qui est chargé d'*exécuter* un programme écrit dans un langage sur un processeur qui exécute un autre langage. Il joue presque le même rôle qu'un compilateur, mais présente des caractères différents. Alors qu'un compilateur effectue un travail équivalent à celui d'un traducteur humain qui traduit un ouvrage d'une langue dans une autre, celui d'un interprète évoque plus d'un *traduction simultanée*, qui produit la traduction à mesure que le discours est tenu.

En général, un interprète est beaucoup plus facile à réaliser qu'un compilateur.

Un compilateur traduit le programme une fois pour toutes : le résultat est un programme dans le langage cible, qu'on peut ensuite exécuter un nombre indéfini de fois. En revanche, un interprète doit traduire chaque élément du langage source à chaque fois qu'il faut l'exécuter (mais seulement quand il faut l'exécuter : pas besoin de traduire ce qui ne sert pas). En terme de temps, la compilation est une opération *beaucoup* plus lente que le chargement d'un programme dans un interprète, mais l'exécution d'un programme compilé est *beaucoup* plus rapide que celle d'un programme interprété.

Le compilateur ne peut détecter des erreurs dans le programme qu'à partir d'une analyse *statique* : en examinant le programme sans le faire tourner. Un interprète pourra aussi détecter des erreurs de façon *dynamique*, lors de l'exécution du programme.

Certains langages se prêtent plus à la compilation et d'autres plus à l'interprétation. Par exemple les langages où les données ne sont pas déclarées avec un type précis (comme Lisp) sont plus faciles à interpréter qu'à compiler, alors qu'un langage comme C a été pensé dès sa conception pour être facile à compiler.

Exemple : l'addition de deux nombres par un processeur est une opération complètement différente suivant qu'il s'agit de nombres entiers ou de nombres en virgule flottante. On peut en Lisp écrire une fonction (stupide) qui additionne deux nombres sans spécifier leur type :

```
(defun foo (a b) (+ a b))
```

Quand on appelle cette fonction avec deux arguments entiers (par exemple avec `(foo 1 1)`) l'interprète effectue une addition entière parce qu'il détecte que les deux arguments sont du type entier ; si on l'appelle avec deux nombres en virgule flottante (par exemple avec `(foo 1.1 1.1)`), alors il effectue une addition en virgule flottante. Dans un langage où les données sont typées, comme le C, le compilateur a besoin de connaître le type des arguments *au moment où il compile la fonction foo*, sans pouvoir se référer à la façon dont la fonction est appelée. Il faudra par exemple avoir deux fonctions différentes pour traiter les nombres entiers et les nombres en virgule flottante

```
int iadd(int a, int b){ return a+b; }
float fadd(float a, float b){ return a+b; }
```

1.3.2 Compilateur + interprètes

Pour mélanger les avantages des interprètes et des compilateurs, on rencontre souvent des situations où les concepteurs mélangent interprétation et compilation.

Dans les langages fonctionnels, il est souvent possible d'affecter des types aux données pour faciliter la compilation de parties du programme.

On a souvent un compilateur qui traduit le langage source dans un langage cible universel, proche du langage machine, puis un interprète qui se charge d'exécuter ce programme en langage cible (plus rapidement qu'il ne le ferait à partir du langage source). Pour des raisons historiques, on appelle souvent ce langage intermédiaire du *byte-code*. Le compilateur est écrit une fois pour toutes, et l'interprète du langage intermédiaire est plus petit, plus rapide et beaucoup plus facile à porter sur un nouveau processeur qu'un interprète pour le langage source. On peut avoir des compilateurs pour des langages différents qui produisent tous du byte-code pour la même machine virtuelle, et on pourra alors exécuter les programmes du langage source vers tous les processeurs pour lesquels il existe des interprètes. Voir la figure 1.1.

Un exemple historique est le compilateur *Pascal UCSD* qui traduisait les programmes en langage Pascal dans un langage machine universel, appelé P-code qui était ensuite interprété.

Usuellement, les programmes Prolog sont interprétés ; il est aussi souvent possible de les compiler, en général vers le langage machine d'un processeur adapté à Prolog qu'on appelle WAM (comme *Warren Abstract Machine*), puis les instructions de la WAM sont traduites par un interprète dans le langage machine du processeur. (L'ouvrage de Hassan Aït-Kaci, *Warren's Abstract Machine : A Tutorial Reconstruction*, 1991, est disponible sur le web et contient une description pédagogique et relativement abordable de la WAM.)

Un exemple récent concerne le langage Java ; normalement un programme Java est traduit par un compilateur dans le langage d'une machine universelle

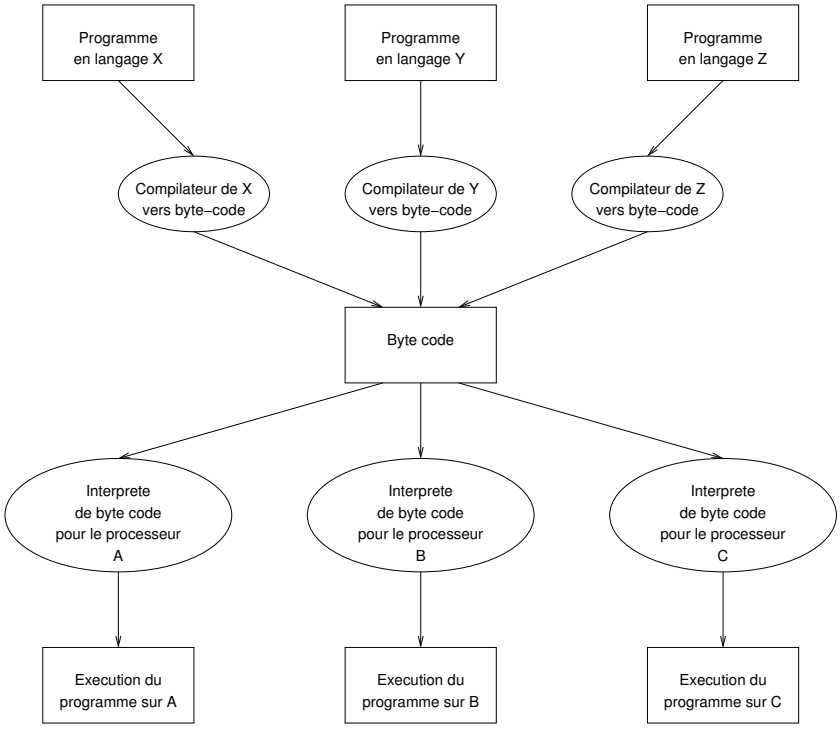


FIGURE 1.1 – La combinaison d'un compilateur vers un byte-code unique et d'interprètes pour ce byte-code permet de combiner des avantages des compilateurs et des interprètes.

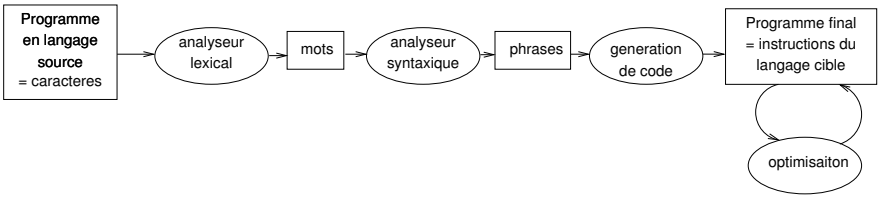


FIGURE 1.2 – La structure générale simplifiée d'un compilateur, qui prend un programme en langage source et produit un programme équivalent en langage cible.

appelée JVM (comme *Java Virtual Machine*) ; ce programme pour la JVM peut ensuite être transmis entre machines puis interprété. Pour obtenir des performances décentes, ce programme pour la JVM peut aussi être compilé juste avant son exécution sur la machine cible (on parle alors de compilation JIT – *Just In Time*). La compilation du langage de la JVM vers le langage du processeur est beaucoup plus simple et beaucoup plus rapide que la compilation du programme Java originel. L'exécution du programme compilé est plus rapide que l'interprétation du byte-code d'origine.

1.4 La structure générale d'un compilateur

Un compilateur est en général un programme trop gros pour qu'il soit possible de le maîtriser d'un coup. Pour avoir des morceaux plus faciles à digérer, on le divise usuellement en parties distinctes qui jouent chacune un rôle particulier. La structure usuelle est résumée dans la figure 1.2

On va voir sommairement le rôle de chacune ces parties dans cette section, puis en détail dans des chapitres ultérieurs.

1.4.1 L'analyseur lexical

Au départ, le programme en langage source se présente comme une suite de caractères. Le premier travail est effectué par l'*analyseur lexical*, qui va découper cette séquence de caractères en mots constitutifs du langage. Le résultat est une séquence de mots, avec leurs types et leurs valeurs autant qu'il lui est possible de les reconnaître. Par exemple, le programme C

```
main(){printf("Hello world\n");}
```

sera découpé en 10 mots comme dans la table suivante :

caractères	type	valeur
`main`	identificateur	`main`
`(`	parenthèse ouvrante	
`)`	parenthèse fermante	
`{`	accolade ouvrante	
`printf`	identificateur	`printf`
`(`	parenthèse ouvrante	
`"Hello world\n"`	chaîne de caractères	`Hello world\n`
`)`	parenthèse fermante	
`;`	point virgule	
`}`	accolade fermante	

Le travail n'est pas toujours aussi facile que dans cet exemple simple, et il n'est pas tout à fait évident de découper du C valide comme `a+++b` ou `a--2`. Les nombres flottants notamment offrent toutes sortes de pièges divers qui compliquent les opérations.

1.4.2 L'analyseur syntaxique

L'analyseur syntaxique regroupe les mots produits par l'analyseur lexical en phrases, en identifiant le rôle de chacun des mots. Les phrases en sortie sont souvent représentées sous la forme d'un *arbre syntaxique* dont les feuilles contiennent les mots, leurs valeurs et leurs rôles et les noeuds internes la manière dont il sont regroupés, ou bien sous la forme d'un programme en langage intermédiaire.

Les mots du programme précédent seront ainsi regroupés, directement sous la forme d'un arbre syntaxique (implicitement ou explicitement) comme dans la figure 1.3

1.4.3 La génération de code

La génération de code consiste à traduire les phrases produites par l'analyseur syntaxique dans le langage cible. Pour chaque construction qui peut apparaître dans une phrase, le générateur de code contient une manière de la traduire.

1.4.4 L'optimisation du code

Il y a souvent des modifications du code généré qui permettent d'obtenir un programme équivalent mais plus rapide. Pour cette raison, l'optimiseur examine le code généré pour le remplacer par du meilleur code.

Il y a trois raisons principales pour ne pas produire directement du code optimisé dans le générateur de code. D'une part cela permet au générateur de code de rester plus simple. D'autre part l'optimisation peut être très coûteuse en temps et en espace mémoire, et cela permet de sauter cette étape et

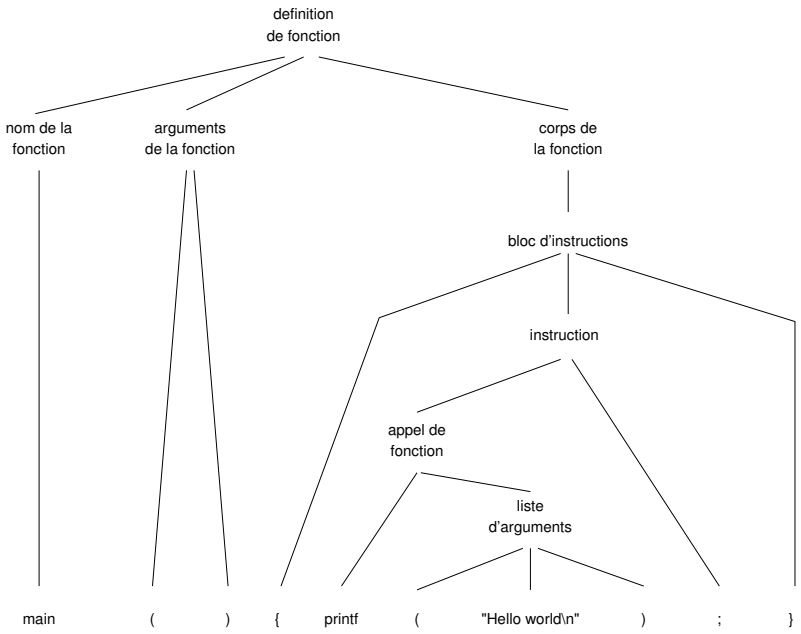


FIGURE 1.3 – Un arbre syntaxique permet d'identifier le rôle de chaque mot et de chaque groupe de mots d'un programme.

d'obtenir plus rapidement un programme à tester qui ne tournera qu'une seule fois ; le gain de temps obtenu en sautant l'optimisation compense largement le (modeste) gain de temps qu'on aurait obtenu en faisant tourner une seule fois le programme optimisé. Finalement il est plus difficile de mettre au point un programme optimisé : certaines instructions du programme source ont pu être combinées ou réordonnées, certaines variables ont pu disparaître.

1.4.5 Variantes

En C, avant l'analyseur lexical, il y a un traitement par le *préprocesseur*, qui traite les lignes qui commencent par `#`, comme les `# include` et les `# define` ; on peut voir le préprocesseur comme un compilateur, qui possède lui aussi un analyseur lexical et un analyseur syntaxique et qui produit du C.

Le langage cible du premier compilateur C++, nommé `cfront`, a été pendant longtemps le langage C ; le compilateur C se chargeait ensuite de traduire le programme C en langage machine.

Fréquemment, la génération de code se fait en deux étapes ; l'analyseur syntaxique (ou un premier générateur de code) produit des instructions dans un langage intermédiaire, proche de l'assembleur ; c'est là-dessus que travaille l'optimiseur. Ensuite ce langage intermédiaire est traduit en langage cible. Cela

permet d'avoir un optimiseur qui fonctionne quel que soit le langage cible.

1.5 Qui compile le compilateur ?

Le plus souvent, un compilateur produit du code pour le microprocesseur sur lequel il s'exécute. Il y a deux exceptions importantes : quand la machine qui doit exécuter le programme compilé n'est pas accessible ou pas assez puissante pour faire tourner le compilateur (c'est souvent le cas dans les systèmes embarqués) et quand on fabrique le premier compilateur d'une (famille de) machine, ce qu'on appelle *bootstrapper* un compilateur, par analogie avec le démarrage de la machine.

On appelle ce type de compilateur un compilateur *croisé*, par opposition au cas courant qu'on appelle un compilateur *natif*.

1.5.1 La compilation croisée

On utilise fréquemment les compilateurs croisés pour produire du code sur les machines embarquées, qui ne disposent souvent pas de ressources suffisantes pour faire tourner leur propre compilateur.

La méthode est évoquée sur la figure 1.4. Sur une machine A, on a utilise le compilateur pour A pour compiler le compilateur pour B : on obtient un programme qui tourne sur A et qui produit du code pour B. Ensuite, toujours sur A, on utilise ce compilateur pour produire du code susceptible de s'exécuter sur B. Ensuite on doit transfèrer ce code sur la machine B pour l'exécuter.

L'installation et l'utilisation des compilateurs croisés sont souvent des opérations complexes. Il n'y a pas que le programme compilateur à modifier, mais aussi les bibliothèques et souvent une partie de leurs fichiers de `/usr/include`.

Dans le système Plan 9, ce problème a été résolu d'une façon particulièrement élégante : toutes les compilations sont des compilations croisées, y compris lorsqu'on compile pour la machine sur laquelle on compile. Le résultat est d'une simplicité d'utilisation déroutante.

1.5.2 Le bootstrap d'un compilateur

Obtenir un compilateur sur une nouvelle architecture est un cas spécial de compilation croisée, comme indiqué sur la figure 1.5.

Le programme destiné à s'exécuter sur la machine B est le compilateur lui-même. Une fois qu'il est transféré sur B, il va servir à recompiler le compilateur pour B, de manière à obtenir un compilateur natif compilé par un compilateur natif. En guise de test, on va sans doute procéder à encore une compilation afin de comparer les deux compilateurs.

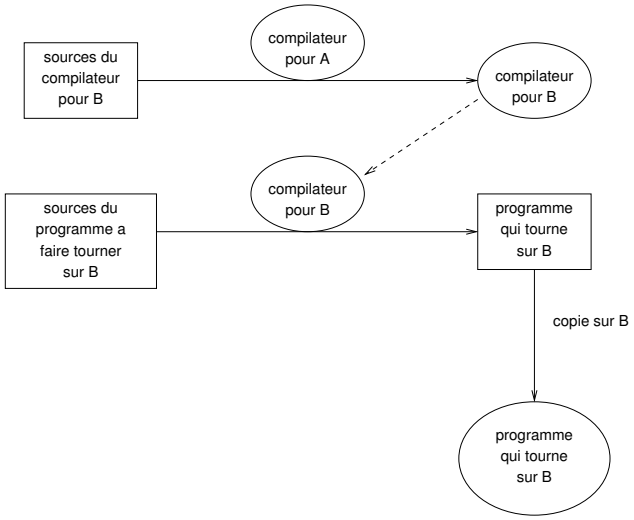


FIGURE 1.4 – Compilation croisée d'un programme : les rectangles entourent des fichiers dont le contenu n'est pas exécutable dans le contexte où ils se trouvent, alors que les ellipses entourent des programmes qui peuvent s'exécuter. Les deux compilations ont lieu sur la machine A.

1.6 Après la compilation, l'édition de liens (Supplément)

Sous Unix le compilateur C (et beaucoup d'autres compilateurs) produisent du langage machine mais avant l'exécution du programme se placent encore trois étapes : l'édition de liens, le chargement et la résolution des bibliothèques dynamiques. On pourrait discuter pour savoir si ces étapes font parties de la compilation ou pas.

1.6.1 L'édition de liens

Le compilateur compile indépendamment chacun des fichiers et pour chacun produit un fichier qui contient la traduction du programme en langage machine accompagné d'une table des symboles, dans un fichier qui porte en général un nom qui se termine par `.o` (comme *objet*). Si on appelle le compilateur C avec l'option `-c`, il s'arrête à cette étape.

Si on l'appelle sans l'option `-c`, le compilateur passe ensuite la liste des fichiers `.o` qu'il vient de traiter à l'éditeur de liens. Celui-ci regroupe tous les fichiers `.o` dans un seul fichier et y ajoute le code nécessaire qu'il va prendre dans les bibliothèques.

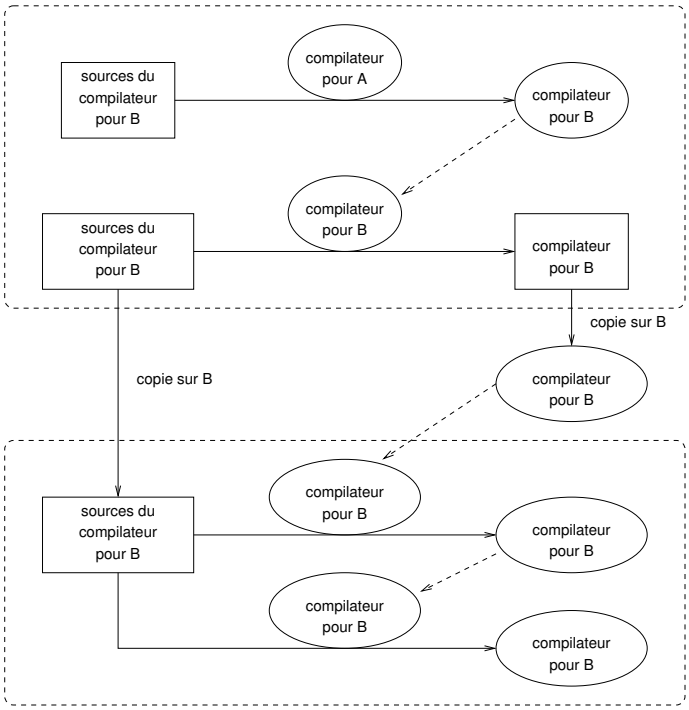


FIGURE 1.5 – La fabrication du premier compilateur natif pour la machine B, à l'aide de la machine A. Les deux compilations du haut se produisent sur A, les deux du bas sur B, la machine cible.

Le regroupement des fichiers `.o`

Quand le compilateur produit du langage machine, il doit spécifier à quel endroit de la mémoire les instructions seront placées. (Par exemple on verra au prochain chapitre que les instructions de branchement contiennent l'adresse de l'instruction à laquelle sauter ; il faut bien savoir à quelle adresse se trouve cette instruction pour pouvoir y sauter.) Le compilateur spécifie que toutes les instructions sont placées à partir de l'adresse 0 et il ajoute au langage machine une *table des symboles* qui contient tous les endroits où une adresse est utilisée, ainsi que les endroits du programme qui font appel à des adresses que le compilateur ne connait pas (par exemple l'adresse de la fonction `printf`).

L'éditeur de lien prend les morceaux de code présents dans chaque fichier objet et les regroupe en leur assignant des adresses différentes. En utilisant les tables des symboles, il met aussi à jour chaque référence à une adresse qui a été modifiée.

Les choses sont un tout petit plus complexes que décrit ici parce que le code produit par le compilateur est en fait affecté d'un *type* : les instructions, les données qui ne seront pas modifiées, celles qui sont initialisées à 0 et celles initialisées avec autre chose que 0. L'éditeur de lien regroupe dans le fichier résultat le code par type. Ceci permet d'utiliser le gestionnaire de mémoire virtuelle pendant l'exécution du programme pour s'assurer que le programme ne modifie pas ses propres instructions et pour faire partager les instructions et les données pas modifiées par les différents processus qui exécutent le même programme. (De plus, la mémoire qui est initialisée à 0 n'est stockée que sous la forme d'un compteur.)

Le mot clef `static` affecté à une variable globale ou une fonction C permet d'éviter qu'elle apparaisse dans la table des symboles du fichier objet et donc qu'elle soit utilisée dans un autre fichier.

C'est à cette étape qu'il est possible de détecter les fonctions ou les variables qui sont définies plusieurs fois dans le programme (par exemple parce qu'elles sont définies dans un fichier `.h` qui est inclus dans plusieurs fichiers `.c` ; dans les fichiers `.h`, on prévient le compilateur de l'existence des variables globales, mais avec `extern` on lui indique qu'il ne doit pas les définir.

Les bibliothèques

Le programme compilé fait le plus souvent appel à des fonctions qu'il n'a pas définies comme `printf`. L'éditeur de liens va chercher dans ces bibliothèques le code qui correspond aux fonctions dont il n'a pas trouvé le code dans le programme.

(Une erreur commune chez les débutants consiste à imaginer que la fonction `printf` se trouve dans le fichier `stdio.h` ; en réalité, les fichiers `.h` contiennent des *prototypes* de fonction qui permettent de prévenir le compilateur de leur existence, du type d'arguments qu'elles attendent et du type de valeurs qu'elles

renvoient, mais le code de printf est lui compilé une fois pour toutes et placé dans une bibliothèque.)

Le plus souvent, ces bibliothèques sont placées dans les répertoires `/lib` ou `/usr/lib`. Par défaut, le compilateur C demande à l'éditeur de lien d'utiliser la seule bibliothèque C standard. On peut lui demander d'utiliser d'autres bibliothèques avec l'option `-l` ; par exemple l'option `-lm` indique qu'il faut utiliser la bibliothèque de fonctions mathématiques et l'éditeur de liens ira consulter aussi le contenu des fichiers `libm.a` ou `libm.so.*`.

Les bibliothèques sont une forme compacte pour stocker de nombreux fichiers `.o` ; il peut s'agir d'une bibliothèque statique (dans ce cas son nom se termine par `.a`, l'éditeur de lien extrait le code de la fonction et l'ajoute dans le fichier résultat) ou bien d'une bibliothèque dynamique (dans ce cas son nom se termine par `.so` suivi d'un numéro de version et l'éditeur de lien ajoute au fichier résultat l'information nécessaire pour que le code soit ajouté au moment où le programme sera lancé).

À cette étape, il n'est pas possible de détecter les fonctions qui existent dans la bibliothèque mais qui ne sont pas ajoutées au programme parce que le programme contient déja une fonction de ce nom.

Le code de démarrage

Une tâche supplémentaire confiée par le compilateur à l'éditeur de lien est de mettre en place l'environnement nécessaire pour faire tourner un programme C. Pour cela, le compilateur passe à l'éditeur de lien un fichier `.o` supplémentaire qui contient l'initialisation de l'environnement, l'appel à la fonction `main` et la sortie du processus (si la fonction `main` n'est pas sortie directement avec l'appel système `exit`).

Ce fichier porte usuellement un nom qui contient `crt` comme *C runtime*. Il en existe différentes versions suivant le compilateur et la manière dont le programme est compilé.

Le résultat de l'édition de lien est placé dans un fichier qui porte par défaut le nom `a.out`.

Les bibliothèques dynamiques

L'éditeur de lien peut aussi ajouter au programme des références à des bibliothèques *dynamiques* ; ce n'est pas le code extrait de la bibliothèque qui est ajouté au fichier, mais une simple référence au code qui reste dans le fichier bibliothèque.

Dans tous les systèmes Linux que j'ai utilisé, le compilateur choisit par défaut d'utiliser les bibliothèques dynamiques. Personnellement, je pense le plus grand mal de ces bibliothèques. À cause d'elles, on ne peut pas se contenter d'installer un programme sous la forme d'un fichier exécutable (et éventuellement de son fichier de configuration) ; on a aussi besoin d'installer les librairies qui vont avec

si elles ne sont pas présentes ; si elles sont présentes, c'est souvent avec un numéro de version qui ne va pas ; alors on installe la nouvelle version de la librarie ; mais les autres programmes qui utilisaient l'ancienne version ne fonctionnent plus.

1.6.2 Le chargement en mémoire

Finalement, lorsqu'un processus fait un appel système `exec` d'un fichier objet, le contenu du fichier est chargé en mémoire. Avec les MMU qu'on trouve à peu près sur tous les processeurs, ce n'est pas un problème puisque chaque processus dispose de son propre espace d'adressage : il suffit de positionner la MMU pour que l'espace d'adressage virtuel du processus corresponde à de la mémoire physique. En revanche les systèmes embarqués n'ont fréquemment pas de MMU ; il faut alors que le système *translate* le programme entier, pour que l'image en mémoire corresponde aux adresses physiques où le programme est effectivement chargé.

1.6.3 Commandes Unix

Voici une série de manips pour mettre en œuvre ce qu'on vient de voir. Les manipulations sont présentées sous la forme d'exercices, mais il s'agit essentiellement de lancer des commandes et d'en constater les effets, qui sont indiqués dans l'énoncé.

On partira avec deux fichiers ; l'un contient une fonction `pgcd` que nous verrons dans un chapitre ultérieur et l'autre une fonction `main` qui sert à appeler cette fonction. Le fichier `pgcd.c` contient

```
1  /* pgcd.c
2   */
3  int
4  pgcd(int a, int b){
5          int t;
6
7          while(a != 0){
8                  if (a < b){
9                          t = b;
10                         b = a;
11                         a = t;
12                 }
13                 a -= b;
14         }
15         return b;
16 }
```

Le fichier `main.c` contient

```
1  /* main.c
```

```
 */
# include <stdio.h>

int
main(int ac, char * av[]){
        if (ac < 3){
                fprintf(stderr, "usage: %s N M\n", av[0]);
                return 1;
         }
        printf("%d\n", pgcd(atoi(av[1]), atoi(av[2])));
        return 0;
}
```

Produire un exécutable et l'exécuter avec les commandes

```
$ gcc main.c pgcd.c
$ a.out 91 28
```

On obtient un message d'erreur si on essaye de produire un exécutable à partir d'un seul de ces fichiers avec

```
$ gcc pgcd.c
$ gcc main.c
```

Ex. 1.1 — Quel sont les messages d'erreur ? À quelle étape de la compilation sont-ils produits ?

Compiler chaque fichier indépendamment pour obtenir un fichier .o

```
$ gcc -c main.c
$ gcc -c pgcd.c
```

Examiner avec la commande `nm` la table des symboles des fichiers `.o` produits par ces commandes. Dans la sortie de `nm`, chaque symbole occupe une ligne ; si le symbole est marqué comme `U`, c'est qu'il reste à définir ; s'il est marqué comme `T` c'est qu'il s'agit de code à exécuter et on voit sa valeur.

Ex. 1.2 — Quels sont les symboles indéfinis dans chacun des fichiers ?

Combiner les deux fichiers `.o` pour avoir un exécutable et l'exécuter avec

```
$ gcc main.o pgcd.o
$ a.out 85 68
```

Ex. 1.3 — Qu'imprime la commande `a.out` ?

La commande `nm` permet aussi d'examiner la table des symboles du fichier exécutable.

Ex. 1.4 — Quels sont les symboles indéfinis dans le fichier `a.out` obtenu précédemment ?

Ex. 1.5 — A quelles adresses se trouvent les fonctions `main` et `pgcd` ?

Sauver le fichier a.out sous le nom `dyn.out` et compiler le programme sous la forme d'un programme qui utilise les bibliothèques statiques sous le nom `stat.out` avec

```
$ mv a.out dyn.out
$ gcc -static pgcd.c main.c -o stat.out
```

Ex. 1.6 — Quelles sont les tailles respectives de `dyn.out` et de `stat.out` ?

Ex. 1.7 — Le `T` majuscule dans la sortie de `nm` permet à peu près d'identifier les fonctions. Combien y a-t-il de fonctions présentes dans chacun des deux exécutables ?

Ex. 1.8 — La commande `ldd` permet de déterminer les librairies dynamiques nécessaires à un programme. Indiquer les librairies nécessaires pour les deux fichiers exécutables.

Ex. 1.9 — Recompiler les programmes avec l'option `-verbose` de gcc de manière à voir toutes les étapes de la production de l'exécutable.

On va maintenant refaire toutes les étapes en appelant chaque commande directement au lieu de passer par gcc.

On peut appeler le préprocesseur soit avec la commande `cpp` (comme *C PreProcessor* ; il y a parfois un problème avec certains systèmes où le compilateur C++ porte ce nom) ou avec l'option `-E` de `gcc`.

Faire passer le préprocesseur C sur chacun des fichiers C, placer la sortie dans des fichiers nommés `x` et `y`.

Ex. 1.10 — Combien les fichiers `x` et `y` comptent-ils de lignes ?

On peut trouver le compilateur `gcc` proprement dit, avec l'analyse lexicale, l'analyse syntaxique et la génération de code dans un fichier qui porte le nom `cc1`. On peut le trouver avec l'une des deux commandes

```
$ find / -name cc1 -print
$ locate */cc1
```

Ex. 1.11 — Où se trouve le fichier `cc1` sur votre ordinateur ?

Le programme `cc1` peut lire le programme sur son entrée standard et place dans ce cas la sortie dans un fichier qui porte le nom `gccdump`. On peut donc compiler la sortie de l'étape précédente avec

```
$ cc1 < x
$ mv gccdump.s x2
$ cc1 < y
$ mv gccdump.s y2
```

Ex. 1.12 — Que contiennent les fichiers `x2` et `y2` ?

Il faut ensuite faire traiter le contenu des fichiers `x2` et `y2` par la commande `as`, qui place sa sortie dans un fichier nommé `a.out`. On peut donc faire

```
$ as x2
$ mv a.out x3
$ as y2
$ mv a.out y3
```

Ex. 1.13 — Que contiennent les fichiers `x3` et `y3` ?

Il est aussi possible de pratiquer l'édition de lien directement sans passer par la commande gcc, mais ça devient franchement pénible. On revient donc au programme principal pour la dernière étape.

```
$ mv x3 x.o
$ mv y3 y.o
$ gcc x.o y.o
```

Ex. 1.14 — Examiner le fichier `a.out` produit avec la commande `readelf` (pas de correction).

Ex. 1.15 — Trouver sur votre ordinateur tous les fichiers qui portent un nom du type `crt*.o`.

Ex. 1.16 — Trouver sur votre machine la librairie C standard statique (qui s'appelle `libc.a`).

Ex. 1.17 — Examiner le contenu de `libc.a` avec la commande `ar t` (pas de correction).

Ex. 1.18 — Extraire, de nouveau avec `ar`, le fichier `printf.o` de `libc.a`. Combien occupe-t-il d'octets ?

1.7 Plan du reste du cours

Les deux chapitres qui suivent parlent du langage cible le plus courant d'un compilateur : l'assembleur. Le premier traite de l'assembleur en tant que tel, le second examine l'assembleur produit lors de la compilation d'un programme C et permet d'entrer dans le détail de l'organisation de la pile d'un processus qui exécute un programme C.

Le quatrième chapitre présente sommairement l'analyse lexicale, certaines de ses difficultés et la façon de les résoudre avec les outils usuels, sans lex ni les expressions régulières qui sont abordés dans un chapitre ultérieur.

Les trois chapitres qui suivent présentent l'analyse syntaxique. Le premier présente la problématique et un exemple élémentaire d'analyseur à précédence d'opérateur pour les expressions arithmétiques ; le deuxième contient une introduction à Yacc (ou Bison) fondée sur des exemples ; dans le troisième, on trouvera ce qu'il faut de théorie pour mettre au point les grammaires pour des analyseurs LALR(1).

Le huitième chapitre traite de la génération de code (rapidement) et présente des techniques ordinaires d'optimisation.

Le neuvième chapitre contient une étude de cas : il analyse en détail quelques aspects du compilateur Gcc, notamment les règles de la grammaire du langage à partir de ses sources, ainsi les étapes de la génération de code telles que les décrit la documentation.

Le dixième chapitre présente les sources d'un compilateur complet en une passe pour un sous-ensemble du langage C.

Le onzième chapitre, optionnel, parle des expressions régulières (ou expressions rationnelles) et des outils Lex et Flex qui permettent de les utiliser pour fabriquer des analyseurs lexicaux.

1.8 Références bibliographiques

Il y a de nombreux ouvrages sur la compilation, dont plusieurs sont à la fois excellents et traduits en français.

Le livre de référence sur le sujet était celui de Aho et Ullman, qui a connu une deuxième version avec un auteur supplémentaire : A. Aho, R. Sethi et J. Ullman, *Compilers: Principles, Techniques and Tools*, 1986. Ce livre est traduit en français chez InterEditions sous le nom *Compilateurs : principes, techniques, outils*. On fait souvent référence à ce livre sous le nom de *Dragon Book* parce que la couverture est illustrée avec un dragon.

Il y a eu une deuxième édition de la deuxième version du dragon book en 2006, avec un quatrième auteur, M. Lam ; cette deuxième édition a aussi été traduite en français, mais je ne l'ai pas lue.

Sur les deux outils d'aide à l'écriture des compilateurs Lex et Yacc, il y a un livre chez O'Reilly de J. Levine, intitulé `Lex & Yacc`. L'ouvrage est une bonne introduction, mais à mon avis il accorde trop de place à lex et ne descend pas suffisamment dans les détails du fonctionnement de Yacc pour permettre son utilisation pour des projets sérieux. Il a été traduit en français chez Masson, mais cette traduction ne semble pas disponible à l'heure actuelle.

Je recommande également le livre *Modern Compiler Design* de D. Grune, H. Bal, J. H. Jacobs et K. Langendoen paru chez John Wiley and Sons en 2000, qui a été traduit chez Dunod sous le titre *Compilateurs*. Il contient notamment un chapitre sur la compilation du langage fonctionnel Haskell et un autre sur la compilation de Prolog.

Chapitre 2

L'assembleur

Ce chapitre contient une brève présentation de l'assembleur du processeur Intel 386 en mode 32 bits. Il vous donne les compétences nécessaires pour lire et écrire des programmes simples en assembleur.

2.1 Assembleur et langage machine

Le processeur lit des instructions dans la mémoire sous la forme de codes numériques ; ce sont ces codes numériques qu'on appelle du langage machine. Comme il est très peu pratique pour nous de manipuler ces valeurs numériques pour écrire de gros programmes corrects [1], on y accède en général sous la forme de chaînes de caractères qui sont traduites directement (et une par une) en langage machine ; c'est ce qu'on appelle de l'*assembleur*.

En plus des instructions, les programmes en assembleur contiennent des *directives* qui ne sont pas traduites directement en langage machine mais indiquent à l'assembleur de quelle manière il doit effectuer son travail.

2.2 Brève introduction à la programmation du processeur Intel 386

J'utilise le processeur Intel 386 quoiqu'il n'ait pas une architecture très plaisante, parce qu'il est très répandu ; on peut l'utiliser sur tous les processeurs Intel 32 bits. De toute façon, quand on sait programmer un processeur en assembleur, il est plutôt facile de s'adapter à un nouveau processeur.

Si le système installé sur votre machine est un système 64 bits, reportez-vous aux instructions spécifiques de la fin du chapitre suivant.

1. C'est facile d'écrire des gros programmes faux directement en code machine et pas très difficile d'en écrire de petits corrects.

Le processeur 386 possède un mode 16 bits dont nous ne parlerons à peu près pas, qui existe principalement pour lui permettre de continuer à exécuter les programmes écrits pour le processeur précédent d'Intel, le 286. Nous utiliserons le mode 32 bits.

Un processeur se caractérise par 3 choses :
- le jeu d'instructions, qui décrit les choses qu'il sait faire
- les types de données disponibles (la mémoire et les registres, qui sont des mots mémoire pour lesquels l'accès est super-rapide)
- les modes d'adressage, c'est à dire la façon de désigner les opérandes

et la manière dont tout ceci se combine ; par exemple, certaines instructions ne fonctionneront que sur certains opérandes adressés d'une certaine façon.

2.2.1 Les opérandes

Les opérandes sont les registres et la mémoire.

Sur le processeur 386, il y a 16 registres de 32 bits, dont 8 registres généraux :
- Il y a quatre vrais registres généraux `%eax`, `%ebx`, `%ecx`, `%edx`. Sur les processeurs précédents de la série, ils contenaient 8 bits (un octet) et portaient les noms `a`, `b`, `c` et `d`. Quand ils sont passés à 16 bits, on leur a ajouté un `x` comme *eXtended*, et ils se sont appelés `%ax`, `%bx`, `%cx` et `%dx`. Quand ils ont passés à 32 bits, on a ajouté un `e` devant comme *Extended* et ils portent maintenant ces noms ridicules où *extended* apparaît deux fois avec deux abréviations différentes
- `%ebp` et `%esp` sont deux registres, pas si généraux que ça, qu'on utilise pour la base et la pile (*Base Pointer* et *Stack Pointer*. Nous reviendrons longuement sur leurs rôles dans le prochain chapitre.
- `%esi` et `%edi` ne sont pas tellement généraux non plus ; ils sont spécialisés dans les opérations sur les blocs d'octets dans la mémoire ; on peut cependant les utiliser comme espace de stockage et pour les calculs arithmétiques courants.
- Il y a six registres de segment : `%cs`, `%ds`, `%ss`, `%es`, `%fs`, and `%gs` : on peut les oublier ; puisqu'on n'utilise pas le mode d'adressage 16 bits du 386, nous ne les utilisons pas.
- Il y a enfin deux registres spécialisés : `%eflags` est le registre d'état, qui contiendra notamment le résultat de la dernière comparaison et la retenue de la dernière opération ; `%eip` (comme *Extended Instruction Pointer*) contient l'adresse de la prochaine instruction à exécuter.

Attention, les registres existent en un seul exemplaire, comme une variable globale du C. Ils ne sont pas spécifiques d'un appel comme une variable locale.

La mémoire est organisée sous la forme d'un tableau d'octets, indicé par une valeur qui varie entre 0 et $2^{32} - 1$ (codée sur 32 bits d'adresse).

On peut adresser soit un octet, soit un mot de 16 bits (sur deux octets), soit un long de 32 bits (sur quatre octets).

Les mots peuvent contenir n'importe quoi, mais le 386 ne saura calculer que

sur les valeurs entières sans signe ou codées en complément à deux (et aussi, pour mémoire, sur le Décimal Codé Binaire). Les successeurs du 386 permettent aussi de calculer sur les nombres flottants.

2.2.2 Les modes d'adressages

Pour chaque mode d'adressage, je donne un exemple d'utilisation avec son équivalent en C en commentaire, puis quelques explications. Dans l'équivalent C, `i` et `j` désignent des variables automatiques entières et `p` un pointeur. Presque tous les exemples utilisent l'instruction `movl` qui recopie un mot de 32 bits. (Comme son nom ne l'indique pas, `movl` ne modifie pas la valeur désignée par l'opérande de gauche.)

Adressage registre

Désigne un registre.

```
movl %eax,%ebx          // j = i
```

copie dans le registre `%ebx` la valeur présente dans le registre `%eax` C'est ce qu'on utilise pour accéder à l'équivalent des variables ordinaires en C.

Adressage immédiat

Désigne une valeur immédiate (présente dans l'instruction).

```
movl $123,%eax          // i = 123
```

place la valeur 123 dans le registre `%eax`; comme la valeur est codée dans l'instruction, cela signifie qu'on ne peut absolument pas la modifier. C'est l'équivalent des constantes dans un programme C.

Dans les assembleurs d'Unix le caractère `$` sert à indiquer qu'il s'agit de la *valeur* 123. Les autres assembleurs utilisent souvent le dièse `#` pour cela mais sous Unix ce caractère est trop largement utilisé par le préprocesseur C.

Adressage absolu

Désigne un mot mémoire par son adresse

```
movl $123,456
```

place la valeur 123 dans les quatre octets qui se trouvent aux adresses 456 à 459, considérés comme un seul mot de 32 bits. C'est l'équivalent de l'utilisation d'une variable globale dans un programme C.

Adressage indirect

Désigne un mot mémoire dont l'adresse est dans un registre.

```
movl (%ebp),%eax        // i = *p
```

place dans `%eax` la valeur dont l'adresse se trouve dans `%ebp`. C'est l'équivalent d'une indirection à travers un pointeur en C.

Adressage indirect+déplacement

```
movl 24(%esp),%edi      // z = tab[8]
```

place dans `%edi` la valeur qui se trouve 24 octets au-dessus du mot dont l'adresse est dans `%esp`. C'est l'équivalent d'une indexation dans un tableau par une constante en C, ou de l'accès à un champs d'une structure.

Adressage indirect+index

```
movl $123,(%eax,%ebp)   // tab[x] = 123
```

additionne le contenu d'`%eax` et d'`%ebp` : ça donne l'adresse du mot où il faut placer la valeur 123. C'est l'équivalent de l'accès à un tableau avec un index variable.

Adressage indirect + déplacement + multiplication + index

On a aussi un mode dans lequel on peut combiner les deux derniers, plus multiplier le contenu d'un des registres par 2, 4 ou 8. Par exemple :

```
movl 24(%ebp,%eax,4),%eax
```

multiplie le contenu de `%eax` par 4, lui ajoute le contenu de `%ebp` et 24 ; on a ainsi l'adresse d'un mot mémoire dont le contenu est recopié dans `%eax`.

2.2.3 Les instructions

Je commence par présenter les types de données manipulées par le 386 ; ensuite j'ai regroupé les instructions par type d'opération effectuée, avec d'abord celles qui me semblent indispensables puis les autres, qu'on peut généralement ignorer.

Les types de données

La plupart des instructions peuvent manipuler des octets, des mots ou des longs ; pour les distinguer, on les postfixe avec `b`, `w` ou `l`. Ainsi l'instruction

`movb` déplacera un octet, l'instruction `movw` un mot de 16 bits sur deux octets et l'instruction `movl` un mot de 32 bits sur quatre octets.

Les registres ne portent pas non plus le même nom suivant la taille de l'opérande à utiliser :

`movb %al,%ah`	recopie les bits 0–7 d'`%eax` dans les bits 8–15
`movw %ax,%bx`	recopie les bits 0–15 d'`%eax` dans ceux d'`%ebx`
`movl %eax,%ebx`	recopie les 32 bits d'`%eax` dans `%ebx`
`movb %al,31415`	recopie les bits 0-7 d'`%eax` à l'adresse 31415
`movw %ax,31415`	recopie les bits 0-15 d'`%eax` aux adresses 31415-31416
`movl %eax,31415`	recopie les bits 0-32 d'`%eax` aux adresses 31415-31418

Déplacement de données

`mov` : recopie une donnée

`xchg` : échange deux données

`push` : empile une valeur

`pop` : dépile une valeur

`lea` : calcule une adresse et copie la (*Load Effective Address*)

`pea` : calcule une adresse et empile la (*Push Effective Address*)

conversions

`movsx` : convertit une valeur signée sur un octet (ou un mot) en une valeur sur un mot (ou un long)

`movzx` : pareil avec une valeur non signée

arithmétique

`addl %eax,%ebx` : ajoute le contenu d'`%eax` à celui d'`%ebx`

`incl %eax` : ajoute 1 au contenu d'`%eax`

`subl %eax,%ebx` : ôte le contenu d'`%eax` de celui d'`%ebx`

`decl %eax` : retire 1 au contenu d'`%eax`

`cmpl %eax,%ebx` : comme une soustraction mais sans stocker le résultat

`negl %eax` : change le signe du contenu d'`%eax`

`imull %ebx,%ecx` : multiplication, résultat dans `%ecx`

`idivl %ecx` : divise `%edx-%eax` par `%ecx`, résultat dans `%eax` et reste dans `%edx`

logique, décalage, rotation

`and` : et logique (bit à bit) entre les deux opérandes

`or` : ou logique (bit à bit) entre les deux opérandes

`xor` : ou exclusif (bit à bit) entre les deux opérandes

`not` : nie chaque bit de son opérande

`sal, shr` : décalage à gauche, signé ou pas (c'est pareil)

`sar, shr` : décalage à droite, signé ou pas (ce n'est pas pareil ; dans un décalage de nombre non signé les places libérées sont remplies avec des bits à 0 alors que pour un nombre signé elles sont remplies avec des copies du bit de signe).

transfert de contrôle

`jb` : jump or branch, la prochaine instruction sera celle dont l'adresse est indiquée par l'opérande ; `jump` et `branch` sont deux instructions différentes pour le processeur dans lesquelles l'adresse de la prochaine instruction n'est pas codée de la même manière. L'assembleur choisit à notre place la plus compacte des deux. suivant la valeur de l'adresse destination.

`jXX` : jump conditionnal ; le saut ne sera effectué que si la condition est remplie. XX indique la condition et peut être `e` (equal), `ne` (not equal), `g` (*greater*), `l` (*less*), `ge`, `le` etc. Le résultat testé est celui de la dernière opération.

`call xxx` : pour les appels de fonctions, cette instruction empile l'adresse qui suit l'instruction et place `xxx` dans `%eip`.

`ret` : pour revenir d'une fonction, l'instruction dépile l'adresse qui se trouve au sommet de la pile et la place dans `%eip`.

`int` : active une exception, pour effectuer un appel système.

prologue et épilogue de fonctions

Nous reviendrons plus en détail sur ces instructions dans le chapitre suivant

`enter $24,%ebp` : équivalent à `pushl %ebp ; movl %esp,%ebp ; subl $24,%esp`

`leave` : équivalent à `movl %ebp,%esp; popl %ebp`

mouvement (facultatif)

`pusha/popa` : push/pop all registers (les 8 registres dits généraux)

`lahf` : recopie %flags dans %ah

`sahf` : recopie %ah dans %flags

`pushf/popf` : push/pop %flags

conversions (facultatif)

`cdq (cwd)` : duplique le bit de signe d'%eax (%ax) dans %edx (%dx).

`cbw (cwde)` : duplique le bit de signe d'%al (%ax) dans %ax (%eax)

arithmétique (facultatif)

`addc` : addition avec retenue

`subb` : soustraction avec retenue

`mulb %bl` : multiplication non signée de %al, résultat dans %ah-%al

`mulw %bx` : idem pour %ax, résultat dans %ax-%dx

`mull %ebx` : idem pour %eax résultat dans %eax-%edx

`imul` : avec un seul opérande, c'est comme comme mul, mais pour des entiers signés

`imul` : avec trois opérandes, constante × opérande dans registre

`div, idiv` : division pour les octets et les mots courts

`daa, das, aaa, aas, aam, aad` : pour le Décimal Codé Binaire ; le principe est d'avoir deux chiffres décimaux codés dans les deux groupes de quatre bits d'un octet ; ce codage des nombres est utile pour les langages comme Cobol ou PL/1 dans lesquels on peut spécifier le nombre de chiffre décimaux de précision des variables.

logique, décalage, rotation (facultatif)

`bt` : bit test

`bts` : bit test and set

`btr` : bit test and reset

`btc` : bit test and complément

`bsf` : bit scan forward : trouve l'index du premier bit à 1 (dans un octet, mot ou long)

`bsr` : bit scan reverse : idem pour le dernier bit à 1

`shld, shrd` : décale deux mots pour en produire un seul (ça permet de décaler des zones de mémoire facilement)

`rol, ror` : rotate left or right

`rcl, rcr` : idem sauf que ça décale aussi la retenue

`test` : comme and, mais ne stocke pas le résultat

transfert de contrôle (facultatif)

`iret` : retour d'exception n'est utilisé que dans le noyau du système d'exploitation pour un retour après un appel système.

`loop` : décrémente `%ecx` et se débrancher si `%ecx` vaut 0

`loope, loopne` : idem mais regarde aussi la valeur du flag Z

chaînes (faculatatif)

`movs, cmps, scas, lods,stos` : move, compare, scan, load, store opèrent sur des suites d'octets (des chaînes de caractères) sur lesquelles pointent `%esi` et `%edi` (*source* et *destination index*) et dont la longueur est dans `%ecx`. Il y a dans le mot d'état un flag (`DF` : direction flag) qui indique s'il faut incrémenter ou décrémenter `%esi` et `%edi` et un préfixe "rep" qui permet de répéter l'opération jusqu'à ce que `%ecx` contienne 0.

`std, cld` : mettre DF à un (décrémenter) ou à zéro (incrémenter)

autres (facultatif)

`setXX` : positionne un octet suivant l'état des codes conditions

`stc, clc, cmc` : set/clear/complement le carry (= la retenue)

`xlat` : translate, %al = %ebx[%al]

Il y a aussi des instructions spécialisées pour la lecture, écriture et modification des registres de segments, mais on ne parle pas de mémoire segmentée. De même, nous n'aurons pas l'occasion de faire des entrées sorties directement depuis l'assembleur.

2.2.4 Les directives

`.long, .word, .byte, .string` : réserve de la place dans la mémoire

`.globl` : fait apparaître un symbole dans la table des symboles du fichier fabriqué.

`.text, .data` : spécifie dans quelle zone de la mémoire (une *section*) l'assembleur doit placer le code qui suit.

Ex. 2.1 — Indiquer ce qui est modifié (avec sa nouvelle valeur) par chacune des instructions suivantes, exécutées en séquence (en supposant que la mémoire utilisée est disponible) :

```
movl $24,%eax
movl $28,%ebx

movl %eax,24

addl %ebx,%eax
movl %eax,(%ebx)
movl %eax,(%eax)

movl 24(%ebx),%ecx
movl 24,%eax
subl (%eax),%ecx
```

```
movl %ecx,4(%eax)

movl $123,(%ecx,%eax)
movl $12,%eax
movl (%ecx,%eax,2),%ebx
```

2.3 Des exemples de programmes en assembleur

Le but de la section n'est pas de devenir un programmeur de bon niveau en assembleur, mais d'en saisir les mécanismes essentiels pour être en mesure de comprendre l'assembleur produit par le compilateur C.

2.3.1 Interactions entre l'assembleur et le C

Comme c'est un peu compliqué de faire des appels systèmes en assembleur, nous allons utiliser des fonctions en C pour imprimer et pour terminer le programme.

La première fonction en assembleur : accès aux variables globales

Je commence avec une fonction (stupide) `add11` qui ajoute 11 à la variable globale `a`. En C, on la définirait avec

```
extern a;

void
add11(void){
        a += 11;
}
```

Le code assembleur est le suivant (les numéros de lignes n'appartiennent pas au fichier ; ils ne sont là que pour faire référence au code plus facilement)

```
1          // ba-var-asm.s
2          .globl a
3          .globl add11
4
5          .text
6  add11:
7          addl    $11,a
8          ret
```

Le fichier commence par deux directives `.globl` qui indiquent à l'assembleur que `a` et `add11` doivent apparaître dans la table des symboles du fichier résultat pour pouvoir être utilisées dans d'autres fichiers du même programme. A part qu'on ne spécifie pas le type de `a` et `add11`, c'est l'équivalent du C :

```
extern int a; extern void add11(void);
```

La ligne 5 contient aussi une directive, `.text`, pour indiquer que ce sont des instructions qui suivent ; l'éditeur de lien pourra donc les placer dans une zone mémoire que le processus ne pourra pas modifier (parce que les instructions ne sont pas modifiées lors de l'exécution d'un processus).

On a ensuite, ligne 6, la définition d'une étiquette `add11` pour notre fonction, puis sur les deux lignes suivantes les instructions de la fonction : l'instruction de la ligne 6 ajoute la constante 11 à la variable `a` et celle de la ligne 7 effectue un retour de la fonction.

Notre fichier en assembleur n'est pas suffisant pour lancer le programme ; il faut également définir et initialiser la variable `a`, avoir quelque chose qui commence et termine le programme et entre les deux, appeler la fonction et imprimer la valeur de `a`. Je fais ceci avec un fichier C :

```
1  /* ba-var.c
2   Appel d'une fonction en assembleur
3   */
4  # include <stdio.h>
5
6  int a;
7
8  int
9  main(){
10   a = 23;
11   add11();
12   printf("a = %d\n", a);
13   return 0;
14 }
```

On voit que l'appel de la fonction définie en assembleur se fait comme pour une fonction ordinaire.

Pour obtenir un programme exécutable, nous passons les deux fichiers au programme `gcc` ; il va compiler le fichier C et se contenter de traduire le fichier assembleur en langage machine. On peut exécuter le fichier `a.out` résultat.

```
$ gcc -g ba-var.c ba-var-asm.s
$ a.out
a = 34
$
```

Renvoi de valeurs entières de l'assembleur vers le C

Au lieu de modifier une variable globale, il est plus élégant de renvoyer une valeur. Nous modifions en conséquence notre programme C :

```
/* bb-ret.c
 Utiliser une valeur renvoyee de l'assembleur
 */
# include <stdio.h>

int a;

int
main(){
  a = 23;
  printf("a = %d\na + 11 = %d\n", a, add11bis());
  return 0;
}
```

Pour renvoyer une valeur entière, la convention adoptée par le compilateur est que la fonction qui appelle trouvera cette valeur dans le registre `%eax`; la fonction appelée l'y aura placée avant d'effectuer le retour avec `ret`[2]. Le fichier assembleur contient

```
        // bb-var-asm.c
        .globl a
        .globl add11bis

        .text
add11bis:
        movl    a,%eax
        addl    $11,%eax
        ret
```

Ex. 2.2 — Écrire en assembleur une fonction `delta()` qui renvoie $b^2 - 4ac$ calculée avec trois variables globales `a`, `b` et `c`.

Passage d'un argument du C vers l'assembleur

Plutôt que de passer à la fonction assembleur la valeur avec laquelle calculer dans une variable globale, il serait préférable de la lui passer en argument. Le programme C deviendrait alors :

```
/* bc-arg1.c
 Passer un argument a l'assembleur
 */
# include <stdio.h>

```

2. On peut noter que la fonction qui appelle n'a aucune manière de savoir si la fonction appelée a placé ou pas quelque chose dans `%eax`; le registre contient *toujours* une valeur et il n'y a pas de manière de savoir si elle a été placée là délibérément par la fonction appelée ou bien s'il s'agit du résultat d'un calcul intermédiaire précédent.

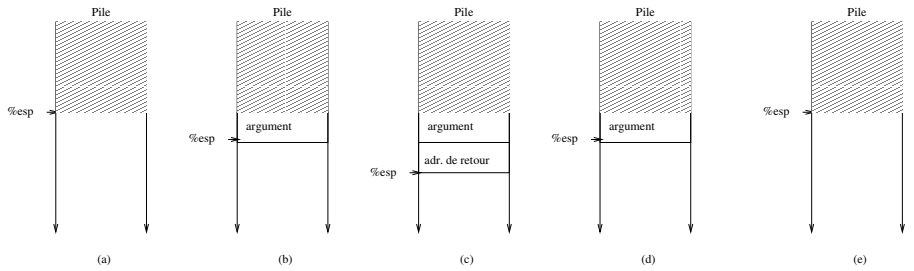


FIGURE 2.1 – Pour passer un argument, la fonction qui appelle l'empile (a), puis exécute l'instruction `call` qui empile par dessus l'adresse de retour (b). La fonction appelée trouve l'argument juste au-dessus de l'adresse de retour (c). Pour finir, la fonction appelée exécute un `ret` qui dépile l'adresse de retour (d). C'est la fonction appelante qui doit dépiler l'argument qu'elle avait empilée pour retrouver la situation initiale de la pile (e).

```
6  int
7  main(){
8    int a = 23;
9
10   printf("a = %d, a + 11 = %d\n", a, add11ter(a));
11   return 0;
12 }
```

Pour passer les arguments, la convention utilisée par le compilateur gcc sur le 386 est de les empiler, sur la pile principale pointée par `%esp`, en dessous de l'adresse de retour[3]. Sur ce processeur, la pile commence dans les adresses hautes et croit vers le bas. Le mécanisme est montré sur la figure 2.1. Le code assembleur sera donc le suivant (attention, ce code ne fonctionne pas sur les processeurs Intel 64 bits) :

```
1          // bc-arg1-asm.s
2          .globl add11ter
3          .text
4  add11ter:
5          movl    4(%esp),%eax
6          addl    $11,%eax
7          ret
```

L'instruction intéressante est celle de la ligne 5, qui récupère l'argument dans `4(%esp)` pour le placer dans `%eax`. La fonction `add11ter` trouve la pile dans

3. Le passage des arguments par la pile est couramment utilisé par de nombreux compilateurs sur beaucoup de processeurs. Une autre convention possible qu'on rencontre aussi fréquemment consiste à les faire passer par des registres, mais dans ce cas les choses se compliquent pour compiler les fonctions qui ont des arguments trop gros pour tenir dans un registre (une structure C) ou qui ont plus d'arguments que le processeur n'a de registres.

l'état décrit dans le schéma (c) de la figure 2.1. Comme la pile croit vers les adresses basses et que l'adresse de retour occupe 4 octets (32 bits), l'argument se trouve 4 octets au-dessus de l'adresse contenue dans le registre `%esp` qui désigne le sommet de la pile et on l'adresse donc avec `4(%esp)`.

Ex. 2.3 — Écrire en assembleur une fonction `delta()` qui renvoie $b^2 - 4ac$ calculée avec trois arguments `a`, `b` et `c`.

Appel d'une fonction C depuis l'assembleur

Puisque nous connaissons les modalités de passage d'un argument dans le langage C, nous pouvons aussi appeler des fonctions C depuis l'assembleur. Ainsi le programme suivant :

```
1          // bd-callc.s
2          .globl main
3
4          .data
5  str:    .asciz "Bonjour tout le monde\n"
6  strbis:.asciz "Impossible\n"
7
8          .text
9  main:
10         pushl   $str
11         call    printf
12         popl    %eax
13
14         pushl   $0
15         call    exit
16         popl    %eax
17
18         pushl   $strbis
19         call    printf
20         .end
```

est à peu près équivalent au programme C

```
int
main(){
        printf("Bonjour tout le monde\n");
        exit(0);
        printf("Impossible\n");
}
```

La fonction `main` est ici écrite en assembleur. Elle empile la chaîne de caractères à imprimer, appelle la fonction `printf` puis dépile la chaîne[4]. Elle appelle en-

4. L'instruction `popl %eax` dépile l'adresse de la chaîne dans le registre `%eax`, ce qui est sans intérêt. En revanche elle se code sur deux octets alors que l'instruction normale pour ôter

suite la fonction `exit` après avoir empilé la constante 0 [5]. Les trois instructions suivantes ne sont normalement pas exécutées, puisque `exit` tue le processus et ne revient donc pas.

2.3.2 Tests

La principale différence entre l'assembleur et les langages de programmation usuels est la manière dont on effectue des tests en deux étapes. La plupart des instructions qui modifient les données positionnent dans le registre `%eflags` des bits qui indiquent si le résultat vaut 0, s'il est négatif, s'il y a eu une retenue, etc. Pour effectuer un test, on fait suivre ces instructions d'un saut conditionnel en fonction de l'état des bits de `%eflags`. Par exemple, au lieu d'écrire en C :

```
if (a > 0){
        b += 1;
        a -= 1;
}
```

On pourra écrire en assembleur

```
        andl    a,a    // tester la valeur de a
        jle     next    // si a <= 0, sauter les deux
                        //      instructions suivantes
        incl    b       // incrémenter b
        decl    a       // décrémenter a
next:
```

On commence par positionner les bits du registre `%eflags` en fonction de la valeur de `a` en calculant `a & a` ; la valeur de `a` ne sera pas modifiée et les bits de `%eflags` seront positionnés. Ensuite, si le résultat de la dernière opération était inférieur ou égal à 0 (c'est à dire si `a` n'était pas strictement positif), on saute sur l'étiquette `next`, ce qui nous évite d'exécuter les deux instructions suivantes qui incrémentent `b` et décrémentent `a`.

Ex. 2.4 — Écrire en assembleur une fonction `médian3` qui reçoit trois arguments et renvoie celui qui n'est ni le plus grand, ni le plus petit. La fonction est utile dans Quick Sort pour choisir le pivot entre le premier élément, le dernier et celui du milieu.

2.3.3 Boucles

Souvent les processeurs n'ont pas d'instructions spéciales pour les boucles. (Le processeur 386 dispose d'instructions spécifiques pour les boucles, présentées

---

quelque chose de la pile `addl $4,%esp` a besoin d'un octet pour coder l'instruction *plus* quatre octets pour coder la constante 4 sur 32 bits.

5. Comme dans l'exemple précédent, on pourrait utiliser les deux instructions `xorl %eax %eax ; pushl %eax` pour n'utiliser que quatre octets au lieu des cinq qui sont nécessaires ici.

sommairement plus haut, mais leur usage n'est souvent pas aisé à cause des contraintes qu'elles imposent sur l'utilisation des registres.) On se contente de combiner un test et un branchement conditionnel vers le début du corps de la boucle. Ainsi, une boucle qui multiplie les dix premiers entiers non nuls entre eux :

```
for(prod = i = 1; i <= 10; i++)
  prod *= i;
```

sera traduite en assembleur par du code organisé comme les instructions C :

```
        prod = i = 1;
redo:
        if (i > 10)
                goto out;
        prod *= i;
        i += 1;
        goto redo;
out:
        ;
```

Cela donne en assembleur 386, en plaçant `prod` dans `%eax` et `i` dans `%ebx` :

```
        movl    $1,%ebx
        movl    %ebx,%eax
redo:
        cmpl    $10,%ebx
        jg      out
        imull   %ebx,%eax
        incl    %ebx
        jmp      redo
out:
```

Notons qu'on peut retirer une instruction au corps de la boucle en la réorganisant ; il suffit d'inverser le test et le placer à la fin ; on n'a plus de cette manière qu'une seule instruction de saut dans la boucle.

```
        movl    $1,%ebx
        movl    %ebx,%eax
        jmp      in
redo:
        imull   %ebx,%eax
        incl    %ebx
in:
        cmpl    $10,%ebx
        jle     redo
```

Ex. 2.5 — Traduire en assembleur la fonction `indexa` suivante, qui renvoie l'adresse du premier a dans une chaîne (ou 0 s'il n'y est pas).

```
char *
indexa(char string[]){
        int i;

        for(i = 0; string[i] != 0; i++)
                if (string[i] == 'a')
                        return & string[i];
        return 0;
}
```

ou bien :

```
char * indexa(char * p){
        for(; *p; p++)
                if (*p == 'a')
                        return p
        return 0;
}
```

Ex. 2.6 — Écrire une fonction `rindexa` qui renvoie l'adresse du *dernier* caractère 'a' dans la chaîne.

Ex. 2.7 — Écrire en assembleur une fonction équivalente à la fonction strlen. (Rappel : pour n'accéder qu'à un seul octet, il faut postfixer l'instruction avec `b` (comme *byte*) au lieu de `l` (comme *long*)).

Ex. 2.8 — Traduire la fonction suivante du C vers l'assembleur

```
int
fact(int n){
        int r;

        for(r = 1; n > 1; n--)
                r *= n;
        return r;
}
```

2.3.4 Pile

La pile est une zone de mémoire ordinaire sur laquelle pointe le registre `%esp`. Les instructions `call` et `ret` l'utilisent pour empiler et dépiler les adresses de retour lors des appels de fonctions. On a vu également qu'elle servait à passer les arguments dans le code généré par le compilateur C. La pile peut également servir à stocker les variables locales quand elles sont trop nombreuses ou trop grosses pour être stockées dans des registres ou bien quand le code appelle une

fonction qui pourrait modifier le contenu des registres.

Voici par exemple une fonction qui calcule récursivement la factorielle de son argument :

```
1         .globl  factR
2         .text
3 factR:
4         cmpl    $0,4(%esp)
5         jne     cont
6         movl    $1,%eax
7         ret
8 cont:
9         movl    4(%esp),%eax
10        decl    %eax
11        pushl   %eax
12        call    factR
13        addl    $4,%esp
14        imull   4(%esp),%eax
15        ret
```

Elle traduit le code C

```
1         int
2         factR(int n){
3           if (n != 0)
4             return n;
5           else
6             return n * factR(n - 1);
7         }
```

Ex. 2.9 — Traduire la fonction suivante du C vers l'assembleur

```
int
fib(int n){
        if (n < 2)
                return n;
        return fib(n - 1) + fib(n - 2);
}
```

Ex. 2.10 — Traduire l'assembleur suivant en C

```
        .text
        .globl  heron
heron:
        pushl   %ebx
        movl    8(%esp),%eax
        movl    12(%esp),%ebx
        movl    16(%esp),%ecx
        movl    %eax,%edx
```

```
addl    %ebx,%eax
addl    %ecx,%eax
sarl    $1,%eax
subl    %eax,%ebx
subl    %eax,%ecx
subl    %eax,%edx
imull   %ebx,%eax
imull   %ecx,%eax
imull   %edx,%eax
negl    %eax
popl    %ebx
ret
```

2.4 Les autres assembleurs, les autres processeurs

On trouve souvent plusieurs assembleurs pour un même processeur. La différence la plus frappante est l'ordre dans lequel apparaissent les opérandes. Dans l'assembleur que nous avons vu, pour l'instruction `mov`, on a la source en premier et la destination en second. D'autres assembleurs utilisent la convention inverse avec la destination en premier et la source en second, dans le même ordre que quand on écrit une affectation en langage évolué comme `a = b` en C.

Chaque processeur possède ses caractéristiques propres, mais les principes de base restent presque tous les mêmes. Il n'est pas bien difficile, quand on a pratiqué un peu l'assembleur, de s'adapter suffisamment à un nouveau processeur pour être en mesure de déchiffrer le code produit par le compilateur. En revanche, coder efficacement en assembleur nécessite une connaissance approfondie de la manière dont il code les instructions et du temps nécessaire pour effectuer chacune d'entre elles.

2.4.1 Parenthèse : les machines RISC

Pendant longtemps, ce qui a limité la vitesse de calcul des processeurs, c'est le temps d'accès à la mémoire, qui croissait beaucoup moins vite que la vitesse de calcul des processeurs. On appelait ce point (le bus qui relie le processus à la mémoire) le *goulot d'étranglement de von Neumann* (en anglais *von Neumann bottleneck*). On évaluait même grossièrement la vitesse d'un processeur simplement par la taille d'un programme exécutable.

Les architectes de processeurs ont longtemps cherché à éviter le problème en fabriquant des instructions élémentaires le plus compactes possible, afin de diminuer le nombre d'échanges entre la mémoire et le microprocesseur. Un exemple typique est l'instruction `rep cmps` du 386 qui, alors qu'elle est codée sur deux octets seulement, incrémente ou décrémente deux registres (`%esi` et `%edi`), en décrémente un troisième (`%ecx`), effectue une comparaison et recommence jusqu'à trouver deux octets différents.

Au milieu des années 1980, deux équipes universitaires californiennes et une équipe d'IBM ont réalisé que ce genre d'instruction de longueur variable compliquait énormément la partie contrôle du processeur, qui de ce fait utilisait une grande partie de la surface disponible dans le circuit intégré ; d'autre part les conditions d'utilisation de ces instructions étaient si complexes que les compilateurs avaient rarement l'occasion de les utiliser. Ils ont mis au point des ordinateurs avec des instructions beaucoup plus simples qu'ils ont appelé des ordinateurs à jeu d'instruction réduit (en anglais *Reduced Instruction Set Computer*, d'où vient le nom *RISC*). La caractéristique principale en est que toutes les instructions occupent le même nombre d'octets, en général la taille d'un mot mémoire. Les instructions de transfert entre les registres et la mémoire ne font que du transfert ; les instructions de calcul n'opèrent que sur des registres.

La simplification obtenue dans l'unité de contrôle libère de la place pour disposer d'un plus grand nombre de registres et permet d'avoir des mémoires caches sur le processeur, ce qui limite l'impact de la lenteur de l'accès à la mémoire. On peut avoir une exécution en pipe-line (l'instruction suivante commence à être exécutée avant que l'instruction courante ne soit terminée) et de l'exécution spéculative (on commence à exécuter l'instruction qui suit un branchement ; si le branchement est pris, on ne stocke pas son résultat ; sinon le calcul est déjà effectué et on peut utiliser le résultat de suite).

Chapitre 3

Comprendre un programme C compilé

Dans ce chapitre, nous approfondissons notre approche de l'assembleur en examinant la manière dont le compilateur traduit le C en langage machine. Le point crucial ici réside dans le fonctionnement du prologue et de l'épilogue des fonctions et le chapitre se termine par quelques applications que la compréhension du mécanisme nous permet.

3.1 Du C vers l'assembleur

Nous allons considérer ici une fonction `pgcd` écrite en C. La fonction calcule le plus grand commun diviseur de deux nombres positifs non nuls qu'elle reçoit en argument, seulement avec des soustractions, sans aucune division. Le principal intérêt de cette fonction ici est qu'elle est simple, mais se compose d'une boucle qui contient elle-même un test.

Le code C de la fonction est le suivant :

```
1  int
2  pgcd(int a, int b){
3          int t;
4
5          while(a != 0){
6                  if (a < b){
7                          t = b;
8                          b = a;
9                          a = t;
10                 }
11                 a -= b;
12         }
```

```
13          return b;
14  }
```

On peut se représenter le travail de cette fonction sous forme graphique comme dans la figure 3.1.

Au lieu de produire du langage machine, difficile à déchiffrer, on peut demander au compilateur avec l'option `-S` de produire de l'assembleur que nous pourrons lire plus facilement ; si le fichier de départ s'appelle `pgcd.c`, l'assembleur sera posé dans un fichier `pgcd.s`. Sur une machine Intel 32 bits, j'obtiens le fichier suivant (avec les numéros de lignes rajoutés par mes soins pour référence) :

```
 (1)            .file   "pgcd.c"
 (2)            .text
 (3)             .globl pgcd
 (4)            .type   pgcd, @function
 (5)    pgcd:
 (6)            pushl   %ebp
 (7)            movl    %esp, %ebp
 (8)            subl    $16, %esp
 (9)            jmp     .L2
(10)    .L4:
(11)            movl    8(%ebp), %eax
(12)            cmpl    12(%ebp), %eax
(13)            jge     .L3
(14)            movl    12(%ebp), %eax
(15)            movl    %eax, -4(%ebp)
(16)            movl    8(%ebp), %eax
(17)            movl    %eax, 12(%ebp)
(18)            movl    -4(%ebp), %eax
(19)            movl    %eax, 8(%ebp)
(20)    .L3:
(21)            movl    12(%ebp), %eax
(22)            subl    %eax, 8(%ebp)
(23)    .L2:
(24)            cmpl    $0, 8(%ebp)
(25)            jne     .L4
(26)            movl    12(%ebp), %eax
(27)            leave
(28)            ret
(29)            .size   pgcd, .-pgcd
(30)            .ident  "GCC: (Debian 4.3.2-1.1) 4.3.2"
(31)            .section        .note.GNU-stack,"",@progbits
```

Dans ce paquet de ligne en assembleur, le plus facile à repérer est sans doute les lignes qui correspondent aux trois affectations des lignes 7 à 9 du fichier

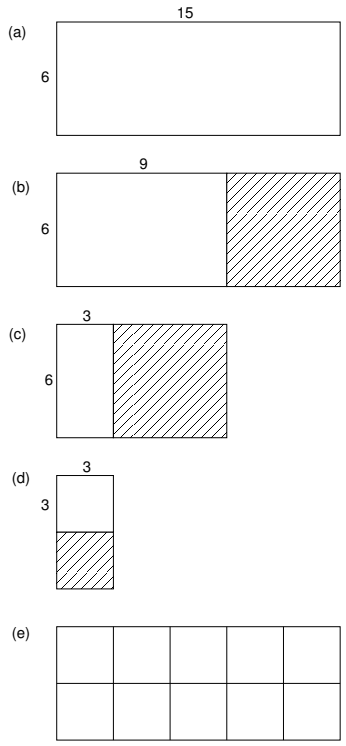


FIGURE 3.1 – La recherche du pgcd de deux nombres a et b revient à chercher le plus grand carré qui permet de paver exactement le rectangle $a \times b$. En (a), on a un rectangle 15×6 en guise d'exemple ; on voit que si on enlève le plus grand carré possible de ce rectangle, comme dans (b), on obtient un carré 9×6 et que le plus grand carré qui permettra de paver la partie restante sera aussi celui qui permettra de paver le rectangle de départ. On recommence à retirer le plus grand carré possible pour obtenir un rectangle 3×6 en (c) puis 3×3 en (d). Puisqu'on a maintenant un carré, on a obtenu la réponse. Le retrait du plus grand carré possible correspond à la soustraction de la ligne 11 de la fonction.

C. On les reconnaît assez facilement dans l'assembleur entre les lignes 14 à 20. On peut supposer que le compilateur n'a pas modifié l'ordre des affectations et puisque ces lignes déplacent, avec deux `movl`, ce qu'il y a dans `12(%ebp)` dans `-4(%ebp)`, puis ce qu'il y a dans `8(%ebp)` vers `12(%ebp)` et finalement ce qui se trouve dans `-4(%ebp)` vers `8(%ebp)`, on a identifié l'endroit où sont stockés les arguments `a` (dans `12(%ebp)`) et `b` (dans `8(%ebp)`) ainsi que la variable locale `t` (dans `-4(%ebp)`).

On peut continuer en regardant les lignes précédentes (11 à 13) de l'assembleur. Elles nous confirment dans notre supposition : il s'agit de comparer la valeur de `a` et de `b` puis de sauter par dessus les trois affectations jusqu'à l'étiquette `.L3` si `a` est supérieur ou égal à `b`, c'est à dire si le test de la ligne 6 du fichier C est faux.

Enfin les lignes 21 et 22 du fichier en assembleur traduisent la soustraction de la ligne 11 du fichier C.

On a donc le corps de la boucle `while` du programme C dans le bloc des lignes assembleur 11 à 22.

Les lignes 9–10 et 24–25 de l'assembleur traduisent clairement la logique de la boucle `while` : la ligne 9 branche directement sur le test de la ligne 21 ; la ligne 21 compare la variable `a` avec 0 et si on a quelque chose de différent, le `jne` de la ligne 25 branche sur l'étiquette `.L4` qui précède le corps de la boucle.

Il ne reste donc à analyser que les lignes 1–4 et 29–31 qui contiennent des directives (dont deux que nous avons vues au chapitre précédent `.text` et `.globl` ; je vous laisse le soin de deviner le rôle des autres), ainsi que le prologue de la fonction aux lignes 6–9 et son épilogue aux lignes 27–28.

Vois le code généré avec des commentaires ajoutés par mes soins :

```
 (1)            .file   "pgcd.c"
 (2)            .text
 (3)             .globl pgcd
 (4)            .type   pgcd, @function
 (5)    pgcd:
 (6)            pushl   %ebp            // prologue de la fonction
 (7)            movl    %esp, %ebp
 (8)            subl    $16, %esp
 (9)            jmp     .L2             // va au test de la boucle
(10)    .L4:
(11)            movl    8(%ebp), %eax   // si a >= b
(12)            cmpl    12(%ebp), %eax
(13)            jge     .L3             //   pas d'affectations
(14)            movl    12(%ebp), %eax  // t = a;
(15)            movl    %eax, -4(%ebp)
(16)            movl    8(%ebp), %eax   // a = b;
(17)            movl    %eax, 12(%ebp)
(18)            movl    -4(%ebp), %eax  // b = t;
```

```
(19)            movl    %eax, 8(%ebp)
(20)    .L3:
(21)            movl    12(%ebp), %eax  // a -= b;
(22)            subl    %eax, 8(%ebp)
(23)    .L2:                            // test de la boucle
(24)            cmpl    $0, 8(%ebp)     // si a != 0
(25)            jne     .L4             //   recommencer
(26)            movl    12(%ebp), %eax  // return b;
(27)            leave                   // épilogue de la fonction
(28)            ret
(29)            .size   pgcd, .-pgcd
(30)            .ident  "GCC: (Debian 4.3.2-1.1) 4.3.2"
(31)            .section        .note.GNU-stack,"",@progbits
```

3.2 Prologue et épilogue des fonctions C

En plus du pointeur sur le sommet de la pile, les fonctions C utilisent un registre qui pointe de façon permanente (pendant l'exécution de la fonction) sur un point fixe dans la zone de la pile utilisée par la fonction. On appelle couramment ce registre le *frame pointeur* (prononcer comme *frème pohïneteur*; je crois qu'en français on devrait appeler cela le *pointeur sur l'enregistrement d'activation*). Par défaut, notre compilateur utilise le registre `%ebp` du 386 comme frame pointer

3.2.1 Le prologue

Au début de la fonction, le sommet de pile pointe sur l'adresse de retour, avant laquelle les arguments ont été empilés, en commençant par le dernier; le frame pointer peut pointer n'importe où, mais le plus probable est qu'il contient une adresse dans la pile. Ceci est évoqué par la figure 3.2

Dans le prologue, la fonction va s'installer sur la pile. Elle commence par sauver la valeur du registre `%ebp`, puis elle fait pointer `%ebp` sur la valeur sauvegardée (aux lignes 6 et 7), comme dans la figure 3.3.

Finalement, la fonction va réserver sur la pile l'espace nécessaire pour sauver les registres qui ont besoin de l'être, pour les variables locales et les expressions temporaires. Ici, la ligne 8 de l'assembleur réserve 16 octets (je ne sais pas pourquoi il réserve autant de place alors que seuls 4 octets sont utilisés pour la variable `t`; je soupçonne que c'est en espérant que le frame de la fonction suivante commencera sur une adresse qui correspondra au début d'une ligne de cache). L'état de la pile est maintenant celui de la figure 3.4.

Il existe une instruction `enter` qui effectue ces trois étapes du prologue de la fonction en une seule instruction. Je ne sais pas pourquoi le compilateur gcc ne l'utilise pas. (Il me semble avoir lu quelque part il y longtemps qu'à la suite

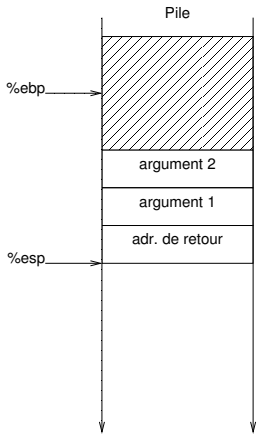


FIGURE 3.2 – A l'entrée dans la fonction, la pile contient l'adresse de retour sur le sommet de pile et dessous les arguments, empilés avec le premier argument le plus près du sommet. Le frame pointer peut pointer n'importe où.

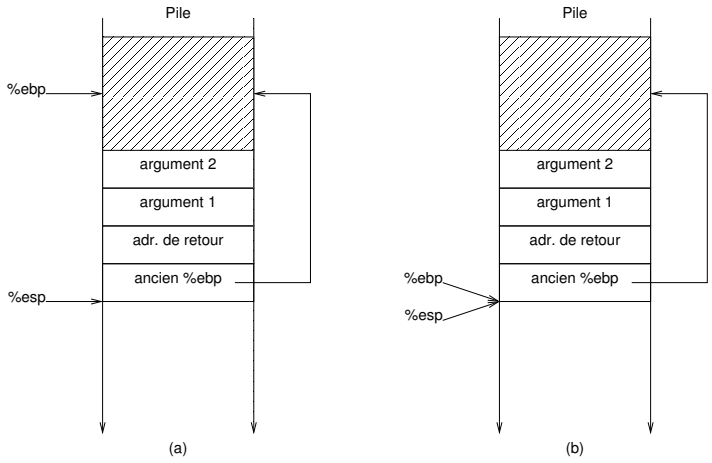


FIGURE 3.3 – Le prologue commence par empiler la valeur du frame pointer (a), puis fait pointer le frame pointer sur la valeur qu'il vient de sauver.

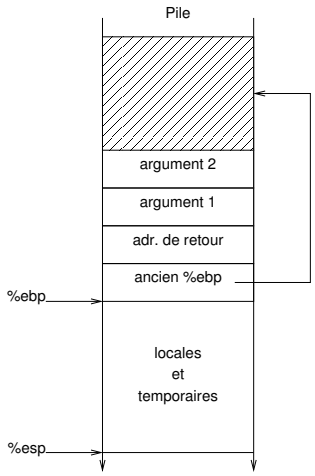


FIGURE 3.4 – Le prologue de la fonction se termine par la réservation d'espace sur le sommet de pile pour les variables locales et les expressions temporaires.

d'une erreur de conception, l'exécution de cette instruction est plus lente que celle des trois instructions équivalentes.)

3.2.2 L'épilogue

Lorsque la fonction a terminé son travail, elle entre dans son épilogue pour remettre la pile dans l'état où elle l'a trouvé en entrant. Dans le fichier assembleur produit par le compilateur, l'épilogue correspond aux lignes 27–28.

L'instruction `leave` de la ligne 27 est équivalente aux deux instructions

```
(27.1)          movl    %ebp,%esp
(27.2)          popl    %ebp
```

Elle dépile tout ce que la fonction a pu empiler au dessus de la sauvegarde de l'ancien frame pointer puis dépile et réinstalle l'ancien frame pointer. Finalement, le `ret` de la ligne 28 dépile l'adresse de retour et rend la main à la fonction qui a appelée. Ceci est montré sur la figure 3.5.

3.2.3 L'intérêt du frame pointer

Le frame pointer permet d'avoir un repère fixe sur la zone de la pile qu'utilise un appel de la fonction. Il est installé au prologue et reste en place jusqu'à l'épilogue. Le sommet de pile, lui, peut varier au cours de l'exécution de la fonction, à mesure qu'on empile et qu'on dépile des arguments pour appeler d'autres fonctions. Cela garantit qu'on peut trouver les arguments de la fonction

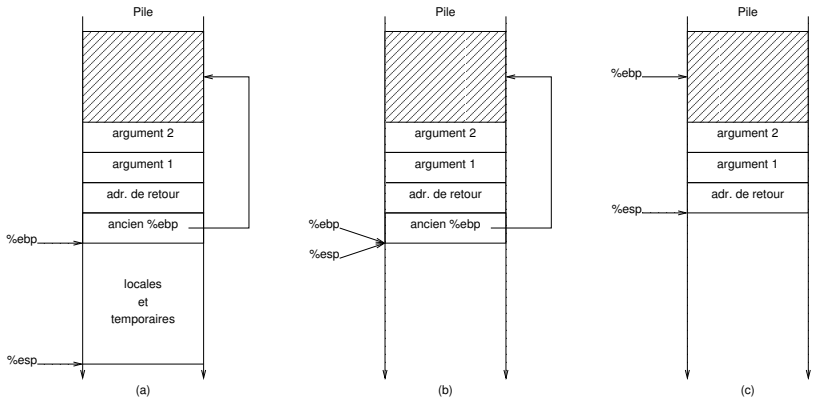


FIGURE 3.5 – L'épilogue de la fonction démarre avec la pile comme en (a) ; elle dépile les locales et les temporaires en recopiant le frame pointer dans le pointeur de pile (b), dépile l'ancien frame pointer (c) puis l'adresse de retour avec l'instruction `ret` (cette étape n'est pas représentée sur la figure).

courante, ses variables locales et ses temporaires à une distance qui ne varie pas de l'endroit où pointe le frame pointer.

Surtout, le frame pointer permet de remonter dans la chaîne des appels de fonction. C'est grâce à lui que le débugger nous indique non seulement quelle est la fonction courante (ce qu'il peut découvrir en regardant l'adresse contenue dans `%eip`, le compteur ordinal), mais aussi quelle fonction a appelé la fonction courante, et quelle fonction a appelé cette fonction, et ainsi de suite. La valeur courante du frame pointer pointe juste au-dessus de l'adresse de retour, qui permet de savoir quelle était la fonction appelante, De plus il pointe sur la sauvegarde du frame pointer de la fonction appelante, ce qui permet de connaître sa propre adresse de retour (figure 3.6).

Finalement, dans les langages de programmation qui autorisent les définitions de fonctions imbriquées, le frame pointer peut être le moyen le plus pratique de retrouver les variables locales d'une fonction enveloppante. Imaginons le fragment de code suivant :

```
(1)    void
(2)    foo(int n){
(3)      int a = 23;
(4)
(5)      void bar(int n){
(6)        if (n == 0)
(7)          a += 1
(8)        else
(9)          bar(n - 1);
```

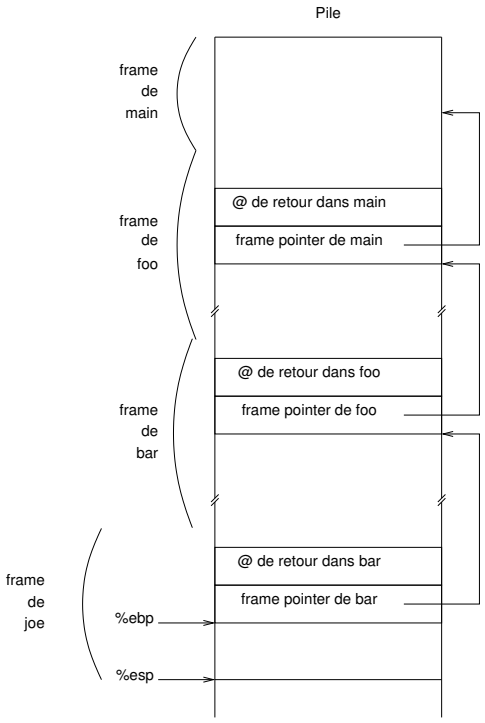


FIGURE 3.6 – L'état de la pile quand la fonction `main` a appelé la fonction `foo` qui a appelé la fonction `bar` qui elle même à appelé la fonction `joe`. La sauvegarde du frame pointer par chacune des fonctions permet de remonter la chaîne des appels.

```
(10)      }
(11)      ...
(12)      bar(n);
(13)      ...
(14)    }
```

La fonction `bar` modifie `a`, une variable locale de `foo` qui existe donc dans la pile dans le frame de la fonction `foo`. Au moment où il examine le code, le compilateur ne peut en aucune manière savoir à quelle distance dans la pile se trouve cette variable, puisque cela dépend du nombre d'appels récursifs de la fonction `bar`, qui dépend lui-même d'un argument inconnu au moment de la compilation. Pour résoudre le problème, le compilateur doit fabriquer du code qui va remonter de frame en frame dans la pile jusqu'à arriver à celui de l'appel de la fonction `foo` ; à ce moment, la variable `a` se trouvera à une distance fixe connue au moment de la compilation du frame pointer de la fonction `foo`.

On peut demander à `gcc` de ne pas utiliser de frame pointer avec l'option `-fomit-frame-pointer`. On a un registre disponible de plus pour nos variables, mais on a aussi la garantie que le code produit sera parfaitement impossible à mettre au point.

3.3 Le code optimisé

Si on compile la même fonction `pgcd` avec l'option `-O` en plus de `-S`, on voit l'assembleur qui correspond au code optimisé :

```
 (1)            .file   "pgcd.c"
 (2)            .text
 (3)             .globl pgcd
 (4)            .type   pgcd, @function
 (5)    pgcd:
 (6)            pushl   %ebp
 (7)            movl    %esp, %ebp
 (8)            movl    8(%ebp), %edx
 (9)            movl    12(%ebp), %ecx
(10)            testl   %edx, %edx
(11)            je      .L2
(12)    .L6:
(13)            cmpl    %ecx, %edx
(14)            jge     .L3
(15)            movl    %ecx, %eax
(16)            movl    %edx, %ecx
(17)            movl    %eax, %edx
(18)    .L3:
(19)            subl    %ecx, %edx
(20)            jne     .L6
```

```
(21)    .L2:
(22)            movl    %ecx, %eax
(23)            popl    %ebp
(24)            ret
(25)            .size   pgcd, .-pgcd
(26)            .ident  "GCC: (Debian 4.3.2-1.1) 4.3.2"
(27)            .section        .note.GNU-stack,"",@progbits
```

On voit (aux lignes 15–17) que les arguments `a` et `b` et la variable `t` sont maintenant stockés dans les registres `%edx`, `%ecx` et `%eax`. (Gcc commet d'ailleurs ici une maladresse : s'il plaçait la variable `b` dans le registre `%eax` au lieu de `%ecx`, l'instruction de la ligne 22 serait inutile.)

Puisque la soustraction `a -= b` de la fonction C est traduite par l'instruction de la ligne 19, qui positionne les flags, le test de la boucle a disparu ; on se contente de boucler à la ligne 20 si le résultat de la soustraction (la nouvelle valeur de `a`) est différent de 0. Il faut cependant prendre garde au cas où la fonction ne fait aucun tour de boucle : c'est le rôle des instructions des lignes 10–11.

Dans l'épilogue, on voit que l'instruction `leave` a été remplacée par un simple `popl %ebp`, puisque le pointeur de pile pointe encore sur l'ancien frame pointer.

Ex. 3.1 — En compilant avec l'option `-O3`, on active d'autres optimisations. Examiner le code assembleur produit et identifier lesquelles.

Ex. 3.2 — Examiner le code produit par le compilateur quand il compile la fonction

```
void
foo(){
        if (0 == 1)
                printf("Hello\n"); // jamais exécuté
}
```

L'appel a la fonction `printf` est-il encore dans le code ? Et la chaîne de caractère ? (Attention, le résultat peut dépendre du niveau d'optimisation.)

Ex. 3.3 — Même question pour

```
void
foo(int x){
        if (x == x + 1)
                printf("Hello\n"); // jamais exécuté
}
```

Ex. 3.4 — Même question pour

```
void
foo(){
        for(;;)
```

```
                bar();
        printf("Hello\n");       // jamais exécuté
}
```

Ex. 3.5 — Même question pour

```
int
bar(void){
        return 1;
}

void
foo(){
        if (bar() != 1)
                printf("Hello\n"); // jamais exécuté
}
```

3.4 L'utilisation des registres

On a déja vu, plus haut dans le chapitre, que les variables ne sont placées dans des registres que si on a compilé avec l'optimisation. Ici on examine la question de la sauvegarde des registres lors des appels de fonction.

3.4.1 Variables et expressions intermédiaires dans les registres

Une fonction sera plus rapide si elle utilise les registres pour stocker les variables temporaires et les expressions intermédiaires, plutôt que de les placer dans la pile. En effet d'une part l'accès aux registres est plus rapide qu'à la mémoire de plusieurs ordres de grandeur et d'autre part les instructions qui référencent les registres sont en général plus compactes (et donc plus rapides à charger depuis la mémoire ou le cache) que celles qui font référence à la mémoire.

Sur les huit registres dits généraux du 386, deux ont déjà des rôles affectés (le pointeur de pile `%esp` et le pointeur de frame `%ebp`). Il ne reste donc que six registres disponibles.

3.4.2 La problématique de la sauvegarde

Quand une fonction appelle une autre fonction, il est nécessaire de sauvegarder les registres qui sont utilisés par la fonction qui appelle et qui vont être utilisés par le fonction appelée (ou par une fonction qu'elle appelle). Le problème est que le compilateur C compile fonction par fonction (les fonctions peuvent se trouver dans des fichiers différents et être compilées par des appels différents au compilateur) et ne dispose pas de toute l'information nécessaire. Il faut

pourtant, avec l'information partielle dont il dispose, choisir de faire sauver les registres soit par la fonction qui appelle, soit par la fonction qui est appelée. (On appelle cela *caller-save* et *callee-save* en anglais).

Sauvegarde par la fonction appelée

Si les registres sont sauvegardés par la fonction appelée, elle pourra le faire dans son prologue et remettre les anciennes valeurs dans son épilogue. De plus, on n'a besoin de sauver que les registres dont le compilateur sait qu'ils seront utilisés par cette fonction. Le problème est qu'on risque de sauver (inutilement) des registres que la fonction qui appelle n'utilise pas.

Sauvegarde par la fonction qui appelle

Si les registres sont sauvegardés par la fonction qui appelle, on ne sauvera que les registres effectivement utilisés avant chaque appel et on remettra les valeurs après l'appel. Le problème est qu'on risque de sauver (inutilement) des registres que la fonction appelée n'utilise pas.

Des solutions

Le premier compilateur C affectait au plus trois registres (sur les huit disponibles sur le processeur PDP11, y compris le frame pointer et le pointeur de pile), uniquement aux variables locales qui avaient été déclarées avec le mot clef `register`.

La solution adoptée par Gcc consiste à diviser les six registres disponibles en deux groupes de trois, les uns à sauver par l'appelante et les autres à sauver par l'appelée.

Les registres `%eax`, `%ecx` et `%edx` sont utilisables par la fonction appelée sans précaution particulière ; si elle y stocke une valeur qu'il faut conserver lors d'un appel de fonction, le contenu du registre sera sauvegardé avant l'appel et sera récupéré après.

Les registres `%ebx`, `%esi` et `%edi` seront utilisés pour une fonction qui a beaucoup de variables locales, mais une fonction qui les utilise sauvera leur ancienne valeur dans son prologue et la remettra en place dans son épilogue.

On peut s'en rendre compte en compilant avec `-O -S` la fonction suivante :

```
foo(int a, int b, int c, int d, int e, int f, int g){
        bar(a, b, c, d, e, f, g);
        bar(a, b, c, d, e, f, g);
}
```

Le code produit est le suivant :

```
 (1)            .file   "u.c"
```

```
 (2)            .text
 (3)    .globl foo
 (4)            .type   foo, @function
 (5)    foo:
 (6)            pushl   %ebp
 (7)            movl    %esp, %ebp
 (8)            subl    $40, %esp
 (9)            movl    %ebx, -12(%ebp)
(10)            movl    %esi, -8(%ebp)
(11)            movl    %edi, -4(%ebp)

(12)            movl    24(%ebp), %edi
(13)            movl    28(%ebp), %esi
(14)            movl    32(%ebp), %ebx

(15)            movl    %ebx, 24(%esp)
(16)            movl    %esi, 20(%esp)
(17)            movl    %edi, 16(%esp)
(18)            movl    20(%ebp), %eax
(19)            movl    %eax, 12(%esp)
(20)            movl    16(%ebp), %eax
(21)            movl    %eax, 8(%esp)
(22)            movl    12(%ebp), %eax
(23)            movl    %eax, 4(%esp)
(24)            movl    8(%ebp), %eax
(25)            movl    %eax, (%esp)
(26)            call    bar

(27)            movl    %ebx, 24(%esp)
(28)            movl    %esi, 20(%esp)
(29)            movl    %edi, 16(%esp)
(30)            movl    20(%ebp), %eax
(31)            movl    %eax, 12(%esp)
(32)            movl    16(%ebp), %eax
(33)            movl    %eax, 8(%esp)
(34)            movl    12(%ebp), %eax
(35)            movl    %eax, 4(%esp)
(36)            movl    8(%ebp), %eax
(37)            movl    %eax, (%esp)
(38)            call    bar

(39)            movl    -12(%ebp), %ebx
(40)            movl    -8(%ebp), %esi
(41)            movl    -4(%ebp), %edi
(42)            movl    %ebp, %esp
(43)            popl    %ebp
```

```
(44)            ret
(45)            .size   foo, .-foo
(46)            .ident  "GCC: (Debian 4.3.2-1.1) 4.3.2"
(47)            .section        .note.GNU-stack,"",@progbits
```

Le prologue ici s'étend de la ligne 6 à la ligne 11 : après avoir installé le frame pointer et réservé de l'espace dans la pile, le contenu des trois registres `%ebx`, `%esi` et `%edi` est sauvé. Ensuite les lignes 12–14 récupèrent trois des arguments et les placent dans ces registres pour accélérer leur empilage aux lignes 15–17 (pour le premier appel à `bar`) et 27–29 (pour le second appel à `bar`).

Les autres arguments sont récupérés dans la pile ; ce n'est pas la peine d'en stocker dans les registres `%eax`, `%ecx` et `%edx` puisque la fonction `bar` est susceptible de modifier leur contenu.

Finalement, l'épilogue commence, aux lignes 39–41, par récupérer les anciennes valeurs de ces registres sauvées lors du prologue.

Notons au passage que gcc réserve la pile nécessaire dès le prologue, ce qui permet d'utiliser des déplacements simples au lieu des empilements avant l'appel de la fonction `bar` et un dépilement après. A la ligne 8, il réserve bien les 12 octets nécessaires pour la sauvegarde des registres plus les 28 nécessaires pour les sept arguments de bar.

Ex. 3.6 — La fonction `foo` n'a pas besoin d'avoir sept arguments pour mettre la particularité de gcc en évidence. Quel est le nombre minimum d'arguments ? S'il n'y a qu'un seul argument, ou est-il placé ?

On trouvera un tour d'horizon de différentes conventions d'appels de fonction et d'appels de registres sur wikipedia à la page `http://en.wikipedia.org/wiki/X86_calling_conventions` (en anglais). Il semble notamment que les API d'un système d'exploitation assez répandu sur PC utilisent une autre convention qu'on peut demander à gcc d'utiliser avec l'attribut `__stdcall`.

Dans les sections suivantes, je montre quelques exemples de ce que la connaissance intime du fonctionnement des appels de fonction sur notre processeur permet au niveau de la programmation en C.

3.5 Application : le fonctionnement et le contournement de `stdarg`

Le fichier d'inclusion `<stdargs.h>` permet d'écrire des fonctions qui peuvent être utilisées avec un nombre variables d'arguments à la manière de `printf`. Nous commençons par étudier le passage des arguments autres que des entiers, de taille variable.

3.5.1 Passage de paramètres de types `double`

On peut forcer le compilateur à passer des arguments qui ne tiennent pas dans un `int` en utilisant des arguments du type `double` qui font 8 octets. Noter que les `float`s sont convertis en `double`s pour le passage en argument, comme les `char` sont convertis en `int`. Le bout de code :

```
foobar(1.111111111111)
```

est traduit en

```
pushl $1072809756  // empiler la moitié de l'argument
pushl $1908873853  // empiler l'autre moitié
call foobar        // appeler la fonction
addl $8,%esp       // dépiler l'argument (8 octets)
```

3.5.2 Passages de paramètres de types variés

En passant des structures en arguments, on peut passer des objets dont on peut contrôler finement la taille. Ainsi, avec le code suivant :

```
struct {
    char t[NCAR];
} x;

int toto;

foo(){
    toto = sizeof x;
    foobar(x);
    foobar(x, x);
}
```

on peut faire varier la taille utile de la structure en faisant varier la constante `NCAR`, connaître sa taille effective en regardant la valeur affectée à la variable `toto`, et observer le passage en argument avec un seul argument, et avec deux arguments. (Je souligne encore une fois que le compilateur traduit un appel de fonction, sans rien savoir sur le nombre et le type des arguments que la fonction attend. Quand nous déclarons des prototypes, c'est pour *demander* au compilateur de nous fournir des messages d'erreurs quand nous appelons une fonction avec des arguments qu'elle n'est pas prévue pour recevoir ; la fonction `foobar` n'est pas définie et nous n'avons pas défini son prototype, donc le compilateur ne sait rien à son sujet.

Si on place quatre octets dans la structure en compilant avec

```
gcc -S -DNCAR=4 file.c
```

on obtient l'assembleur (commenté par mes soins) :

```
                // toto = sizeof x
movl $4,toto    // sizeof x = 4
                // foobar(x)
movl x,%eax     // x → %eax
pushl %eax      // %eax → pile
call foobar     // appeler
addl $4,%esp    // dépiler
                // foobar(x, x)
movl x,%eax     // 2ème argument
pushl %eax
movl x,%eax     // 1er argument
pushl %eax
call foobar
addl $8,%esp
```

On voit que si la structure utilise quatre octets, le passage d'argument se fait comme pour un entier : les quatre octets sont empilés froidement dans la pile.

Avec une structure à un seul octet, on a :

```
                // toto = sizeof x
movl $1,toto    // sizeof x = 1
                // foobar(x)
addl $-2,%esp   // deux octets inutilisés dans la pile
movb x,%al      // x → %al
pushw %ax       // %ax → pile
call foobar
addl $4,%esp    // dépiler 4 octets
```

(Je rappelle que les registres `%al` et `%ax` sont des versions de `%eax` quand on n'accède qu'à un ou deux octets). La chose importante est de voir que la structure sur un seul octet a utilisé quatre octets dans la pile, comme l'indique l'instruction finale qui dépile les arguments. La même chose se produit pour les structures de deux et trois octets.

D'une manière générale, les structures passées en argument sont empilées dans un espace dont la taille est arrondie au multiple de quatre supérieur ou égal à la taille de la structure ; par exemple, pour une structure de treize octets, on a :

```
                    // toto = sizeof x
movl $13,toto       // sizeof x = 13
                    // foobar(x)
addl $-16,%esp      // réserver 16 octets sur la pile
movl $x,%eax        // %eax = &x
movl (%eax),%edx    // octets 0 à 3 de x sur la pile
movl %edx,(%esp)
movl 4(%eax),%edx   // idem pour les octets 4 à 7
movl %edx,4(%esp)
movl 8(%eax),%edx   // idem pour les octets 8 à 11
movl %edx,8(%esp)
movb 12(%eax),%al   // idem pour l'octet 12
movb %al,12(%esp)
call foobar
addl $16,%esp       // libérer les 16 octets
```

Quand la structure devient vraiment grande, le compilateur utilise les instructions spéciales pour recopier ses octets vers la pile ; ainsi, avec une structure de 367 octets.

```
                   // toto = sizeof x
movl $367,toto     // sizeof x = 367
                   // foobar(x)
addl $-368,%esp    // réserver 368 octets.
movl %esp,%edi     // %edi = pile = adresse destination
movl $x,%esi       // %esi = x = adresse source
cld                // il va falloir incrémenter %esi et %edi
movl $91,%ecx      // nombre de mots de 4 octets à copier ?
rep movsl          // copier les 91 × 4 octets
movsw              // copier 2 octets
movsb              // copier 1 octet
call foobar
addl $368,%esp     // libérer les 368 octets
```

En résumé

Lors d'un appel de fonction, le compilateur empile les arguments dans une zone mémoire de la pile dont la taille est arrondie au multiple de quatre supérieur ou égal, avec le premier argument sur le sommet de la pile.

3.5.3 La réception des arguments par la fonction appelée

Comme on a vu plus haut, la fonction appelée ne connaît de la pile que le pointeur de pile, dont elle suppose qu'il pointe sur une adresse de retour et au dessus de laquelle se trouvent les arguments avec le premier sur le dessus.

Les deux premières instructions de toute fonction fabriquée par le compilateur sont :

```
pushl %ebp      // sauve le registre %ebp
movl %esp,%ebp  // %ebp pointe sur l'ancien %ebp
```

On a appelé `%ebp` le *pointeur de frame* : ce registre contient l'adresse qui se trouve juste au dessus de l'adresse de retour et sa valeur ne bougera pas jusqu'au retour de la fonction. Puisque la pile croit vers les adresses basses, l'adresse de retour se trouve donc en `4(%ebp)` ; la zone où les arguments ont été empilés par la fonction qui appelle commencent en `8(%ebp)`.

Utilisation de stdarg

Une fonction qui peut recevoir un nombre variable d'arguments, comme la fonction `printf`, peut utiliser les macros *stdarg* pour les récupérer. On trouve dans la page de manuel stdarg(1) un exemple d'utilisation de ces macros :

```
void foo(char *fmt, ...)
{
    va_list ap;
    int d;
    char c, *p, *s;

    va_start(ap, fmt);
    while (*fmt)
        switch(*fmt++) {
        case 's':              /* string */
            s = va_arg(ap, char *);
            printf("string %s\n", s);
            break;
        case 'd':              /* int */
            d = va_arg(ap, int);
            printf("int %d\n", d);
            break;
        case 'c':              /* char */
            /* Note: char is promoted to int. */
            c = va_arg(ap, int);
            printf("char %c\n", c);
            break;
        }
    va_end(ap);
}
```

`va_list` déclare une variable du type approprié ; `va_start` initialise la variable pour qu'elle désigne les arguments qui se trouvent après l'argument `fmt` ; `va_arg` sert à la fois à récupérer l'argument indiqué par `ap`, puis à déplacer `ap` sur l'argument suivant ; `va_end` indique qu'on a terminé d'utiliser `ap`.

3.5.4 Fonctionnement de stdarg

Vu la manière dont sont passés les arguments, la variable initialisée par `va_start` doit faire pointer `ap` juste après l'argument `fmt` qu'on lui passe en argument. La macro `va_next` doit renvoyer la valeur de `ap`, tout en faisant avancer `ap` jusqu'au prochain multiple de quatre supérieur ou égal à la taille du type qu'on lui indique. Je ne vois pas ce que doit faire la macro `va_end`.

3.5.5 Contenu du fichier `stdarg.h`

Le fichier `stdarg.h` contient, une fois qu'on a retiré le copyright et les `#ifdef` :

```
(1)   typedef char *va_list;
(2)
(3)   #define __va_size(type) \
(4)       (sizeof(type) + sizeof(int) - 1) / sizeof(int)) \
(5)           * sizeof(int))
(6)
(7)   #define va_start(ap, last) \
(8)           ((ap) = (va_list)&(last) + __va_size(last))
(9)
(10)  #define va_arg(ap, type) \
(11)      (*(type *)((ap) += __va_size(type), \
(12)          (ap) - __va_size(type)))
(13)
(14)  #define va_end(ap)
```

La macro `va_end` (définie ligne 14) ne fait rien, comme prévu.

Le type `va_list` (défini ligne 1) est un pointeur sur des caractères ; ceci signifie que l'arithmétique sur une variable de type `va_list` fonctionnera correctement : si on ajoute 1 au contenu de la variable, elle pointera sur l'octet suivant. Ceci suggère qu'on pourrait sans dommage définir le type `va_list` comme un `int`.

La macro `va_size` (définie lignes 3–5) prend un type en argument. En supposant que `sizeof type` vaille n, et que `sizeof int` vaille 4, elle calcule $\frac{(n+4-1)}{4} \times 4$; comme il s'agit ici d'une division entière, ceci calcule, comme on s'y attendait, le plus petit multiple de quatre supérieur ou égal au nombre d'octets utilisés par le type ; il s'agit du nombre d'octets utilisés par le compilateur pour passer une variable d'un type donné en argument dans la pile. Noter que l'argument `type` n'est utilisé qu'avec `sizeof` : `va_size` fonctionne donc indifféremment avec un type ou une variable comme argument, ce que n'indiquait pas la documentation.

La macro `va_start` (lignes 7–8) place dans la variable indiquée par `ap` l'adresse de l'argument indiqué par `last` plus la taille occupée par `last` dans la pile, comme indiqué par `va_size`.

La macro `va_arg` (lignes 10–12) modifie la valeur de la variable indiquée par `ap` pour remonter dans la pile par dessus l'espace utilisé par une variable du type `type`; ensuite, pour renvoyer l'ancienne valeur de `ap`, elle soustrait du contenu de `ap` cette même valeur qu'elle vient d'ajouter.

3.5.6 A quoi sert stdarg ?

On a vu que les macros de `stdarg` sont une interface compliquée vers quelque chose de simple. Ainsi, une fonction qui fait la somme d'une liste d'entiers terminée par 0 qu'on lui passe en argument peut se coder, avec stdarg, comme :

```
1  /* corr1e.c
2   */
3  int
4  somme(int premier, ...){
5          va_list ap;
6          int courant;
7          int somme;
8
9          somme = 0;
10         va_start(ap, premier);
11         for(courant = premier; courant != 0;
12                 courant = va_arg(ap, int))
13             somme += courant;
14         va_end(ap);
15         return somme;
16 }
```

On peut aussi la définir, de façon plus simple, comme :

```
1  /* corr1f.c
2   */
3  int
4  somme(int premier, ...){
5    int * p = &premier;
6    int i, somme;
7
8    for(i = somme = 0; p[i] != 0; i++)
9      somme += p[i];
10   return somme;
11 }
```

Le problème est ici celui de la portabilité : rien ne nous garantit que le passage d'argument fonctionne de la même manière sur tous les processeurs. Cette fonction ne fonctionne plus quand la pile croit vers les adresses hautes, ni quand les entiers sont passés dans la pile à l'intérieur de huit octets, ni quand les données sur huit octets doivent se trouver sur une adresse multiple de huit,

ni quand tout ou partie des arguments est passé via des registres. Par exemple, cette fonction ne fonctionne pas sur les processeurs Intel 64 bits.
Ex. 3.7 — (pas de corrigé) Le code présenté ici a été écrit et testé sous le système d'exploitation FreeBSD. Refaire le même travail sous Linux.

Ex. 3.8 — (pas de corrigé) Le code présenté ici ne fonctionne que sur un processeur Intel 32 bits. Refaire le même travail sur un processeur 64 bits.

3.6 Application : le fonctionnement de `setjmp` et `longjmp` en C

Dans la librairie C, on trouve deux fonctions `setjmp` et `longjmp` qui permettent de faire des sauts *entre* les fonctions. Le principe est que la fonction `setjmp` sauve le contexte dans lequel on l'a appelée et plus tard, quand on appelle `longjmp` depuis une autre fonction, on revient juste après l'appel de `setjmp`.

3.6.1 Schéma d'utilisation

Ces fonctions sont principalement utilisées pour les traitements d'erreurs dans les processus organisés autour d'une boucle. Le schéma usuel d'utilisation est le suivant :

```
# include <setjmp.h>
jmp_buf redemarrage;

...

void
boucler(void){
  if (setjmp(redemarrage) == 0)
    printf("On demarre\n");
  else
    printf("On re-demarre apres une erreur\n");

  for(;;)
    traiter();
  printf("Bye\n");
}
```

La ligne 7 déclare une variable du type `jmp_buf` : il s'agit d'un peu d'espace dans lequel stocker le pointeur de pile et le frame pointer. Avant de démarrer la boucle de traitement, la ligne 29 appelle la fonction `setjmp` pour sauver l'environnement au démarrage de la boucle ; la fonction renvoie 0 lorsqu'on l'a appelée.

Ensuite, aux lignes 34–35, on rentre dans la boucle; dans notre exemple elle appelle répétivement la fonction `traiter`. Quand la fonction `traiter` *ou une fonction appelée par elle* détecte une erreur, elle peut appeler la fonction `longjmp` avec `redemarrage` comme argument. Le retour de la fonction `longjmp` ne se fait pas à la suite du programme, mais comme si on revenait de l'appel de la fonction `setjmp` à la ligne 29.

Pour avoir un exemple qui fonctionne, je définis une fonction `traiter` qui lit une ligne et appelle la fonction `erreur` quand la ligne est vide. La fonction `erreur` appelle `longjmp` pour redémarrer.

```
1  void
2  erreur(char * str){
3    fprintf(stderr, "erreur: %s\n", str);
4    longjmp(redemarrage, 1);
5  }
6
7  void
8  traiter(void){
9    char ligne[1024];
10
11   while(fgets(ligne, sizeof ligne, stdin) != NULL){
12     if (ligne[0] == 0 || ligne[1] == 0) // il peut y avoir \n dans ligne[0]
13       erreur("ligne vide");
14     printf("La ligne lue contient %s", ligne);
15   }
16 }
```

3.6.2 Fonctionnement intime de `setjmp`–`longjmp`

Quand elle est appelée, la fonction `setjmp` photographie l'état de la pile en recopiant dans le `jmp_buf` le frame pointer et l'adresse de retour avant de renvoyer 0. Quand on appelle la fonction `longjmp`, elle remplace son frame pointer et son adresse de retour par ce qu'elle trouve dans le `jmp_buf` (voir figure 3.7.)

En fait, il reste une difficulté, puisqu'il faut aussi s'assurer que la pile après le retour est revenue dans l'état où elle se trouvait au moment de l'appel à la fonction `traiter`. Cela implique une ligne de code en assembleur.

La maitrise du mécanisme permet de comprendre pourquoi, comme l'indique la page de manuel de `setjmp`, « le contexte de pile sera invalide si la fonction qui appelle `setjmp` retourne ». Quand la fonction qui a appelé `setjmp` effectue son `return`, la pile qu'elle employait est libérée et un `longjmp` sur l'environnement sauvé renverra sur une zone de pile inutilisée ou bien utilisée pour autre chose.

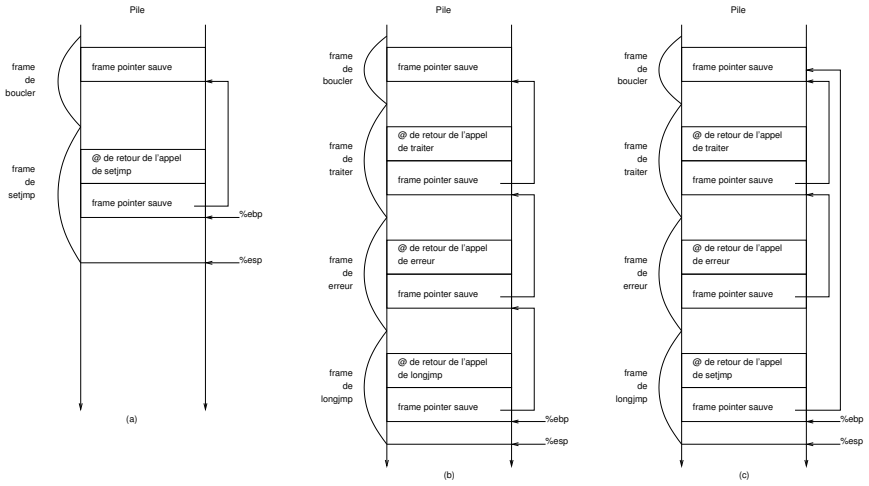


FIGURE 3.7 – La fonction `setjmp` sauve son adresse de retour et son frame pointer. Plus tard, quand la fonction`erreur` appelle `longjmp`, celui-ci remplace le frame-pointer sauvé et son adresse de retour par ceux sauvés par l'appel de `setjmp`.

3.6.3 Rapport avec les exceptions d'autres langages

Certains langages évolués disposent de constructions d'échappement spécifiques pour traiter les erreurs et les exceptions ; la plus courante est celle qu'on rencontre par exemple en Java et en C++, qui se présente sous la forme d'un `try ... catch` et d'un `throw`, organisés de la façon suivante

```
try {
    ici du code qui peut provoquer une exception avec throw
}
catch (exception) {
    traitement de l'exception
}
```

Ce code fonctionne avec des manipulations de piles similaires à celles qu'on rencontre dans `setjmp`–`longjmp`. Il est équivalent au C

```
jmp_buf etat;
if (setjmp(etat) == 0){
    le code qui peut provoquer une exception ; ici il
    faut utiliser longjmp(etat) au lieu de throw
} else {
    traitement de l'exception
}
```

3.7 L'ordre des arguments dans la bibliothèque d'entrées-sorties standard

Si vous aussi vous avez du mal à vous souvenir de l'ordre des arguments dans les appels de fonctions, voici un truc mnémotechnique pour ceux de la `stdio` : *Toutes les fonctions de la stdio ont le descripteur de fichier comme dernier argument.*

Comme dans les règles de grammaires, cette règle a une exception : elle s'applique à toutes les fonctions de la stdio *sauf celles qui ont un nombre variable d'arguments comme fprintf ou fscanf*; pour celles-ci le descripteur de fichier est le *premier* argument.

Quand on a compris le mécanisme de passage des arguments, la raison de cette exception est assez limpide. Puisque ces fonctions ont un nombre d'arguments variable et que ceux-ci sont de types variés, elle doivent parcourir le format pour savoir où les trouver dans la pile. Si le descripteur de fichier était passé en dernier, comme dans les autres fonctions de la stdio, `fprintf` aurait besoin de parcourir le format deux fois : une première fois pour trouver le descripteur de fichier où écrire puis une seconde fois afin d'écrire effectivement les caractères du format et les autres arguments formattés.

En passant le descripteur de fichier dans le premier argument, les fonctions comme `fprintf` peuvent le récupérer immédiatement (il est sur le sommet de la pile), récupérer le format (il est juste en dessous), puis parcourir le format en descendant dans la liste des arguments suivant leur type indiqué dans le format.

C'est bien sur la même chose pour des fonctions comme `snprintf` (qui formatte dans la mémoire au lieu d'écrire dans un fichier) ou comme `fscanf` qui lit au lieu d'écrire. Pour toutes les autres, le descripteur de fichier est le dernier argument. [1]

3.8 Manipuler l'adresse de retour

```
1  # include <stdio.h>
2
3  void
4  foo(int x){
5    printf("On rentre dans la fonction foo, x vaut %d\n", x);
6  }
7
8  void
9  bar(int x){
10   void ** p = &x - 1; // adresse de retour
```

1. En revanche, je n'ai pas de moyen de me souvenir de quels sont les deuxième et troisième arguments des fonctions `fread` et `fwrite` ; c'est une assez bonne raison pour les éviter et appeler directement `read` et `write`.

```
11   *p = (void *)foo;
12   printf("On rentre dans la fonction bar, x vaut %d\n", x);
13 }
14
15 int
16 main(){
17   printf("main appelle bar(12)\n");
18   bar(12);
19   printf("fin normale du main\n");
20   return 0;
21 }
```

3.9 Si vous avez un système 64 bits

Avec un système 64 bits, les exemples donnés dans ce chapitre ne fonctionnent pas. Beaucoup d'ordinateurs récents peuvent supporter un système 32 ou 64 bits, la différence étant principalement le nombre d'octets pour stocker une adresse. Quand on installe une distribution Linux, on est souvent conduit à choisir entre une version 32 et 64 bits.

Pour étudier la matière du chapitre, deux solutions : trouver une machine (éventuellement installer une machine virtuelle) 32 bits, ou bien traduire les exemples du chapitre (et du précédent).

Les différences les plus importantes à mon avis sont :

- Les adresses occupent bien sur huit octets. Les int et les long n'en occupent que quatre.
- La version 64 bits des registres porte un nom qui commence par `r` . Par exemple, le registre `%eax` s'appelle `%rax` quand on utilise ses 64 bits ; le pointeur de pile s'appelle `%rsp` et le pointeur de frame `%rbp`.
- Il y huit registres généraux supplémentaires, qui portent le nom `%r8` à `%r15` (en version 64 bits) et `%r8d` à `%r15d` en version 32 bits. (Ces registres sont disponibles aussi sur les processeurs récents qu'on utilise en mode 32 bits mais il faudrait prévenir le compilateur qu'on ne compte pas faire tourner le programme sur un vieux processeur 386 pour qu'il puisse les utiliser sans danger.)
- Le passage des six premiers arguments se fait par les registres : dans l'ordre `%edi`, `%esi`, `%edx`, `%ecx`, `%r8d` puis `%r9d`. Les arguments à partir du septième vont dans la pile, comme dans le compilateur pour le 386.
- L'appelant passe des informations supplémentaires à la fonction appelée via le registre `%eax` ; le plus simple est de le mettre à zéro avant tous les appels de fonction ; ça fonctionne pour (presque) tous les exemples du cours.
- La valeur renvoyée est toujours placée dans `%eax` (ou dans `%rax` si c'est une adresse).
- Les prologues et les épilogues de fonctions sont pleins de directives qui commencent par `.cfi`. Vous pouvez les ignorer pour les exemples du cha-

pitre.

– Quand la dernière chose que fait une fonction est d'en appeler une autre, l'optimiseur remplace l'appel par un saut après avoir mis la pile dans l'état adéquat : le retour de la fonction appelée reviendra en réalité dans la fonction qui a appelé la fonction courante. (Formulé autrement : gcc reconnaît et traite correctement la récursion terminale.)

Chapitre 4

L'analyse lexicale

L'analyseur lexical est chargé de découper en mots le flux de caractères que contient le fichier source. Ce petit chapitre introduit seulement le sujet. Pour trouver une présentation de Lex, fondée sur les expressions régulières, voir le dernier chapitre du support.

4.1 Analyse lexicale, analyse syntaxique

Ce paragraphe évoque le problème que pose la délimitation entre analyse lexicale et analyse syntaxique, puis montre l'interaction normale des deux analyseurs dans un compilateur ordinaire.

4.1.1 Analyse lexicale versus analyse syntaxique

La limite entre analyse lexicale et analyse syntaxique est relativement floue. Comme on le verra, les analyseurs syntaxiques sont plus puissants que les analyseurs lexicaux et on peut leur confier, en plus de leur travail principal, le travail de l'analyseur lexical. C'est souvent le cas dans les définitions formelles de langages qui utilisent une présentation BNF (comme *Backus Naur Form* ; il s'agit d'une manière de décrire les langages dont nous reparlerons dans le prochain chapitre). Cependant, distinguer le travail (assez simple) de l'analyseur lexical de celui (plus complexe) de l'analyseur syntaxique permet de circonscrire la complexité inhérente aux analyseurs syntaxiques et nous conserverons la distinction entre les deux.

En pratique, on considère que ce que l'analyseur lexical ne sait pas faire ressort de l'analyse syntaxique ; on verra vers la fin du chapitre (sous la forme de calembours C) des exemples avec un compilateur réel de formes de programmes valides qui ne sont pas compilés correctement à cause de défaillances de l'analyseur lexical.

4.1.2 Analyse lexicale et analyse syntaxique

Dans l'architecture usuelle des programmes, c'est l'analyseur syntaxique qui dirige le travail. Il s'efforce de construire l'arbre syntaxique en utilisant les mots que lui a déjà renvoyé l'analyseur lexical. Quand il est prêt à recevoir le mot suivant, alors il appelle la fonction qui fait le travail de l'analyseur lexical. Cette fonction lit le mot suivant et le renvoie à l'analyseur syntaxique.

4.2 En vrac

Ici, j'évoque quelques questions relatives aux analyseurs lexicaux et j'apporte quelques réponses. Les indications de cette section sont mise en œuvre dans l'exemple qui conclue le chapitre.

4.2.1 Renvoyer un type *et* une valeur

Le rôle de l'analyseur lexical est d'identifier le mot suivant. Il doit donc retourner le type du prochain mot.

Quand ce mot est par exemple un mot clef comme `while` ou `else`, le type est suffisant pour décrire complètement le mot. En revanche, quand il s'agit de quelque chose comme un identificateur ou une constante, le type indiquera de quelle sorte d'objet il s'agit, mais l'analyseur lexical devra aussi renvoyer la *valeur* de l'objet : pour un identificateur, au moins la chaîne de caractères qui contient son nom ; pour une constante, sa valeur.

L'usage veut qu'on nomme la fonction principale de l'analyseur lexical `yylex`, que cette fonction renvoie le type du mot sous la forme d'un entier et qu'elle place la valeur du mot dans une variable globale nommée `yylval` (*lval* comme *lexical value*). Quand les mots reconnus par l'analyseur lexical peuvent avoir des valeurs de types différents (par exemple un nombre entier ou un nombre flottant ou une chaîne de caractères), alors `yylval` sera défini comme une union en C. L'analyseur syntaxique utilisera le type renvoyé pour savoir quel champs de l'union il doit utiliser.

Cet usage est en fait destiné à permettre l'intégration de l'analyseur lexical avec les analyseurs syntaxiques fabriqués par Yacc et Bison, présentés dans les chapitres suivants.

4.2.2 Le caractère qui suit le mot, `ungetc`

Il est nécessaire à l'analyseur lexical de lire tous les caractères qui composent un mot. Dans la plupart des cas, il lui faut aussi lire le premier caractère qui suit le mot afin d'être en mesure d'en détecter la fin.

Par exemple, quand il voit un chiffre au début d'un mot, un analyseur lexical pour le langage C doit traiter la lecture d'un nombre, qui peut se composer de

plusieurs chiffres ; il va donc rentrer dans une boucle qui lira les caractères de ce nombre jusqu'au dernier. Pour déterminer quel est le dernier caractère du nombre, l'analyseur lexical devra lire un caractère de plus : celui qui ne sera pas un chiffre indiquera que le précédent était le dernier.

Après avoir lu le caractère qui suit le mot, l'analyseur lexical *doit* le remettre en place pour que l'appel suivant trouve ce caractère. (Il ne s'agit pas toujours d'un espace ; en C par exemple, les caractères `0+1` ne contiennent aucun espace mais composent trois mots distincts.)

La mauvaise manière On peut installer une (fine) couche logicielle entre l'analyseur lexical et les fonctions qui lisent les caractères. Par exemple on pourra écrire deux fonctions `lirecar` qui lit un caractère et `annulerlire` qui annule la lecture du dernier caractère :

```
static int dernierlu;
static int annule;

/* annuler  --  annuler la dernière lecture de caractère */
void
annulerlire(void){
  annule = 1;
}

/* lirecar  -- lire le prochain caractère */
int
lirecar(void){
  if (annulé == 1){
    annule = 0;
    return dernierlu;
  }
  return dernierlu = getchar();
}
```

La fonction `lirecar` lit un caractère et le renvoie, sauf si la variable `annule` lui indique qu'elle doit renvoyer de nouveau le dernier caractère lu, qui a été sauvé dans la variable `dernierlu`. (La variable `dernierlu` doit être du type `int` et non du type `char` parce que `getchar` peut aussi renvoyer `EOF` pour indiquer la fin du fichier ; or `EOF` *n'est pas* le code d'un caractère.)

La bonne manière La librairie d'entrées sorties standard contient déjà un mécanisme pour effectuer ce travail, sous la forme de la fonction `ungetc` ; c'est cette fonction qu'il convient d'utiliser (si on utilise la librairie d'entrées sorties standard) ; la seule différence avec le code précédent est qu'on ne peut pas annuler la lecture du code `EOF` (puisque ce n'est pas un code de caractère). Dans la fonction `yylex`, le code de lecture d'un entier décimal pourra ressembler à :

```
int
yylex(void){
  int c;

redo:
  c = getchar();             // lire un caractère :

  if (c == EOF)              // - fin de fichier
    return EOF;

  if (isspace(c))            // - espaces : à sauter
    goto redo;

  if (c >= '1' && c <= '9'){ // - constante décimale
    int res;

    for(res = c - '0'; isdigit(c = getchar());
        res = res * 10 + c - '0')
      ;
    if (c == EOF){
      erreur("EOF juste après l'entier %d (ignoré)\n", res);
      return EOF;
    }
    // défaire la lecture du caractère qui suit
    ungetc(c, stdin);
    yylval = res;            // yylval = valeur du nombre
    return INT;              // signaler qu'on a lu un entier
  }

  if (c == '0'){             // - constante octale ou hexadécimale
    ...
  }
  ...
}
```

4.2.3 Les commentaires

L'analyseur lexical est en général chargé aussi de retirer les commentaires du programme source. Une question, à laquelle répond en général la description du langage, est de savoir si les commentaires sur une ligne incluent ou pas la fin de ligne. A ma connaissance, le langage TEX est le seul pour lequel c'est le cas. Cela permet que

```
foo% Ce commentaire inclue la fin de la ligne
bar
```

soit lu comme le mot `foobar`.

Notez qu'en C, les commentaires ne sont pas imbriqués (un `/*` à l'intérieur d'un commentaire est traité comme des caractères ordinaires). Cela complique un peu les choses quand on souhaite commenter un bloc de code (mais on peut utiliser `# if 0` ou `# ifdef NOTDEF`), et permet une devinette : *qu'imprime le programme suivant ?*

```
1  # include <stdio.h>
2
3  int
4  main(){
5          int a, *p;
6
7          a = 123;
8          p = &a;
9          printf("%d\n", a/*p);
10         printf("La ligne precedente ne contient pas %d\n",
11                 1 /* multiplication inutile ? */ * 321);
12         return 0;
13 }
```

(Cette devinette devrait probablement aller dans la section *Calembours en C*.)

4.2.4 Quelques difficultés de l'analyse lexicale

L'analyse lexicale est un travail plutôt facile mais elle contient quelques pièges, dont certains sont évoqués dans les paragraphes suivants.

Les constantes en virgule flottante en C

Il y a de nombreuses manière d'écrire les nombres en virgule flottante en C, comme dans la plupart des langages de programmation.

- Une série de chiffres qui contient un point, comme dans `000.000`.
- Il peut ne pas y avoir de chiffres à gauche ou à droite du point décimal (comme dans `000.` ou `.000`. (Peut-il n'y avoir de chiffres ni à gauche ni à droite, d'après votre compilateur C favori ?)
- Les chiffres significatifs peuvent être suivis d'une indication d'exposant (*« notation scientifique »*) ; l'indication d'exposant commence par un `e` (majuscule ou minuscule) et est suivie par un nombre entier, positif ou négatif, comme dans `0.0e0`, `0.0e-0` ou `0.0e+0`.
- Quand il y a une indication d'exposant, alors le point décimal dans la mantisse n'est pas nécessaire, comme dans `0e0`.
- Avant ceci, il peut y avoir ou pas une indication du signe du nombre avec `+` ou `-`.

– Après ceci, il peut y avoir un `l` ou un `L` pour indiquer que le nombre flottant est du type `double`, ou bien `f` ou `F` pour le forcer au type `float`.

Cette liste de variantes de la notation des nombres en virgule flottante est incomplète (voir exercice).

Ex. 4.1 — Trouver, dans une norme du langage C, la description précise de toutes les formes que peut prendre une constante en virgule flottante. (L'exercice demande bien de trouver cette description dans une *norme*, pas dans un ouvrage sur le langage C.)

Identificateurs, mots clefs

Les identificateurs (en première approximation les noms de variables et de fonctions) ont la même structure que la plupart des mots réservés comme `while`, `for` ou `return`. Plutôt que de traiter spécialement ces mots clefs, il est souvent plus simple *et plus efficace* de les lire comme des identificateurs ; une fois que le mot est lu, on vérifie s'il est présent dans la table des mots clefs et dans ce cas on le traite spécialement.

Le fragment de code suivant donne la structure générale de ce traitement :

```
typedef struct Keyword Keyword;

struct Keyword {
  char * string;  // le mot clef
  int retvalue;   // la valeur à renvoyer
};

/*
 La table des mots clefs
 (Dans la vraie vie, il vaudrait mieux construire une hash table)
 */
Keywork kwtable[] = {
  { "while", WHILE },
  { "for", FOR },
  { "return", RETURN },
  ...
  { 0 }
};

int
yylex(void){
  int c;

redo:
  c = getchar();               // lire un caractère :
  ...
```

```
  if (isalpha(c) || c == '_'){ // identificateur, type ou mot clef
    Keyword * p;
    char name[MaxNameLen];
    int i;

    for(name[0] = c, i = 1; isalnum(c = getchar()) || c == '_'; i++)
      if (i == MaxNameLen - 1)
        erreur("identificateur trop long (%d caractères max)",
               MaxNameLen);
      else
        name[i] = c;
    name[i] = 0;
    if (c == EOF)
      erreur("EOF juste après %s\n", name);

    p = lookupkw(string, kwtable);
    if (p != 0)
      return p->retvalue;
    else {
      yylval.ident = lookupident(string);
      return IDENT;
    }
  }
  ...
}
```

Le fragment de code ci-dessus montre l'appel de la fonction `lookupkw` qui recherche l'identificateur lu dans la table des mots clefs, mais pas sa définition.

La fonction `lookupident` regarde si le mot lu est présent dans la table des identificateurs et sinon l'y rajoute ; le fragment de code ne contient pas non plus sa définition.

Identificateurs et types définis

On souhaite confier à l'analyseur lexical la distinction entre les noms de types (comme `int`, `long` ou `double`) et les identificateurs qui désignent des noms de variables ou de fonction.

Dans le langage C, la chose est à peu près impossible à cause du mécanisme du `typedef` qui permet de renommer un nom de type avec une chaîne choisie par l'utilisateur. De plus, ce nom de type *n'est pas* un mot réservé et peut être utilisé pour autre chose dans le même programme. Par exemple, la déclaration suivante de `jill` est valide

```
typedef int foo, * bar, (*joe)(foo, bar);
```

```
joe jill;
```

Pour cette raison, c'est à l'analyseur syntaxique que la plupart des compilateurs C confient la distinction entre identificateurs et types définis avec `typedef`. On est ici à la limite entre lexical et syntaxique évoquée au début du chapitre.

Calembours en C et leur traitement par gcc

Dans les programmes réels, les programmeurs prennent soin de placer des espaces entre les mots de leurs programmes de manière à pouvoir les relire. Cependant un analyseur lexical ne peut pas compter sur ces espaces pour lui faciliter le travail, parce que chacun place les espaces d'une façon différente. (D'ailleurs, chaque programmeur C, y compris moi, est fermement convaincu que *sa* manière de disposer les espaces est la seule correcte).

Ex. 4.2 — Les expressions C qui suivent contiennent les constantes 1 et -1, deux variables entières `a` et `b`, l'opérateur arithmétique binaire `-` (comme dans `a - b`), l'opérateur unaire `-` (comme dans `- a`) et l'opérateur `--`, préfixé ou postfixé.

Si `a` vaut 1 et que `b` vaut 2, quelle est la valeur de l'expression et la valeur de `a` et `b` après son exécution ?

Si on supprime les espaces de ces expressions, lesquelles sont encore correctes d'après le compilateur ?

Pour les expressions correctes sans les espaces, quelles sera leur valeur et celle de `a` et `b` après leur exécution ?

Expliquer

	valeur	`a` après	`b` après	ok sans espace	valeur	`a` après	`b` après
`a - 1`				o			
`a - -1`				o			
`a - - 1`				o			
`a - - - 1`				o			
`a - - - - - b`				o			
`a -- - -- b`				o			
(supplément)							
`a -- + -- b`				o			
`a ++ + ++ b`				o			

4.3 Un exemple élémentaire mais réaliste d'analyseur lexical

Dans cette section, je présente un analyseur lexical simple. Pour qu'il soit utilisé, je présente aussi un analyseur syntaxique élémentaire. Le tout donne un petit programme qui peut servir à additionner deux nombres entiers.

4.3.1 Le langage

Le programme lit des lignes, qui devront contenir chacune une expression arithmétique élémentaire à effectuer. Un exemple simple de session pourra être :

```
$ a.out              # lancement du programme
? 234 + 432          # une expression arithmétique simple
= 666                #   affichage du résultat
? 234+432            # la même sans espace
= 666                #   c'est pareil
? 1+1+1              # expression trop complexe
erreur de syntaxe
? 1                  # expression trop simple
erreur de syntaxe
? ^D                 # fin de fichier
Ciao
$
```

4.3.2 L'analyseur syntaxique

Le rôle de l'analyseur syntaxique est joué par la fonction `yyparse`. Elle appelle la fonction `yylex` pour lire chaque mot : d'abord le premier nombre, puis l'opérateur, puis le second nombre, puis la fin de ligne. Elle effectue l'opération et affiche le résultat.

En cas d'erreur, la fonction `erreur` met en œuvre la reprise avec `setjmp`/`longjmp` présentée à la fin du chapitre précédent : elle affiche le message d'erreur puis revient à la fonction `yyparse` dans le `setjmp` présent au début.

4.3.3 L'analyseur lexical

L'analyseur lexical est réalisé par la fonction `yylex` : la version présentée ici lit l'expression ligne par ligne.

On trouvera dans db-elem.c une version qui lit l'expression caractère par caractère, mais elle présente des déficiences dans la récupération d'erreur. Quand la ligne est trop courte, elle considère que la ligne suivante fait partie de l'erreur. Quand une ligne ne se termine pas par un newline, le message d'erreur est incohérent. (Une ligne peut ne pas se terminer par newline si elle est trop longue pour rentrer dans le buffer qu'on a passé à `fgets` ou bien si on l'a envoyée avec Controle-D depuis le clavier.)

Lecture des caractères

La fonction lit une ligne complète dans le tableau de caractères `ligne` et conserve ensuite un index `iligne` qui indique quelle est la position courante

dans la ligne. Ce qui se trouve entre `ligne[0]` et `ligne[iligne-1]` a déja été traité par l'analyseur lexical ; le reste est à analyser.

Au début de l'appel, on vérifie qu'il reste des caractères à traiter. Si ce n'est pas le cas, la fonction lit une nouvelle ligne (avec la fonction `fgets`). Dans le cas où il ne reste rien à lire, elle renvoie 0 pour l'indiquer à l'analyseur syntaxique.

Ensuite, la fonction saute tous les espaces avec la boucle de la ligne 50. Pour faciliter la lecture de la fonction, le caractère courant est placé dans la variable c (ligne 52).

Le caractère # sert à introduire des commentaires. Dans ce cas, la boucle qui suit (ligne 55 et seq.) conduit à ignorer tous les caractères jusqu'à la fin de la ligne.

Le test de la ligne 60 traite tous les caractères qui forment un mot à eux tous seuls : dans notre cas, seulement le +, la fin de ligne et la fin de chaîne.

Le test de la ligne 63 traite le cas où c'est un chiffre qui a été lu. Cela annonce un nombre. L'analyseur lexical doit renvoyer la contante Nombre (définie comme la valeur 256) et placer la valeur du nombre dans la variable globale `yylval`. Le problème est que la fonction doit lire tous les chiffres du nombre pour déterminer sa valeur, ce qui peut se faire de diverses façons. À mon avis elles présentent toutes des inconvénients. Le code présente différentes méthodes, entre lesquelles on peut choisir avec des directives de compilation conditionnelles. J'ai placé les méthodes dans l'ordre dans lequel elles me semblent préférables.

Conversion explicite par le programme À la ligne 65 (et suivantes), le programme effectue lui-même la conversion entre les codes des caractères qui composent le chiffre et la valeur du chiffre en représentation interne de l'ordinateur. Pour obtenir la représentation interne de la valeur du chiffre, il retire le code ASCII de 0 de celui du chiffre.

Utilisation de la fonction `strtol` Le code des lignes 70 et suivantes utilse la fonction `strtol` pour effectuer la conversion que les lignes précédentes effectuaient explicitement. On utilise le second argument pour récupérer (dans `p`) l'adresse du caractère qui suit le nombre et on l'utilise pour mettre à jour `iligne`, pour qu'il contienne l'index du premier caractère qui suit le nombre.

Utilisation de sscanf C'est avec `sscanf` que le code de la ligne 75 et suivantes effectue la conversion. On vérifie avec l'assertion de la ligne 77 que la conversion s'est bien effectuée *(il faut toujours vérifier la valeur renvoyée par scanf et ses variantes)*. Comme la fonction n'indique pas le nombre de caractères occupés par le nombre, la boucle de la ligne 78 est nécessaire pour faire avancer l'index du caractère courant après le nombre.

Utilisation de atoi La fonction `atoi` fait le même travail que `strtol`, mais seulement en base 10 et sans donner d'indication du nombre caractères utilisés

par le nombre.

Ex. 4.3 — Modifier le programme pour pouvoir aussi effectuer les autres opérations arithmétiques usuelles (soustraction, multiplication, division et modulo).

Ex. 4.4 — Ajouter la possibilité de définir des variables simples, avec des noms d'un seul caractère alphabétique, et d'utiliser la valeur de ces variables dans les expressions, comme dans l'exemple suivant :

```
$ a.out
? 12 * 12   # operation simple
= 144
? a = 234   # affectation de variable
= 234
? a + 0     # consultation de variable
= 234
? 0 + a     # dans l'autre sens
= 234
? a + a     # des deux cotés
= 468
? a + 0     # la valeur n'a pas changé
= 234
? b = a     # copie de valeur
= 234
? b + 0     # consultation de b
= 234
? c + 0     # variable indéfinie
= 0         # valeur par défaut
? ^D
Ciao
```

Indication : si on n'accepte que les variables dont le nom ne fait qu'un caractère, on peut utiliser un tableau à 26 entrées. L'analyseur lexical doit maintenant renvoyer une nouvelle constante (`Variable` par exemple) quand il rencontre une variable et placer dans yylval quelque chose qui indique de quelle variable il s'agit, par exemple une valeur entre 0 et 25 qui indique la place du nom de la variable dans l'alphabet. On peut conserver la valeur des variables dans un simple tableau de valeur à 26 entrées.

Ex. 4.5 — Modifier le programme pour que les expressions soient représentées par des nombres en virgule flottante, au lieu d'entiers. La variable globale `yylval` doit maintenant être une `union` pour distinguer le cas où on a lu une variable (la valeur du mot est un entier) et celle où il s'agit d'une nombre (la valeur du mot est un nomre en virgule flottante).

(Indication : pour le lecture des nombres en virgule flottante, le plus facile à mon avis est d'utiliser `strtod`).

On pourra tester l'analyseur lexical par exemple avec :

```
$ a.out
? a = 0     # juste des chiffres
= 0
? a = 0.    # des chiffres et la virgule
= 0
? a = .0    # la virgule et des chiffres
= 0
? a = .     # scanf n'en veut pas
nombre flottant mal formé
? a = 0e0   # exposant simple
= 0
? a = 0e-0  # exposant avec un signe -
= 0
? a = 0e+0  # le même avec un signe +
= 0
? a = 0.0e0 # la virgule et l'exposant
= 0
```

)

Ex. 4.6 — Modifier le programme précédent pour accepter les noms de variables composés de plusieurs caractères.

(Indication : il faut maintenant que l'analyseur lexical lise tous les caractères qui compopsent le nom de la variable ; l'union yylval doit maintenant pouvoir recevoir une chaîne de caractères comme argument. La structure de donnée qui associe une variable avec une valeur ne peut plus être un simple tableau de valeur ; le corrigé utilise un tableau de structures qui associent le nom d'une variable et sa valeur ; un programme sérieux utiliserait sans doute une table de hash.)

Ex. 4.7 — (Trop difficile, ne pas faire) Modifier le programme pour pouvoir entrer des nombres négatifs.

Ex. 4.8 — (Hors sujet, pas de correction) Modifier le proramme pour pouvoir placer dans une variable le résultat d'une opération, comme dans `a = b + c`. (Il s'agit d'un travail à effectuer avec l'analyseur syntaxique, dont parle le prochain chapitre.)

Chapitre 5

L'analyse syntaxique : présentation

L'analyse syntaxique prend les mots que produit l'analyse lexicale et les regroupe jusqu'à former des phrases du langage en appliquant des règles de *grammaire*. Dans le cas des compilateurs, ce regroupement se fait dans la plupart des cas sous la forme d'un *arbre syntaxique*.

L'analyse syntaxique est le gros morceau de ce cours, et elle est présentée dans trois chapitres : celui-ci montre deux analyseurs syntaxiques écrits en C. Le chapitre suivant montre des analyseurs syntaxiques qui utilisent les outils Yacc et Bison. Le troisième détaille le fonctionnement interne des parseurs construits par Yacc et Bison.

5.1 Grammaires, langages, arbres syntaxiques, ambiguïtés

Les grammaires que nous étudions dans le cours de compilation ne ressemblent pas formidablement à celle de notre langue maternelle que nous avons étudiée pendant nos premières années d'école. Elles sont constituées d'un ensemble de règles qui décrivent complètement la structure de toutes les phrases d'un langage.

5.1.1 Les règles de grammaires

Les règles de nos grammaires sont formées avec une partie gauche et une partie droite, pour décrire une des constructions du langage. A gauche, un nœud interne d'un arbre ; à droite, la liste des enfants de cet arbre. Par exemple :

```
expression : NOMBRE
```

indique qu'un nombre à lui tout seul (comme reconnu par l'analyseur lexical) forme une expression arithmétique correcte. Les règles peuvent êtres *récursives*, comme dans :

```
expression : expression '+' expression
```

avec laquelle on sait qu'une expression arithmétique suivie d'un + suivie d'une autre expression arithmétique forme encore une expression arithmétique correcte. (La règle est récursive parce que le symbole de gauche apparaît aussi dans sa partie droite.)

5.1.2 Les symboles terminaux et non terminaux

On appelle symboles *terminaux* ceux qui apparaissent à droite dans certaines règles de grammaire mais jamais à gauche ; ce sont les types de mot que renvoie la fonction d'analyse lexicale. Inversement, quand un symbole apparaît à gauche de certaines règles, cela signifie qu'il est composé de mots et on l'appelle un symbole *non terminal*. On abrège souvent ces noms en *terminaux* et *non terminaux*, sans préciser qu'il s'agit d'un symbole. L'usage est de donner aux terminaux des noms en majuscules et des noms en minuscules aux non terminaux.

5.1.3 Les arbres syntaxiques

On représente souvent le résultat du travail de l'analyseur syntaxique sous la forme d'un arbre syntaxique : par exemple, considérons la grammaire :

```
x : A y B ;
y : C x ;
y : D ;
```

Elle se compose de trois règles, pour un langage dont les seuls mots sont A, B, C et D. A partir de ces quatre mots, elle permet de construire une infinité de phrases ADB, $ACADBB$, $ACACADBBB$, etc. On peut décrire les phrases du langage avec *« Un D, précédé d'un seul A puis CA un nombre quelconque de fois et suivi d'autant de B qu'il y a de A »*.

La première règle décrit comment est construit un x, la deuxième et la troisième montrent deux manières différentes de construire un y.

L'analyseur syntaxique lit une phrase et applique les règles sur les mots jusqu'à trouver une séquence d'application des règles qui rende compte de la structure de la phrase, comme dans la figure 5.1 (ou bien jusqu'à avoir détecté une erreur de syntaxe).

La construction de l'arbre syntaxique par l'analyseur s'appelle une *dérivation*. Quand on construit un nœud de l'arbre syntaxique de type x à partir des mots y et z, on dit qu'on *réduit* y et z en x. Le symbole qui doit se trouver à la racine de l'arbre s'appelle le *symbole de départ*.

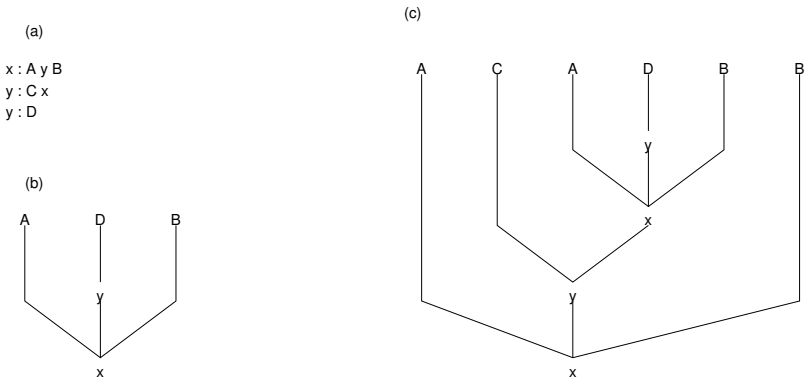


FIGURE 5.1 – A partir de la grammaire simple (a), deux arbres syntaxiques qui rendent compte de la structure des deux phrases les plus simples du langage `A D B` en (b) et `A C A D B B` en (c).

5.1.4 Les grammaires ambiguës, l'associativité et la précédence

Quand il est possible de dériver plusieurs arbres syntaxiques différents à partir d'une même phrase, on dit que la grammaire est *ambiguë*. Les ambiguïtés dans les grammaires proviennent principalement de deux sources : soit on peut dériver un mot ou un groupe de mots en non terminaux différents (en utilisant donc des règles différentes), soit on peut construire une arbre avec une forme différente en utilisant les mêmes règles.

Pour un exemple de la première sorte, considérons la grammaire élémentaire pour un langage qui ne comprend que la phrase A :

```
x : y
x : z
y : A
z : A
```

Sur `A`, la phrase unique de la grammaire, on peut dériver deux arbres syntaxiques différents en appliquant des règles différentes : celui où la racine x est constituée d'un y qui lui même est fait d'un A et celui où la racine x est constituée d'un z lui aussi fait d'un A. Il n'y a pas de manière de déterminer en examinant la grammaire laquelle des deux interprétations doit être préférée.

L'associativité

Pour un exemple de la seconde sorte, regardons une grammaire pour un langage dont les phrases sont constituées d'un nombre quelconque de A :

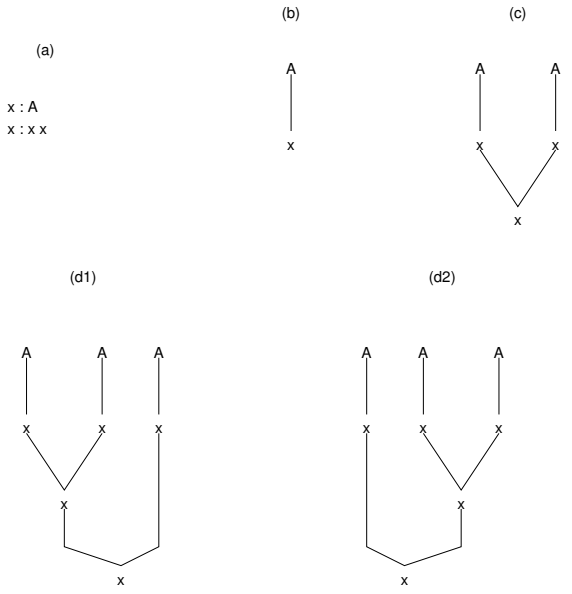


FIGURE 5.2 – A partir de la grammaire simple (a), on ne peut dériver qu'un seul arbre syntaxique pour la phrase `A` (en b) ou `A A` (en c) ; en revanche, il y a deux arbres syntaxiques différents possibles pour la phrase `A A A` (en d1 et d2) : la grammaire est ambiguë.

```
x : A
x : x x
```

Pour la phrase `A`, il n'y a un qu'un seul arbre : celui où le A est réduit en x par la première règle. Pour la phrase `A A`, chaque A est réduit en x (par la première règle) puis les deux x sont réduits en A par la seconde règle. En revanche pour la phrase `A A A`, il existe deux dérivations possibles (figure 5.2). Chacune de ces dérivations emploie les mêmes règles mais les applique dans un ordre différent.

On peut rencontrer ce genre de construction dans une grammaire des expressions arithmétiques ; si on se limite aux nombres et à + :

```
expr : NBRE
expr : expr '+' expr
```

De la même manière que la précédente, cette grammaire permet de dériver deux arbres différents pour une phrase qui contient trois nombres, comme dans la figure 5.3. Le problème ici est celui de l'*associativité* de l'opérateur + : l'arbre de gauche de la figure correspond à un + associatif à gauche, celui de droite à un + associatif à droite.

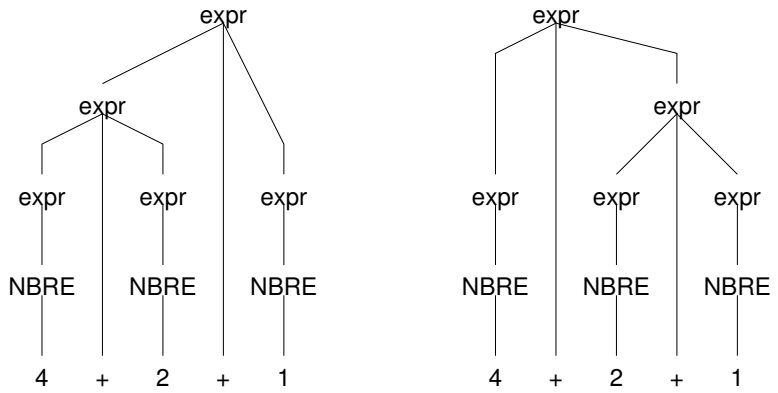


FIGURE 5.3 – Deux arbres syntaxiques différents pour la même expression arithmétiques $4 + 2 + 1$, dérivables à partir de notre grammaire. Chacun des arbres rend compte d'une interprétation différente de la phrase, mais avec les deux arbres l'expression résultat vaut 7 grâce aux propriétés du $+$.

Si on remplace le $+$ par un $-$ dans notre grammaire, alors les deux arbres correspondront à deux interprétations de l'expression arithmétique qui donnent des résultats différents, comme dans la figure 5.4.

La précédence

Avec l'interprétation usuelle des expressions arithmétiques, nous savons qu'une expression comme $1 + 2 \times 3$ s'interprète de manière à ce que sa valeur soit 7. Une grammaire possible serait

```
expr : NBRE
expr : expr + expr
expr : expr * expr
```

mais à partir de cette grammaire, il est possible de dériver plusieurs arbres syntaxiques différents pour une expression arithmétique qui contient à la fois des $+$ et des $\times$ (figure 5.5). Comme dans la section précédente, il s'agit des mêmes règles de grammaires et la question porte sur la manière de les employer.

Réécritures de grammaires

Il est toujours possible de réécrire une grammaire pour en lever les ambiguïtés dues aux questions de précédence et d'associativité.

Quand une règle présente une ambiguïté due à une question d'associativité comme dans

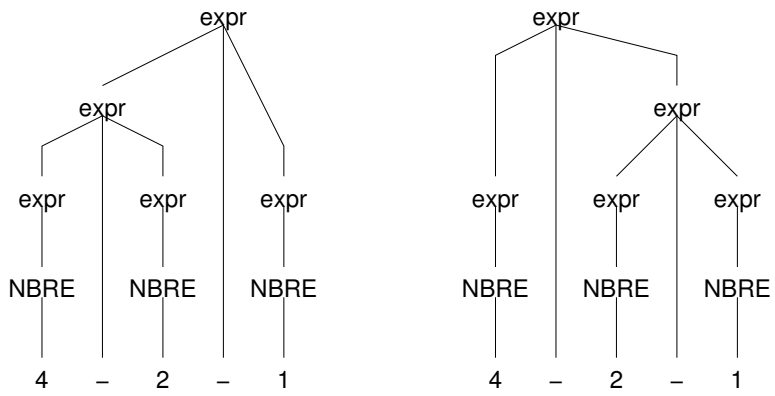


FIGURE 5.4 – Les mêmes arbres syntaxiques que dans la figure 5.3 : du fait des propriétés du $-$, l'expression arithmétique dérivée comme dans l'arbre de gauche vaut 1 alors que dérivée comme dans l'arbre de droite elle vaut 3.

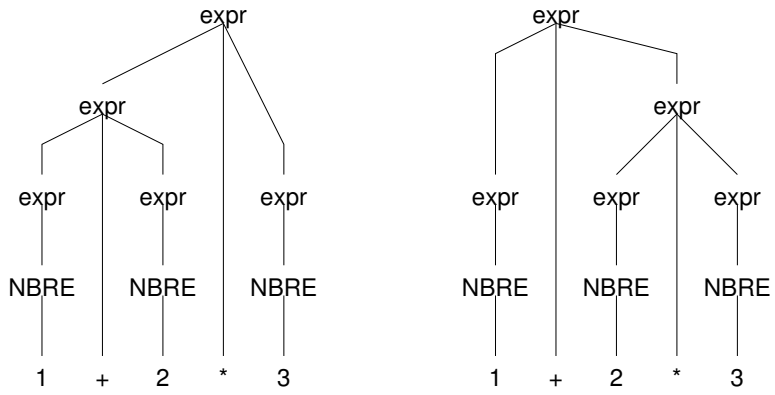


FIGURE 5.5 – À partir de la phrase $1 + 2 \times 3$, on peut dériver deux arbres syntaxiques différents. Celui de gauche donne une une valeur de 9, celui de droite une valeur (habituelle) de 7. La question est celle de la *précédence* des opérateurs $+$ et $\times$.

```
x : A
x : x x
```

on peut la réécrire comme

```
x : A
x : x A
```

ou comme

```
x : A
x : A x
```

pour forcer l'associativité à gauche ou à droite.

Quand une règle présente une ambiguïté due à la précédence deux opérateurs comme dans

```
x : A
x : x B x
x : x C x
```

on peut lever l'ambiguïté en introduisant un type de nœud supplémentaire, comme dans

```
x : x1
x : x B x1
x1 : A
x1 : x1 C x1
```

Toutes les occurrences de l'opérateur C seront réduites dans des applications de la dernière règle de grammaire avant de réduire les occurrences de l'opérateur B par la deuxième règle.

C'est pour fixer la précédence des opérateurs d'addition et de multiplication qu'on voit souvent la grammaire des expressions arithmétiques écrite comme :

```
expr : terme
terme : facteur
terme : terme + facteur
facteur : NBRE
facteur : facteur * NBRE
facteur : ( expr )
```

Ex. 5.1 — Que vaut l'expression $4 - 2 - 1$ en C ?

Ex. 5.2 — Soit la grammaire :

```
p : A B
p : A B p
```

1. Dériver l'arbre syntaxique de `ABABAB`.
2. Décrire d'une phrase (en français !) le langage décrit par cette grammaire.
3. La grammaire est-elle ambiguë ?

Ex. 5.3 — Mêmes questions que pour l'exercice précédent, avec la grammaire

```
p : /* rien */
p :  A B p
```

Ex. 5.4 — Soit la grammaire :

```
p : A B
p : A p B
```

1. Générer à partir de cette grammaire une phrase de 2 mots, de 4 mots, de 6 mots, de 8 mots.
2. Décrire d'une phrase le langage décrit par cette grammaire.

Ex. 5.5 — Soit la grammaire :

```
p : /* rien */
p : A p B p
```

Générer à partir de cette grammaire toutes les phrases de 2 mots, de 4 mots, de 6 mots. On peut remplacer le mot `A` par une parenthèse ouvrante et le mot `B` par une parenthèse fermante. Qu'obtient-on alors ?

Ex. 5.6 — Dériver un arbre syntaxique pour chacune des phrases construites à l'exercice précédent avec la grammaire :

```
p : /* rien */
p : p A p B
```

La grammaire est-elle ambigüe (peut-on dériver plusieurs arbres syntaxiques différents à partir de la même phrase) ?

Ex. 5.7 — Mêmes questions pour la grammaire :

```
p : /* rien */
p : A p B
p : p p
```

5.1.5 BNF, EBNF

Je n'ai pas envie de rédiger cette section. L'article EBNF sur wikipedia.org est très bien.

5.2 Les analyseurs à précédence d'opérateurs

Dans cette section, je montre la structure d'un petit analyseur qui utilise la précédence des opérateurs arithmétiques pour reconnaître la structure d'une expression.

Nous commencerons par examiner une version élémentaire de l'analyseur qui ne traite pas les parenthèses, puis une version un peu plus avancée dans laquelle elles sont prises en compte.

On trouve dans le *dragon book* un squelette d'analyseur à précédence d'opérateur plus évolué qui n'utilise qu'une seule pile.

5.2.1 Un analyseur à précédence d'opérateurs élémentaire

Cet analyseur ne traite que des expressions arithmétiques composées de nombres entiers et des quatre opérations (addition, soustraction, division et multiplication).

L'analyseur lexical

```
1  # include <string.h>
2
3  enum {
4    Num = 1,                        // renvoye par l'analyseur lexical
5    Executer = -1, Empiler = 1,     // comparaison de precedence
6    MaxStack = 1024,                // taille maximum des piles
7  };

8
9  char * operator = "+-*/";         // la liste des operateurs
10 int yylval;                       // la valeur du lexeme
11
12 /* yylex  --  lit le prochain mot sur stdin,
13   place sa valeur dans yylval,
14   renvoie son type,
15   proteste pour les caracteres invalides
16  */
17 int
18 yylex(void){
19   int c;
20   int t;
21
22  redo:
23   while(isspace(c = getchar()))
24     if (c == '\n')
```

```
25        return 0;
26
27    if (c == EOF)
28      return 0;
29
30    if (isdigit(c)){
31      ungetc(c, stdin);
32      t = scanf("%d", &yylval);
33      assert(t == 1);
34      return Num;
35    }
36
37    if (strchr(operator, c) == 0){
38      fprintf(stderr, "Caractere %c (\\0%o) inattendu\n", c, c);
39      goto redo;
40    }
41    return c;
42  }
```

L'analyseur lexical est une simple fonction qui lit des caractères sur l'entrée standard et renvoie le *type* du prochain mot.

Il traite la fin de ligne comme le marqueur de la fin de phrase et renvoie 0. Il ignore tous les autres espaces (lignes 32–37).

Un chiffre annonce un nombre; il remet le chiffre dans les caractères à lire, lit la *valeur* du nombre avec `scanf` dans la variable globale `yylval` et renvoie le *type* avec la constante `Num`.

Pour tous les autres caractères, il vérifie avec `strchr` sa présence dans la liste des opérateurs et renvoie le code du caractère en guise de *type* (ici, les opérateurs n'ont pas besoin d'avoir de valeur).

Le cœur de l'analyseur

```
53  int operande[MaxStack];
54  int operateur[MaxStack];
55  int noperande, noperateur;

...

110   int mot;
111
112   noperateur = noperande = 0;
113   do {
114     mot = yylex();
115
116     if (mot == Num){
117       assert(noperande < MaxStack);
118       operande[noperande++] = yylval;
```

```
119
120     } else {
121       while(noperateur > 0 && preccmp(operateur[noperateur - 1], mot) < 0)
122         executer(operateur[--noperateur]);
123       assert(noperateur < MaxStack);
124       operateur[noperateur++] = mot;
125     }
126   } while(mot != 0);
127
128   if (noperateur != 1 || noperande != 1 || operateur[0] != 0)
129     fprintf(stderr, "Erreur de syntaxe\n");
130   else
131     printf("%d\n", operande[0]);
132 }
```

L'analyseur utilise deux piles : l'une pour les opérateurs et l'autre pour les opérandes, définies aux lignes 53–55. (`noperande` et `noperateur` sont les pointeurs de ces piles ; ce sont les index des premiers emplacements libres.)

Le principe général de l'analyseur est d'empiler les opérandes sur leur pile (lignes 117–119) ; pour les opérateurs, il compare leur précédence (on dit aussi leur *priorité*) avec celle de celui qui se trouve sur le sommet de la pile, avec la fonction `preccmp` présentée plus loin ; tant que cette du sommet de pile est supérieure, on le dépile et on l'exécute ; finalement, on empile l'opérateur (lignes 122–125).

La boucle s'arrête quand on a traité l'indicateur de fin de phrase. Si l'expression arithmétique était bien formée, alors il ne reste qu'un opérande (qui représente la valeur de l'expression) sur la pile des opérandes et le marqueur de fin d'expression sur la pile des opérateurs.

La comparaison des précédences

```
57 /* preccmp  --  prec. de l'operateur gauche - prec. de l'operateur droit */
58 int
59 preccmp(int gop, int dop){
60   assert(gop != 0);
61   if (dop == 0)                  // EOF : executer ce qui reste.
62     return Executer;
63
64   if (gop == dop)                // le meme : executer
65     return Executer;
66
67   if (gop == '+' || gop == '-'){ // + ou -
68     if (dop == '+' || dop == '-')
69       return Executer; // puis + ou - : executer
70     else
71       return Empiler; //  puis * ou / : empiler
```

```
72   }
73
74   return -1; // dans tous les autres cas, executer
75 }
```

La comparaison des précédences se fait avec une fonction `preccmp`, qui renvoie une valeur `Executer` ou `Empiler`. On l'utilise pour comparer la précédence de l'opérateur au sommet de la pile avec celle du nouvel opérateur. Quand la précédence du nouvel opérateur est inférieure à celle qui se trouve au sommet de la pile, il faudra exécuter celui qui est au sommet de la pile avant d'empiler le nouveau. À l'inverse, quand elle est plus faible, il faudra empiler le nouvel opérateur par dessus l'autre.

Pour cet exemple élémentaire, on aurait pu attribuer un nombre à chaque opérateur en guise de niveau de précédence et comparer les nombres, mais cela aurait compliqué les choses pour l'analyseur plus évolué que nous verrons ensuite.

L'exécution des opérateurs

L'exécution des opérateurs est confiée à la fonction `executer`. En fait cette exécution des opérations ne ressort pas réellement de l'analyse syntaxique, mais de la *sémantique* des opérateurs.

Dans notre exemple simple, la fonction se contente de dépiler les opérandes, d'effectuer l'opération et d'empiler le résultat *mais il est important de comprendre que ce faisant la fonction construit implicitement un arbre syntaxique.*

Elle utilise la variable intermédiaire `t` car une expression comme

```
operande[noperande++] = operande[--noperande] + operande[--noperande]
```

n'a pas de résultat bien défini en C : le compilateur peut faire effectuer les `++` et les `--` dans un ordre imprévisible. L'utilisation de la variable intermédiaire `t` nous permet de le forcer à les placer dans l'ordre qui convient.

```
77 void
78 executer(int op){
79   int t;
80
81   switch(op){
82   default:
83     fprintf(stderr, "Operateur impossible, code %c (\\0%o)\n", op, op);
84     return;
85   case '+':
86     t = operande[--noperande];
87     t += operande[--noperande];
88     operande[noperande++] = t;
89     return;
90   case '-':
```

```
91      t = operande[--noperande];
92      t = operande[--noperande] - t;
93      operande[noperande++] = t;
94      return;
95    case '*':
96      t = operande[--noperande];
97      t *=operande[--noperande];
98      operande[noperande++] = t;
99      return;
100   case '/':
101     t = operande[--noperande];
102     t = operande[--noperande] / t;
103     operande[noperande++] = t;
104     return;
105   }
106 }
```

Ex. 5.8 — Qu'est qui change si on remplace la ligne 65 par `return Empiler;` ?

Ex. 5.9 — (trop facile pour mériter un corrigé) Modifier le programme pour accepter l'opérateur modulo (`'%'`), avec la précédence qu'il possède dans le langage C.

Ex. 5.10 — (un peu plus délicat) Modifier le programme pour qu'il accepte l'opérateur d'exponentiation `^` ; attention, cet opérateur a une précédence plus forte que les autres *et est associatif à droite*.

Ex. 5.11 — Remplacer la fonction `executer` par quelque chose qui permet d'imprimer l'expression lue sous forme polonaise postfixée. Par exemple, quand elle lit `2 * 3 + 4 * 5`, elle écrit `2 3 * 4 5 * +`.

5.2.2 Un analyseur à précédence d'opérateurs moins élémentaire

J'étends ici l'analyseur pour lui faire traiter les expressions parenthésées.

L'analyseur lexical

La modification de l'analyseur lexical est élémentaire : il suffit d'ajouter les parenthèses dans la liste des opérateurs en remplaçant la ligne 18 par

```
18      char * operator = "+-*/()";   // la liste des opérateurs
```

Le cœur de l'analyseur

Il n'y a pas de modification à faire au coeur de l'analyseur. C'est là le principal intérêt de l'analyse à précédence d'opérateurs.

La comparaison des précédences

Pour traiter les parenthèses avec la précédence, il faut :
- Quand le nouvel opérateur est une parenthèse ouvrante, l'empiler systématiquement.
- Pour tous les opérateurs usuels, on les empile par dessus une parenthèse ouvrante.
- Quand le nouvel opérateur est une parenthèse fermante, exécuter systématiquement tous les opérateurs usuels et empiler la fermante par dessus sa parenthèse ouvrante.
- Pour tous les opérateurs, quand le sommet de la pile opérateur est une parenthèse fermante, il faut l'exécuter ; l'exécution consistera à dépiler la parenthèse ouvrante associée.

On peut obtenir ce résultat en modifiant la fonction `preccmp` en ajoutant les lignes suivantes :

```
(65.1)  if (gop == ')')       // toujours executer la parenthese fermante
(65.2)    return Executer;
(65.3)  if (dop == ')'){      // avec une nouvelle fermante :
(65.4)    if (gop == '(')
(65.5)      return Empiler; //   l'empiler sur son ouvrante
(65.6)    else
(65.7)      return Executer; //   et executer sur tous les autres.
(65.8)  }
(65.9)  if (dop == '(') // toujours empiler les nouvelles ouvrantes
(65.10)   return Empiler;
```

On peut noter que la relation de précédence n'est pas une relation d'ordre total : par exemple celle de '+' est plus forte que celle de ')' quand le '+' apparait avant et plus faible quand il apparait après la fermante.

L'exécution

```
(121.1)  case '(':
(121.2)    fprintf(stderr, "Ouvrante sans fermante\n");
(121.3)    return;
(121.4)  case ')':
(121.5)    if (noperateur == 0){
(121.6)      fprintf(stderr, "Fermante sans ouvrante\n);
(121.7)      return;
(121.8)    }
```

```
(121.9)    t = operateur[--noperateur];
(121.10)   if (t != '('){
(121.11)     fprintf(stderr, "Cas impossible avec la parenthese fermante\n");
(121.12)     return;
(121.13)   }
(121.14)   return;
```

Les lignes 121.1–121.3 ne seront exécutées que quand il restera une ouvrante dans la pile à la fin de l'expression (puisque tous les autres opérateurs sont empilés par dessus, sauf le marqueur de fin d'expression et la fermante, et que l'exécution de la fermante supprime l'ouvrante de la pile). La présence d'une ouvrante dans la pile indique donc que l'expression ne contient pas suffisamment de fermantes.

Pour `executer` une fermante, on la la retire simplement de la pile avec l'ouvrante correspondante (à la ligne 121.9).

5.2.3 Des problèmes avec les analyseurs à précédence d'opérateurs

Avec les analyseurs à précédence d'opérateurs, c'est difficile de différencier le même opérateur sous sa forme unaire et sa forme binaire (comme `*` ou `-` en C) au niveau de l'analyse syntaxique, ou les opérateurs unaires préfixés ou postfixés comme `++` et `--`. Le moins compliqué est de confier la distinction entre les deux versions de l'opérateur à l'analyseur lexical (un opérateur qui apparaît au début d'une expression ou après un autre opérateur est un opérateur unaire préfixé), mais cela reste malaisé.

La détection d'erreur est complexe avec ces analyseurs ; par exemple mon analyseur traite sans erreur une expression comme `1 2 3 + * + 4` et annonce qu'elle vaut 11.

Chapitre 6

L'analyse syntaxique : utilisation de Yacc

Ce chapitre traite d'un outil important d'Unix : Yacc. Le mot *Yacc* est un acronyme pour *Yet Another Compiler Compiler* (en français : *encore un compilateur de compilateur*). Il s'agit en fait d'un programme qui *fabrique* un analyseur syntaxique à partir d'une description de la grammaire qu'on lui fournit.

Après un bref historique et une description de comment l'utiliser, je montre Yacc avec des exemples. Le chapitre suivant décrit en détail la manière dont il analyse la grammaire pour produire l'analyseur.

6.1 Yacc et Bison, historique

Yacc est un outil qui date du milieu des années 1970 ; il a été réalisé par S. C. Johnson pour le compilateur C qu'il écrivait à l'époque, qui s'appelait *pcc*, comme *Portable C Compiler*.

Yacc est un outil important : il ne permet pas réellement de compiler un compilateur (c'est à dire de produire un compilateur à partir d'une description des langages source et cible) ; en revanche, il simplifie *considérablement* la production d'un analyseur syntaxique correct. À mon avis, c'est grâce à lui (et à l'existence de `pcc` qu'il a permis) que le langage C est resté singulièrement libre de variantes d'implémentations, sans comité de normalisation, pendant plus de quinze ans (de sa conception jusqu'à la fin des années 1980).

Le projet Gnu (qui comprend les commandes gcc, gdb et emacs) a réécrit un programme compatible avec Yacc qu'il a nommé *Bison* ; je parle de Yacc en particulier mais presque tous les points s'appliquent également à Bison ; j'ai tenté d'indiquer toutes les différences significatives entre les deux programmes ; quand je ne mentionne pas de différence, c'est que Yacc et Bison fonctionnent de la même manière.

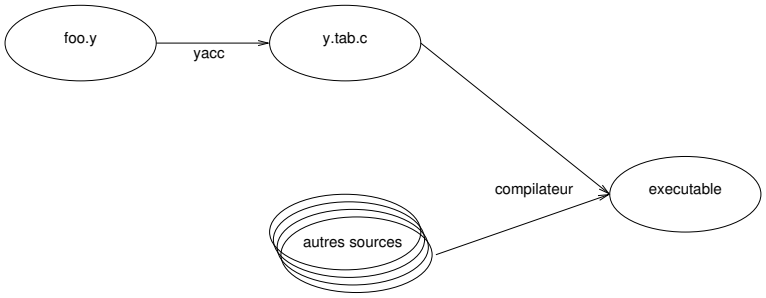


FIGURE 6.1 – Yacc prend une grammaire et ses actions dans le fichier `foo.y` ; il produit un fichier C `y.tab.c` (ou `foo.tab.c` pour Bison) qui contient une fonction C `yyparse` qui implémente un analyseur syntaxique pour cette grammaire.

6.2 Le fonctionnement de Yacc

Le principe est le suivant : dans un fichier (dont le nom se termine en général par `.y`), on décrit une grammaire et des actions à effectuer lorsque l'analyseur syntaxique a déterminé qu'il fallait appliquer une des règles de la grammaire.

A partir de ce fichier, Yacc produit un fichier C qui contient essentiellement une fonction `yyparse`, écrite en C par Yacc, qui implémente un analyseur syntaxique pour cette grammaire. On intègre ce fichier aux sources du programme (voir figure 6.1).

Comme dans les exemples vus plus haut, la fonction `yyparse` appelle une fonction `yylex`, qu'on doit écrire, pour servir d'analyseur lexical. Il faut également lui fournir une fonction `yyerror` qui sera appelée en cas d'erreur (dans la plupart des cas, ce sera à cause d'une erreur de syntaxe et `yyerror` sera appelée avec la chaîne `"Syntax error"` en argument ; en se donnant beaucoup de mal, on peut aussi réussir à provoquer un débordement de pile).

6.3 Un exemple élémentaire

```
1  /* elem.y
2   Un exemple elementaire d'analyseur syntaxique pour un langage stupide.
3  */
4  %term motA
5  %term motB
6
7  %{
8  # define YYDEBUG 1
9  int yydebug;
10
```

```
11 int yylex(void);
12 int yyerror(char *);
13 %}
14
15 %%
16 phrase : troisa
17                 { printf("J'ai trouve la phrase A A A\n"); }
18        | troisb
19                 { printf("J'ai trouve la phrase B B B\n"); }
20        ;
21
22 troisa : motA motA motA
23        ;
24
25 troisb : motB motB motB
26        ;
27 %%
28 # include <stdio.h>
29 # include <ctype.h>
30 # include <stdlib.h>
31
32 int
33 yyerror(char * str){
34   fprintf(stderr, "%s\n", str);
35   return 0;
36 }
37
38 int
39 yylex(void){
40   int c;
41
42   while(isspace(c = getchar()))
43     if (c == '\n')
44       return 0;
45   if (c == EOF)
46     return 0;
47   if (c == 'A')
48     return motA;
49   if (c == 'B')
50     return motB;
51
52   fprintf(stderr, "caractere imprevu %c (\\0%o)\n", c, c);
53   exit(1);
54 }
55
56 int
```

```
57 main(){
58   printf("? ");
59   yyparse();
60   return 0;
61 }
```

Cet exemple élémentaire présente un analyseur syntaxique pour un langage stupide qui se compose en tout et pour tour de deux phrases : `A A A` ou `B B B`. Il est tout entier regroupé dans le fichier `ec-elem.y`.

6.3.1 Structure d'un fichier pour Yacc

Le fichier est découpé en trois parties par des lignes qui ne contiennent que `%%` (lignes 15 et 27). Au début (lignes 1–14 dans l'exemple) on place des déclarations pour Yacc. La grammaire proprement dite avec les actions à effectuer quand on applique une règle se trouve au milieu (lignes 16–26). Après le second `%%` on peut placer ce qu'on veut comme code C : Yacc ne le traitera pas et se contentera de le recopier dans le fichier résultat.

6.3.2 La partie déclaration

La partie déclaration de notre exemple simple contient deux choses : d'une part la déclaration des symboles terminaux que peut renvoyer la fonction `yylex` (ici `motA` et `motB` aux lignes 4–5) avec le mot clef `%term`. D'autre part un peu de code C que je souhaite voir apparaître *avant* l'analyseur syntaxique dans le fichier résultat, encadré par deux lignes qui contiennent respectivement `%{` et `%}` (lignes 7 et 13).

Dans ce code C, il y a le prototype des deux fonctions utilisées par l'analyseur syntaxique mais que je dois définir : la fonction `yylex` qui joue le rôle de l'analyseur lexical et la fonction `yyerror` qui sera appelée par l'analyseur syntaxique en cas d'erreur.

J'ai également, à la ligne 8, défini la constante `YYDEBUG` pour que l'analyseur contienne du code de mise au point. Ce code aidera à comprendre ce qui se passe en affichant des messages lors de la dérivation de l'arbre quand la variable `yydebug` contiendra une valeur différente de 0. (Dit autrement : si `YYDEBUG` n'est pas défini, il n'y a pas de code de mise au point ; si `YYDEBUG` est défini mais que `yydebug` contient 0, il n'y a rien d'imprimé ; si `YYDEBUG` est défini et que `yydebug` vaut 1, on aura des messages pour toutes les actions sur la grammaire) Sauf si on a des contraintes de taille mémoire très strictes, il faut toujours définir la constante `YYDEBUG` pour aider à mettre la grammaire au point quand le besoin s'en fait sentir.

6.3.3 La partie code C

On place ce qu'on veut dans la troisième partie (ici de la ligne 28 jusqu'à la fin). J'y ai mis la fonction `yyerror`, la fonction `yylex` que l'analyseur appelle

pour lire l'un après l'autre les mots de la phrase à traiter, ainsi qu'une fonction `main` qui imprime un prompteur puis appelle l'analyseur via le nom de fonction `yyparse`.

6.3.4 La partie grammaire et actions

Entre les lignes 16 et 26 se trouvent les règles de grammaire ainsi que du code à exécuter quand la règle est appliquée. Le seul intérêt de cette grammaire est d'être simple,.

6.3.5 Utilisation

Compiler faire tourner.

```
$ bison ec-elem.y
$ gcc -g -Wall ec-elem.tab.c
$ a.out
? A A A
J'ai trouvé la phrase A A A
$ a.out
? B B B
J'ai trouvé la phrase B B B
$ a.out
? A B A
syntax error
```

Pour faire tourner avec `yydebug` à 1 (et voir les messages de mise au point de la grammaire), lancer gdb pour examiner le programme :

```
$ gdb -q a.out
(gdb)
```

Mettre un point d'arrêt au début du programme.

```
(gdb) b main
Breakpoint 1 at 0x401416: file ec-elem.y, line 58.
```

Lancer le programme qui s'arrête au début de main.

```
(gdb) run
Starting program: /home/jm/cours/compil/tex/src/a.out

Breakpoint 1, main () at ec-elem.y:58
58        printf("? ");
```

Mettre 1 dans yydebug.

```
(gdb) p yydebug=1
$1 = 1
```

Continuer.

```
(gdb) c
Continuing.
Starting parse
Entering state 0
Reading a token: ?
```

Le processus s'arrête pour lire la phrase. On tape `A A A`.

```
Reading a token: ? A A A
Next token is token motA ()
Shifting token motA ()
Entering state 1
Reading a token: Next token is token motA ()
Shifting token motA ()
Entering state 6
Reading a token: Next token is token motA ()
Shifting token motA ()
Entering state 9
Reducing stack by rule 3 (line 22):
   $1 = token motA ()
   $2 = token motA ()
   $3 = token motA ()
-> $$ = nterm troisa ()
Stack now 0
Entering state 4
Reducing stack by rule 1 (line 16):
   $1 = nterm troisa ()
J'ai trouvé la phrase A A A
-> $$ = nterm phrase ()
Stack now 0
Entering state 3
Reading a token: Now at end of input.
Stack now 0 3
Cleanup: popping nterm phrase ()

Program exited normally.
```

Même opération en tapant `B B A` :

```
(gdb) run
Starting program: /home/jm/cours/compil/tex/src/a.out

Breakpoint 1, main () at ec-elem.y:58
58        printf("? ");
(gdb) p yydebug = 1
$2 = 1
(gdb) c
Continuing.
Starting parse
Entering state 0
Reading a token: ? B B A
Next token is token motB ()
Shifting token motB ()
Entering state 2
Reading a token: Next token is token motB ()
Shifting token motB ()
Entering state 7
Reading a token: Next token is token motA ()
syntax error
Error: popping token motB ()
Stack now 0 2
Error: popping token motB ()
Stack now 0
Cleanup: discarding lookahead token motA ()
Stack now 0

Program exited normally.
```

6.4 Un exemple de calculateur simple

Le fichier `ed-1-calc.y` contient un exemple simple d'utilisation de Yacc pour construire un analyseur syntaxique pour les expressions arithmétiques.

On y retrouve la même structure quand dans l'exemple précédent : une partie déclaration, une partie grammaire, une partie libre.

Les choses nouvelles dans les déclarations sont

- Quand un terminal représente un seul caractère constant, ce n'est pas nécessaire de définir un symbole. On peut utiliser directement le code du caractère. Je l'utilise ici pour les opérateurs arithmétiques et le retour à la ligne. Juste pour montrer que ce n'est pas obligatoire, je ne l'ai pas utilisé pour l'opérateur d'exponentiation.
- La grammaire est ambiguë, mais j'ai rajouté des indications pour lever les ambiguïtés aux lignes 16–19 : chacune des lignes définit une associativité (gauche ou droite) et un niveau de précédence, du plus faible au plus

fort et liste les opérateurs auxquels elle s'applique : d'abord l'addition et la soustraction (associatifs à gauche) ligne 16, puis la multiplication et la division (eux aussi associatifs à gauche) ligne 17, puis l'opérateur d'exponentiation (associatif à droite) ligne 18.
- Pour l'opérateur − unaire, j'ai défini un niveau de précédence supérieur (à la ligne 19) et j'ai annoté la règle de grammaire qui l'utilise (ligne 46) avec le mot clef `%prec` pour indiquer le niveau de précédence à utiliser pour cette règle. C'est nécessaire puisqu'on a un seul symbole terminal (le `-`) qui est utilisé pour deux opérateurs différents : le moins binaire qui a la précédence indiquée pour `-` et le moins unaire pour lequel la précédence sera ce qui est indiqué avec `FORT`.
- J'ai rajouté, ligne 21, une indication du symbole qu'on veut trouver à la racine de l'arbre syntaxique. (Par défaut, Yacc tente d'y placer le symbole qui apparait à gauche de la première règle).

La grammaire définit le langage comme une suite d'expressions arithmétiques séparées par des sauts de ligne (lignes 24–28).

Avec les règles de grammaires, il y a des actions qui seront exécutées quand les règles seront appliquées. Les actions utilisent la possibilité d'attacher une valeur (entière par défaut) à chacun des nœuds de l'arbre syntaxique avec la notation `$` : `$1` est la valeur attachée au premier nœud de la partie droite de la règle, `$2` la valeur attachée au second nœud, etc. Le mot `$$` désigne la (nouvelle) valeur attachée au (nouveau) nœud construit en application de la règle.

Ex. 6.1 — Étant donné le fichier Yacc suivant qui analyse un langage dont les phrases ne sont constituées que de `a` :

```
%term A
%%
phrase : l
l : A
  | l A
  ;
%%
# include <stdio.h>
int main(){yyparse(); return 0;}
int yylex(void){ int c = getchar(); return c == 'a' ? A : 0; }
int yyerror(char*str){fprintf(stderr, "%s\n", str); }
```

(a) Ajouter les actions nécessaires à cette grammaire pour que le programme imprime le nombre de A présents dans la phrase. Le faire *uniquement* avec les actions et les valeurs attachées aux nœuds, sans définir aucune variable supplémentaire.

(b) Procéder de même pour imprimer le numéro de chaque expression à `1-calc.y`, pour avoir par exemple :

```
$ a.out
? 1 + 1
1: 2
```

```
? 1 + 1
2: 2
? 1 + 1
3: 2
? Bye
$
```

6.5 Un calculateur avec des nombres flottants

Le second calculateur, dans le fichier `2-calc.y` travaille sur les nombres flottants et permet de continuer à travailler après une erreur de syntaxe.

6.5.1 La récupération d'erreurs

En ajoutant la règle de la ligne 32

```
exprs : exprs error '\n'
```

on a indiqué à Yacc un point de récupération des erreurs avec le mot clef `error`. Quand on fera une erreur de syntaxe dans une expression arithmétique, Yacc va appeler `yylex` jusqu'à trouver le retour à la ligne, et à ce moment re-synchroniser l'analyseur syntaxique en réduisant le dernier nœud `exprs` et tous les mots qui le suivent en un nouveau nœud `exprs`. Cela permet au compilateur de continuer à traiter le programme pour détecter les erreurs suivantes.

6.5.2 Typage des valeurs attachées aux nœuds

Avec la déclaration des lignes 13–15 :

```
%union {
  float f;
}
```

on a prévenu Yacc que certains nœuds auraient une valeur du type `float`. Il faut maintenant spécifier le type de valeur attaché à chaque nœud avec une déclaration. Pour le symbole terminal `NBRE`, cela est effectué au moment de sa déclaration ligne 17 avec

```
%term <f> NBRE
```

Pour les nœuds de type `expr`, il faut ajouter nouvelle déclaration (ligne 18) :

```
%type <f> expr
```

6.6 Un calculateur avec un arbre véritable

Pour que les choses soient bien claires, j'ai ajouté un troisième calculateur qui construit explicitement l'arbre syntaxique en mémoire au lieu d'effectuer les calculs à mesure qu'il le construit (implicitement) en appliquant les règles de grammaire. On le trouvera dans le fichier `3-calc.y`.

Les déclarations de types sont maintenant faites avec

```
8       typedef struct Noeud Noeud;
...
18      %union {
19        int i;
20        Noeud * n;
21      };
22
23      %token <i> NBRE  /* Les symbole renvoyes par yylex */
24      %type <n> expr   /* Type de valeur attache au noeuds expr */
```

La valeur des feuilles est du type entier, les nœuds internes sont des pointeurs sur des structures `Noeud`.

Pour changer un peu par rapport aux programmes précédents, l'addition et la soustraction sont maintenant traitées par la même règle ; c'est la valeur attachée au mot `ADD` qui permet de faire la différence entre l'addition et la soustraction.

```
43      expr : expr ADD expr
44                      { $$ = noeud($2, $1, $3); }
```

Il a fallu modifier en conséquence l'analyseur lexical

```
106       case '+': case '-':
107         yylval.i = c;
108         return ADD;
```

et prévenir Yacc du type de valeur attaché à ces nœuds avec une déclaration que j'ai faite en même temps que la fixation du niveau de précédence :

```
26      %left <i> ADD
```

La multiplication et la division sont regroupées de la même manière.

A l'aide de l'arbre ainsi construit, le programme imprime l'expression en parenthèsant complètement chaque sous-expression composée avant de calculer et d'afficher sa valeur. Les deux se font avec une descente récursive simple dans l'arbre construit, par la fonction `parenthèse` (lignes 200 et seq.) et la fonction `eval` (lignes 176 et seq.).

Ex. 6.2 — (moyen) Modifier la fonction `parenthese` pour qu'elle n'imprime que les parenthèses nécessaires. (Indication : il faut tenir compte du contexte dans lequel l'expression apparait, par exemple sous la forme d'un pointeur vers le nœud parent.)

Chapitre 7

L'analyse syntaxique : le fonctionnement interne de Yacc

Dans ce chapitre, je détaille le fonctionnement interne de Yacc. Ceci est important à deux titres : d'une part cela permet de résoudre les problèmes qui ne manquent pas de se produire quand on écrit une grammaire pour Yacc et notamment de comprendre et résoudre les conflits *shift-reduce* qu'on rencontre dans les grammaires ; d'autre part, cela permet de comprendre les particularités des automates à piles (*push-down automata* en anglais) qui font partie des concepts structurants important de notre discipline.

Je commence par examiner un peu en détail *l'ordre* dans lequel les règles d'une grammaire sont activés ; ensuite je montre le fonctionnement de l'automate avec sa pile ; finalement j'explique de quelle manière l'automate est construit à partir des règles de grammaires.

7.1 L'analyseur de Yacc est ascendant de gauche à droite

L'analyseur syntaxique produit par Yacc est un analyseur ascendant (*bottom-up* en anglais) : il lit les mots et les regroupe dès que c'est possible, en construisant l'arbre depuis les feuilles vers la racine.

Comme il lit les mots de la phrase dans l'ordre, sans jamais revenir en arrière, on dit que c'est un analyseur de gauche à droite (*LR* comme *Left Right* en anglais).

Ceci est important pour comprendre l'ordre d'exécution des actions associées à une règle. Si on prend le fichier Yacc suivant :

```
%term N
%%
l : l N
        { printf("on ajoute %d à la liste\n", $2); }
   | /* rien */
       { printf("liste vide\n"); }
%%
# include <stdio.h>
int yyerror(char * s){ fprintf(stderr, "%s\n", s); }
int
yylex(void){
  static int it, t[] = { 1, 2, 3, 4, 0 };

  if (t[it] == 0)
    return 0;
  yylval = t[it++];
  return N;
}
int main(){ yyparse(); return 0; }
```

et qu'on le fait traiter par Yacc puis le compilateur C et qu'on le lance, on obtient

```
$ yacc t.y
$ gcc y.tab.c
$ a.out
liste vide
on ajoute 1 à la liste
on ajoute 2 à la liste
on ajoute 3 à la liste
on ajoute 4 à la liste
```

parce que l'arbre syntaxique construit est celui de la figure 7.1.

Si on modifie juste l'ordre dans la partie droite de la première règle, on a la grammaire (et les actions) :

```
l : N l
        { printf("on ajoute %d à la liste\n", $1); }
   | /* rien */
       { printf("liste vide\n"); }
```

L'exécution du programme résultat donne l'ajout des éléments en sens inverse :

```
liste vide
on ajoute 4 à la liste
on ajoute 3 à la liste
```

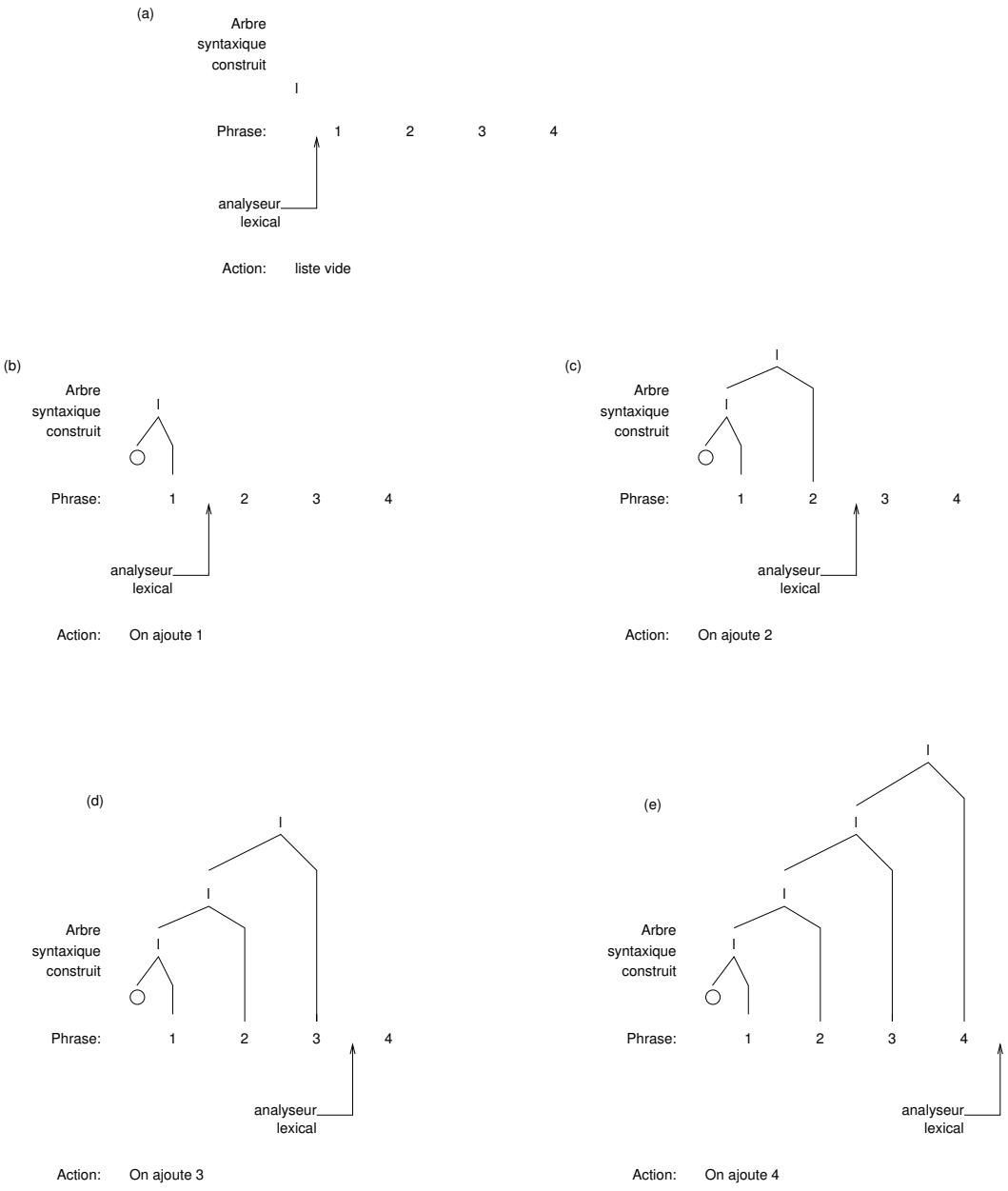


FIGURE 7.1 – La construction de l'arbre syntaxique avec l'action associée : en (a) on applique la deuxième règle `l : /* rien */` avant même la lecture du premier mot. Après la lecture de chaque mot, l'analyseur ajoute un nœud à l'arbre syntaxique et active l'action de la première règle `l : l N`.

```
on ajoute 2 à la liste
on ajoute 1 à la liste
```

L'arbre a été construit dans l'autre sens, comme indiqué dans la figure 7.2

7.2 Utilisation de la pile par l'analyseur

L'automate construit par Yacc fonctionne en plaçant les mots dans une pile. Quand l'automate a déterminé qu'il devait appliquer une règle, il dépile les nœuds qui correspondent à la partie droite et empile à la place le nœud qui apparait en partie gauche.

Le détail du mécanisme est évoqué par la figure 7.3, qui montre les étapes de la construction de l'arbre syntaxique pour la phrase $a + 1 \times 3 + 4$ avec la grammaire :

```
%term N
%left '+'
%left '*'
%%
e : N
        { $$ = $1; }
  | e '+' e
        { $$ = $1 + $3; }
  | e '*' e
        { $$ = $1 * $3; }
  ;
```

7.3 Fermetures LR(n)

Pour déterminer les opérations à effectuer sur la pile, Yacc construit un automate dont chaque état contient la liste des règles de grammaires *qui vont peut-être s'appliquer*. On appelle cet automate une *fermeture LR* (ou *closure* en anglais).

L'idée générale est la suivante : il est impossible de déterminer la règle qu'on est en train d'appliquer dès le départ de la lecture des mots qui composent sa partie droite ; à la place, Yacc construit un automate dont chaque état contient la liste des règles candidates à l'application, avec l'endroit où il en est de la lecture de la partie droite de cette règle. Quand il a trouvé toute la partie droite, alors il peut la *réduire* en la remplaçant dans la pile par le symbole de la partie gauche.

Pour démarrer, il construit un état 0 avec une règle destinée à reconnaître la phrase complète : elle aura en partie droite le symbole de départ suivi du marqueur de fin (noté $), et il indique là où il en est de la reconnaissance de

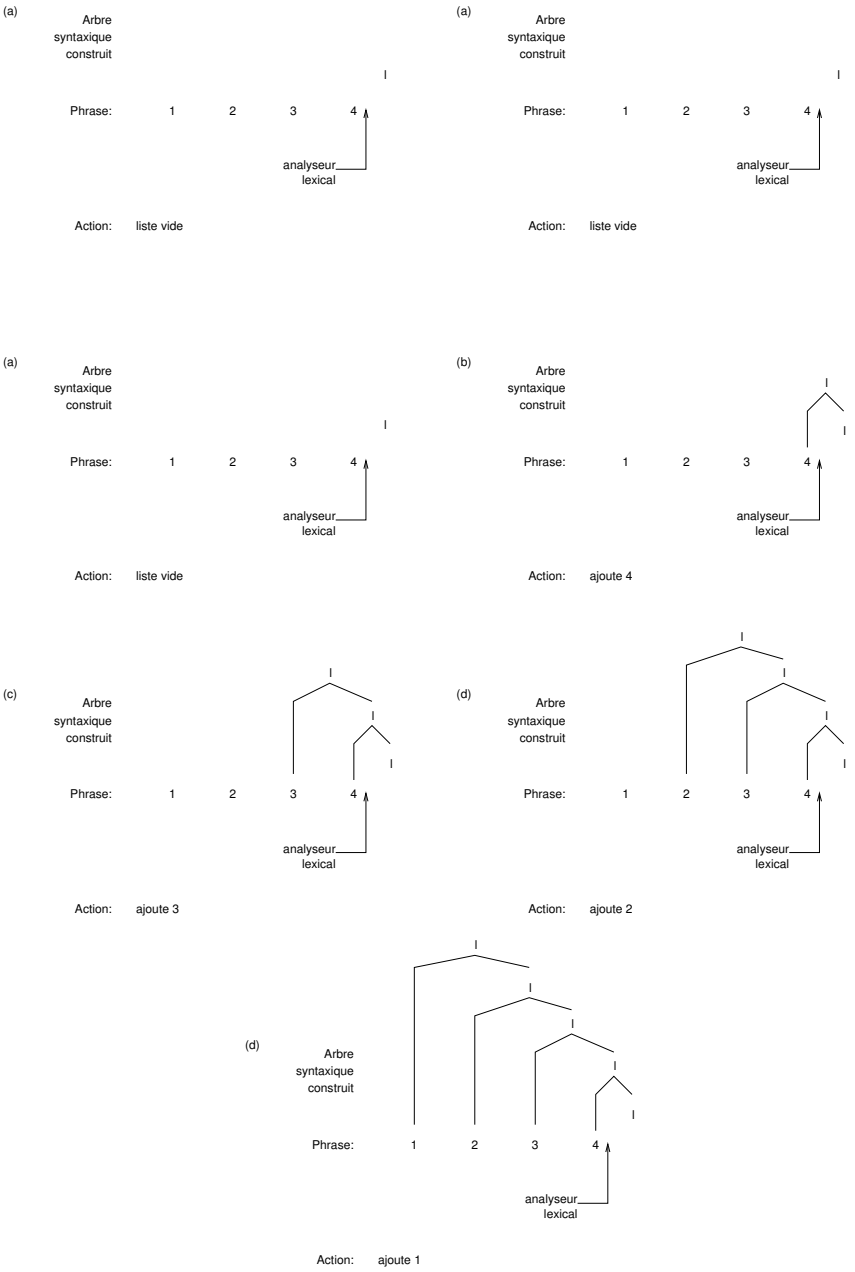


FIGURE 7.2 – La construction de l'arbre après l'échange des deux éléments de la partie droite de la première règle : tous les mots sont lus avant que l'analyseur ne puisse commencer à effectuer les réductions, d'abord par la seconde règle (a) puis par la première en ajoutant les éléments un par un à la liste de la droite vers la gauche. Les actions impriment donc les nombres dans l'ordre inverse de celui où ils apparaissent dans la phrase de départ.

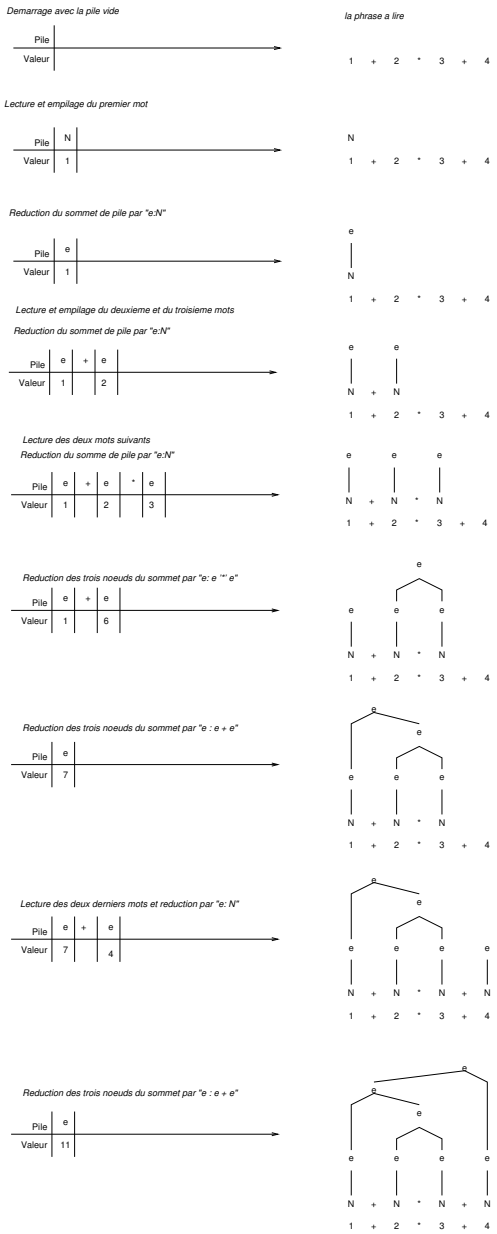
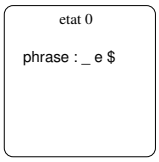
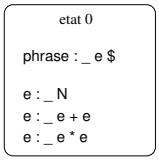


FIGURE 7.3 – La construction de l'arbre syntaxique de la phrase $1 + 2 \times 3 + 4$ correspond à une séquence de manipulations sur la pile de l'automate. Pour faire tenir la figure sur une page je n'ai indiqué que la première réduction de N en e.

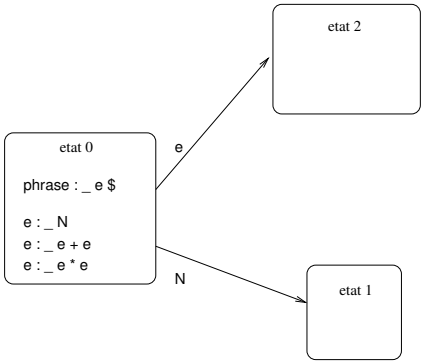
cette règle (je l'indiquerai dans les schémas avec le caractère _). L'état de départ de l'automate contient donc :



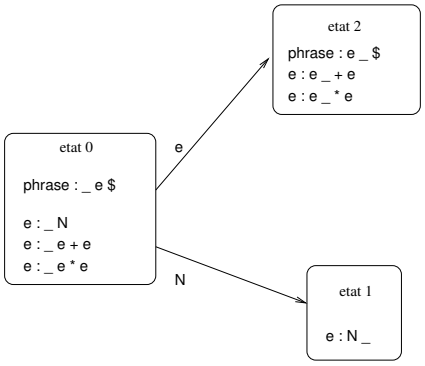
Dans cette état l'automate attend un e : il doit donc aussi être prêt à appliquer, depuis le début, les trois règles qui décrivent la construction des e : cela conduit à ajouter les trois règles `e : N`, `e : e + N` et `e : e * e` en insérant le marqueur de lecture au tout début de la partie droite (on n'a encore rien lu de cette partie droite).



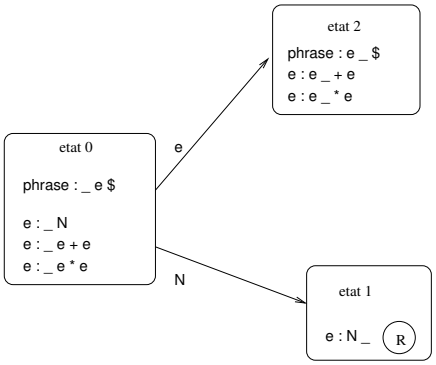
A partir de cet état 0, l'automate doit être prêt à recevoir n'importe lequel des symboles (terminaux ou pas) qui apparaissent juste à droite du marqueur de lecture (ici `e` et `N`). Il ajoute donc deux transitions vers deux nouveaux états, qui seront prises respectivement quand il rencontrera un `e` ou un `N` :



Pour déterminer les règles qui peuvent être en train de s'appliquer dans ces états, il suffit d'extraire des règles en cours d'application dans l'état 0 celle où un `e` apparaît juste à droite du point de lecture pour l'état 2 et celle avec un `N` pour l'état 1. Cela donne :

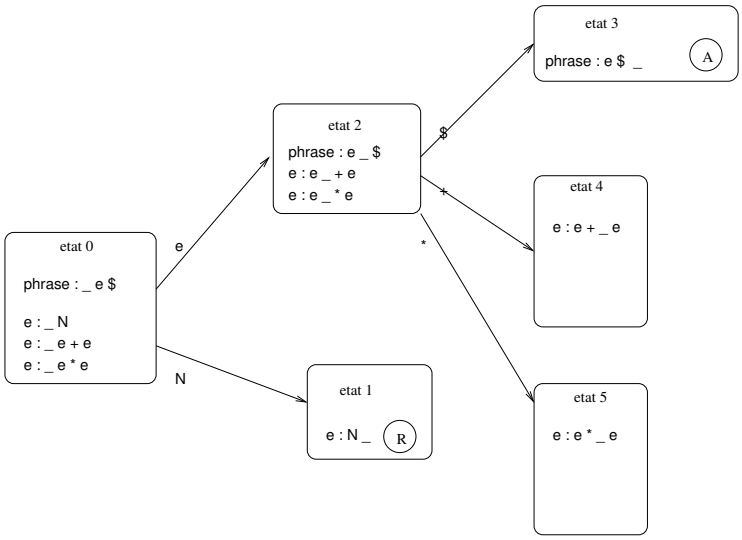


Dans l'état 2, on ne sait encore laquelle des trois règles il va falloir appliquer. Dans l'état 1, il n'y a qu'une seule règle à appliquer et on a complètement vu sa partie droite : cela signifie qu'on peut *réduire* (c'est à dire appliquer) la règle, en retirant le `N` de la pile et en empilant un `e` à la place. Je le note avec un `R` entouré d'un cercle pour *reduce* (*réduire* en français).

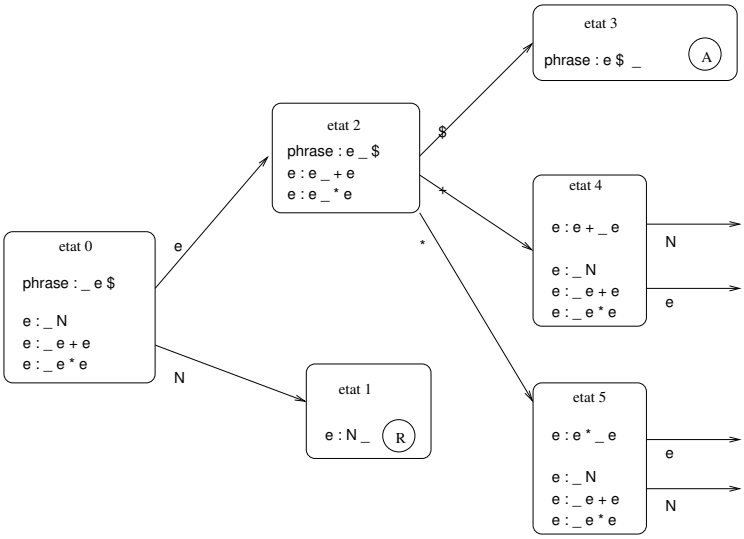


Dans l'état 1, il n'y a pas de règle avec un symbole à droite du marqueur de lecture : il n'y a donc pas de transition qui parte et on a donc terminé le travail pour cet état.

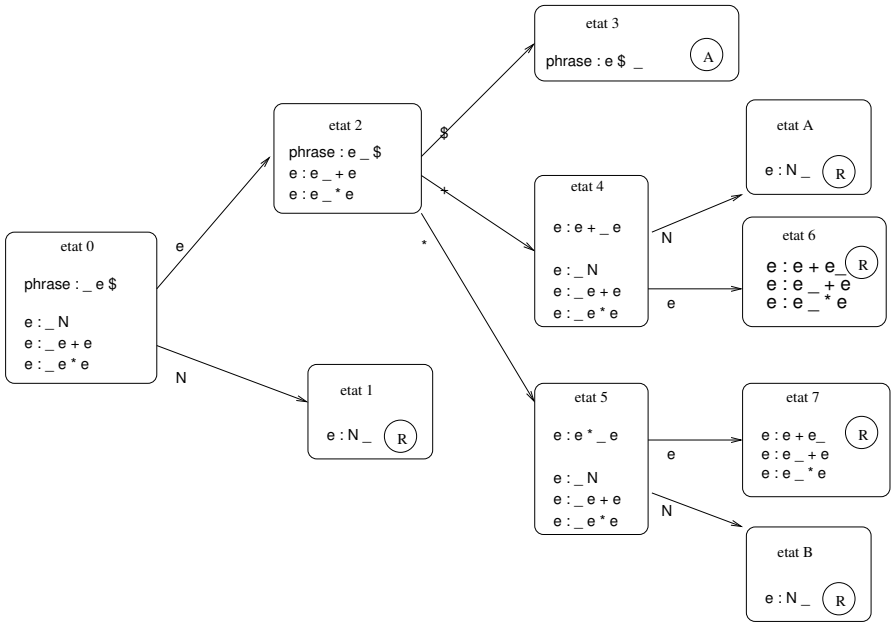
Depuis l'état 2 on à des règles avec `$`, `+` et `*` juste à droite du marqueur de lecture : il va donc en partir trois transitions étiquetées avec `$`, `+` et `*` vers trois nouveaux états :



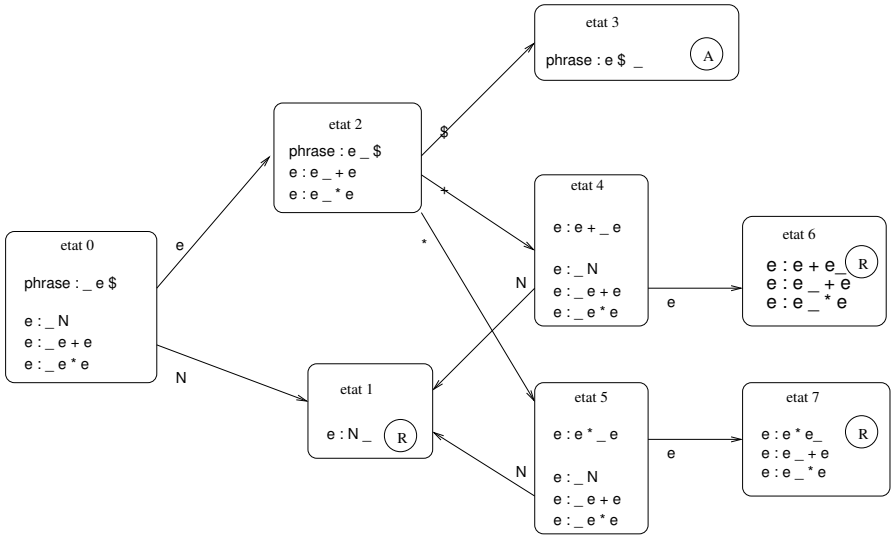
Dans l'état 3, on a reconnu complètement la phrase et l'analyseur a terminé son travail. C'est une action spéciale qu'on appelle *accepter* ; je la marque sur le schéma avec `A` entouré d'un cercle. Dans les états 4 et 5, la prochaine chose qu'on devra lire est un `e` : il faut donc ajouter les règles de la grammaire qui décrivent la structure d'un `e` avec le marqueur de lecture tout au début de la partie droite. Cela implique qu'à partir des états 4 et 5, on va avoir des transitions étiquetées par `e` et par `N` :



On recommence le travail pour les quatre transitions qui partent des états 4 et 5, pour obtenir le nouveau graphe :



On constate alors que les états A et B sont exactement identiques à l'état 1 : ils contiennent les mêmes règles avec le marqueur de lecture au même endroit. Au lieu de les créer, on peut donc faire pointer les transitions étiquetées avec `N` depuis les états 4 et 5 vers l'état 1 :



Pour terminer le travail, il ne reste plus qu'à ajouter les transitions depuis les états 6 et 7 sur le + et le − : on constate qu'elles emmènent vers des états iden-

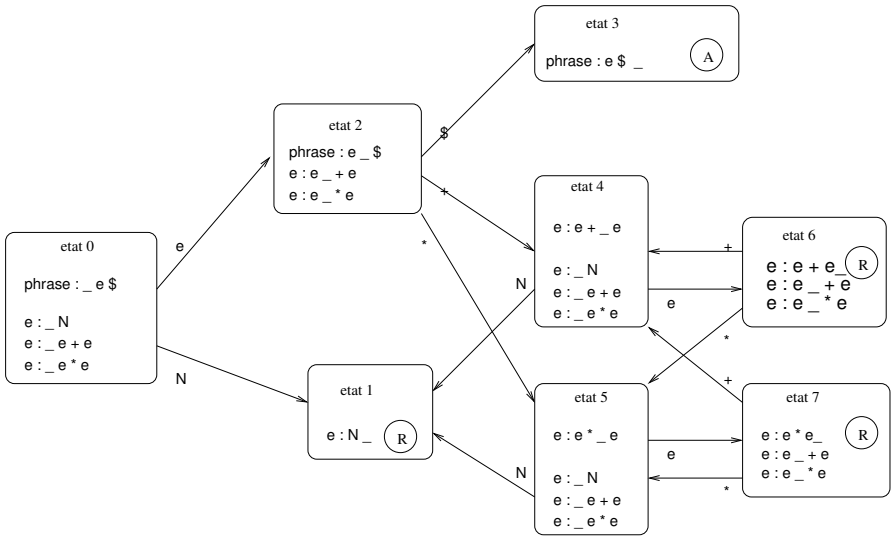


FIGURE 7.4 – La fermeture $LR(0)$ de la grammaire `e : N | e '+' e | e '*' e`.

tiques aux états 4 et 5. On a construit la *fermeture* $LR(0)$ de notre grammaire (figure 7.4).

Nous venons de faire tourner à la main un algorithme de construction de la fermeture LR(0) ; on peut le décrire d'une façon plus générale avec :

```
initialiser un état initial avec Phrase : _ depart $.
placer l'état initial dans la liste des états à traiter
pour chaque état restant à traiter
  départ = état
  pour chaque symbole qui apparait à droite du marqueur dans départ
    courant = symbole
    arrivée = nouvel état
    ajouter une transition de départ vers arrivée
    étiqueter la transition avec le symbole
    pour chaque règle de l'état de départ
      si courant apparait juste à droite du marqueur
        ajouter la règle à l'état d'arrivée
        y deplacer le marqueur d'une position vers la droite
    pour chaque symbole qui apparait à droite du marqueur dans arrivée
      ajouter toutes les règles qui ont symbole comme partie gauche
      y placer le marqueur au début de la partie droite
    si l'etat d'arrivée contient les mêmes règles qu'un état déja construit
      faire pointer la transition vers l'état déja construit
      supprimer l'état d'arrivee
```

```
    sinon
      ajouter l'état à la liste des états à traiter
```

On a la garantie que l'algorithme se termine parce que le nombre de règles est fini et que chaque règle contient un nombre fini de symboles (et donc de positions pour le marqueur de lecture).

Une fois la fermeture construite, on la parcourt simplement comme un automate presque ordinaire, avec :

```
placer l'état de départ dans la pile
lire un mot et l'empiler
boucler sur :
  empiler l'etat indiqué par l'état et le symbole du sommet de pile
  si l'état indique qu'on accepte
    c'est fini avec succès.
  si l'état indique une réduction par une règle
    dépiler la partie droite de la règle (avec ses états)
    empiler la partie gauche
  ou bien s'il y a une transition qui part de l'état
    lire un mot et l'empiler
```

A la différence de la fermeture, l'automate est infini parce que son état est aussi déterminé par le contenu de la pile, qui a une profondeur infinie (potentiellement, ou du moins dans les limites de la mémoire disponible).

Dans le cas où l'état et le symbole du sommet de pile n'indiquent pas une transition, alors c'est qu'on a une erreur de syntaxe.

Je présente plus loin un exemple de fonctionnement de l'automate, une fois résolue la question des conflits encore présents dans la grammaire.

7.3.1 Le fichier `output`

Quand on appelle Yacc avec l'option `-v`, il décrit la fermeture de la grammaire dans un fichier `y.output`. (Quand Bison traite un fichier `foo.y`, il place la description de la fermeture dans le fichier `foo.output`.)

7.3.2 Conflits shift-reduce

La fermeture $LR(0)$ que nous avons construite présente un problème dans les états 6 et 7 : l'automate peut choisir soit de réduire par la règle complètement vue, soit d'empiler le mot suivant si c'est un $+$ ou une $*$. On a là un *conflit shift-reduce*.

Ces conflits sont la peste quand on écrit une grammaire. Ils sont levés de deux façons : d'une part, Yacc utilise un automate LALR(1), plus puissant que le LR(0) présenté dans l'exemple plus haut ; d'autre part il applique des règles de détermination aux ambiguïtés résiduelles de la grammaire.

7.3.3 Fermeture $LR(1)$ et $LALR(1)$

Sur le même modèle que la fermeture $LR(0)$, on peut construire une fermeture $LR(1)$ en ajoutant à chaque règle non seulement la position du marqueur de lecture courant, mais aussi le lexème qui est susceptible de la suivre. De cette manière, il est le plus souvent possible à l'analyseur choisir entre *shift* et *reduce* sans ambigüité.

Au lieu de n'utiliser qu'un seul symbole qui apparait à droite de la règle, on peut en utiliser n et on a alors un analyseur $LR(n)$. L'inconvénient est que les tables utilisées pour décrire la fermeture deviennent énormes quand n grandit.

Yacc utilise une version simplifiée de l'analyseur $LR(1)$ dans laquelle les états similaires sont regroupés, et qu'on appelle $LALR(1)$. Pour les langages de programmation, ces analyseurs font en général l'affaire.

Le *Dragon Book* contient une description détaillée de la construction des fermetures $LR(1)$ et $LALR(1)$ et des propriétés de leurs analyseurs. Je ne pense pas que ce soit essentiel de l'étudier en détail pour utiliser Yacc avec profit.

7.3.4 L'exemple

Quand on fait tourner l'algorithme sur la phrase $1+2\times3+4$, avec l'automate de la figure 5.10, l'analyseur passe dans les états suivants. (On suppose pour le moment que l'automate devine quand il doit effectuer une réduction et quand il doit empiler le mot suivant ; la manière dont le choix est fait est expliqué juste après).

Empilage de l'état de départ.
Pile : 0

Lecture du 1 par l'analyseur lexical qui renvoie N ; empilage du N.
Pile : 0 N

Transition avec N de l'état 0 dans l'état 1.
Pile : 0 N 1

Réduction du N en e.
Pile : 0 e

Transition avec e de l'état 0 dans l'état 2.
Pile : 0 e 2

Lecture du + par l'analyseur lexical qui renvoie + ; empilage du +.
Pile : 0 e 2 +

Transition avec + de l'état 2 dans l'état 4.
Pile : 0 e 2 + 4

Lecture du 2 par l'analyseur lexical qui renvoie N ; empilage du N.
Pile : 0 e 2 + 4 N

Transition avec N de l'état 4 dans l'état 1.
Pile : 0 e 2 + 4 N 1

Réduction du N en e.
Pile : 0 e 2 + 4 e

Transition avec e de l'état 4 à l'état 6.
Pile : 0 e 2 + 4 e 6

Lecture du $\times$ par l'analyseur lexical qui renvoie $*$; empilage de l'$*$.
Pile : 0 e 2 + 4 e 6 $*$

Transition avec $*$ de l'état 6 vers l'état 5.
Pile : 0 e 2 + 4 e 6 $*$ 5

Lecture du 3 par l'analyseur lexical qui renvoie N ; empilage du N.
Pile : 0 e 2 + 4 e 6 $*$ 5 N

Transition avec N de l'état 4 dans l'état 1.
Pile : 0 e 2 + 4 e 6 $*$ 5 N 1

Réduction du N en e.
Pile : 0 e 2 + 4 e 6 $*$ 5 e

Transition avec e de l'état 5 dans l'état 7.
Pile : 0 e 2 + 4 e 6 $*$ 5 e 7

Réduction du $e * e$ en e.
Pile : 0 e 2 + 4 e

Transition sur e de l'état 4 dans l'état 6.
Pile : 0 e 2 + 4 e 6

Réduction de $e + e$ en e.
Pile : 0 e

Transition avec e de l'état 0 dans l'état 2.
Pile : 0 e 2

Lecture du + par l'analyseur lexical qui renvoie + ; empilage du +.
Pile : 0 e 2 +

Transition avec + de l'état 2 dans l'état 4.
Pile : 0 e 2 + 4

Lecture du 4 par l'analyseur lexical qui renvoie N ; empilage du N.
Pile : 0 e 2 + 4 N

Transition avec N de l'état 4 dans l'état 1.
Pile : 0 e 2 + 4 N

Réduction du N en e
Pile : 0 e 2 + 4 e

Transition avec e de l'état 4 dans l'état 6
Pile : 0 e 2 + 4 e 6

Réduction de $e + e$ en e.
Pile : 0 e

Transition avec e de l'état 0 dans l'état 2
Pile : 0 e 2

Lecture de la fin de fichier par l'analyseur lexical qui renvoie \$; empilage du

$.
Pile : 0 e 2 $

Transition avec $ de l'état 2 dans l'état 3
Pile : 0 e 2 $ 3

Acceptation : la phrase est bien formée.

7.4 Exercices

Les exercices qui suivent sont importants pour la maîtrise complète de la matière du cours. Ils mettent en évidence la façon dont les problèmes de précédence et d'associativité se traduisent en conflits pour Yacc.

Note : pour écrire des dérivations d'arbre, le plus intuitif est d'utiliser une feuille, d'y écrire des mots, puis de dessiner l'arbre ; cependant, si on souhaite utiliser un clavier et placer la dérivation dans un fichier de texte sans image, c'est plus facile de placer traduire l'arbre sous la forme d'une expression. Par exemple, avec la grammaire de la page 112, la dérivation de la phrase $1+2\times3+4$ de la figure 7.3 page 114 peut s'écrire avec :

```
e=(e=(e=(N=(1))+e=(e=(N=(2))*e=(N=(3))))+e=(N=(4)))
```

Ex. 7.1 — (Associativité des opérateurs et conflits shift-reduce) Soit la grammaire :

```
p : A
  | p X p
  ;
```

1. Montrer que la grammaire est ambigüe en dérivant deux arbres syntaxiques différents pour la phrase `AXAXA`.
2. (hors sujet) Combien d'arbres syntaxiques différents pouvez-vous dériver pour une phrase constituée d'un `A` suivi de n fois `XA` ? (la question précédente répond à la question pour le cas où n vaut 2)
3. Comment Yacc vous prévient-il que cette grammaire est ambigüe ?
4. À partir du fichier `.output` fabriqué par Yacc, construire l'automate $LR(0)$ correspondant à cette grammaire, et marquer l'état où l'ambigüité se manifeste.
5. Refaire les mêmes questions pour la grammaire

   ```
   p : A
     | A X p
     ;
   ```

   L'opérateur `X` est-il ici associatif à droite ou à gauche ?
6. Refaire les mêmes questions pour la grammaire

   ```
   p : A
   ```

```
    | p X A
    ;
```

L'opérateur X est-il ici associatif à droite ou à gauche ?

7. Reprendre la grammaire de la question 1, et spécifier à Yacc que l'opérateur X est associatif à droite, puis à gauche (avec l'indication `%left X` ou `%right X` dans la partie déclaration du fichier Yacc). Comparer les automates obtenus avec ceux des réponses aux questions 1, 4 et 5. Quelle conclusion en tirer ?

Ex. 7.2 — ($LR(0)$, $LALR(1)$ et $LR(1)$) Soit la grammaire :

```
p : a C
  | b D
  ;
a : X
  ;
b : X
  ;
```

1. Énumérer les phrases que cette grammaire permet de construire. La grammaire est-elle ambigüe ?
2. Construire la fermeture LR(0). La fermeture LR(0) indique-t-elle une ambiguïté ? (Si c'est le cas dans quel état ?)
3. Yacc pense-t-il que cette grammaire est ambigüe ? Pourquoi ?
4. Mêmes questions pour la grammaire :

```
P : a Y C
  | b Y D
  ;
a : X
  ;
b : X
  ;
```

Ex. 7.3 — (Variations sur les listes)

1. Écrire une grammaire pour Yacc qui décrive une liste non vide de X.
2. Écrire une grammaire pour Yacc qui décrive une liste de X qui peut être vide.
3. Écrire une grammaire pour Yacc qui décrive une liste non vide de X séparés par des virgules
4. Écrire une grammaire pour Yacc qui décrive une liste de X séparés par des virgules qui peut être vide.

7.5 Les ambiguïtés résiduelles

Quand il reste des ambiguïtés dans une grammaire, Yacc donne un message d'erreur annonçant le nombre de conflits shift-reduce et de conflits reduce-reduce mais produit quand même un analyseur en adoptant une règle par défaut pour résoudre les conflits.

7.6 Conflits shift-reduce, le *dangling else*

Quand il rencontre un conflit shift-reduce, Yacc choisit le shift. (Cela signifie accessoirement que les opérateurs sont par défaut associatifs à droite, mais de toute manière il est préférable de définir l'associativité des opérateurs avec `%left` ou `%right`.)

Cette règle permet de résoudre de la façon souhaitable le problème du *dangling else* (le *sinon pendant* en français) : dans un fragment de programme comme

```
if (x) if (y) e1(); else e2();
```

la question est celle du `if` avec lequel associer le `else`.

Le choix du shift plutôt que du reduce permet d'avoir l'interprétation usuelle des langages de programmation dans laquelle le `else` est associé avec le dernier `if`.

```
if (x)
    if (y)
        e2();
    else
        e3();
```

Notez que depuis quelques années quand gcc rencontre une forme comme celle-là, il suggère de rajouter des accolades.

7.6.1 Conflits reduce-reduce

Quand la grammaire présente des conflits reduce-reduce, Yacc choisit de réduire par la première règle qui apparait dans la grammaire, comme on peut s'en rendre compte avec la grammaire :

```
%term A
%%
p : x | y;
x : A { printf("A réduit en X\n"); };
y : A { printf("A réduit en Y\n"); };
%%
```

```
# include <stdio.h>
int main(){ return yyparse(); }
int yylex(){ static int t[] = {A, 0}, it; return t[it++]; }
int yyerror(char * s){ return fprintf(stderr, "%s\n", s); }
```

Notez que c'est la définition de la règle par laquelle réduire et non celle qui utilise le terminal qui est utilisée pour lever l'ambiguïté. On obtient le même résultat si on remplace la première règle par

```
p : y | x;
```

7.6.2 Laisser des conflits dans sa grammaire

En première approximation, on ne doit pas laisser de conflits reduce-reduce dans une grammaire. C'est le plus souvent le signe qu'on ne la maîtrise pas et cela risque de conduire à des catastrophes.

On peut laisser des conflits shift-reduce à condition de bien comprendre ce qui les produit. C'est notamment le cas pour le *dangling else*. Bison possède une directive `%expect` qui indique combien la grammaire possède de conflits et permet d'éviter un message d'erreur quand le nombre de conflits correspond à ce qui est attendu.

Dans tous les cas, il *faut* documenter les conflits, en expliquant ce qui les produit, la façon dont ils sont résolus par Yacc et la raison pour laquelle on les conserve.

7.7 Des détails supplémentaires sur Yacc

J'aborde ici des points sur les grammaires Yacc que j'ai laissé de coté jusqu'à maintenant. Ils me semblent moins important mais peuvent être nécessaires pour lire les grammaires écrites par d'autres.

7.7.1 Définition des symboles terminaux

Pour faciliter leur manipulation, Yacc affecte des valeurs aux symboles de la grammaire, sous forme de valeurs numériques définies avec des `#define`. Par exemple la déclaration de

```
%term NBRE
```

va produire dans le fichier `y.tab.c` une ligne du type

```
#define NBRE 318
```

Ces valeurs numériques doivent être accessibles à l'analyseur lexical, puisque c'est ce qu'il doit renvoyer quand il rencontre un `NBRE`. Si l'analyseur lexical est

placé dans la troisième partie du fichier `.y`, ces valeurs sont définies ; en revanche s'il est dans un autre fichier, il faut appeler Yacc avec l'option `-h` ; il place alors ces définitions dans un fichier nommé `y.tab.h` (ou `foo.tab.h` avec Bison quand il traite un fichier nommé `foo.y`). Ce fichier doit ensuite être inclus dans le fichier où est défini l'analyseur lexical.

A mon avis, dans les projets de taille moyenne, il est beaucoup plus sain d'inclure l'analyseur dans le fichier `.y` que de passer par `y.tab.h`.

7.7.2 Les actions au milieu des règles

```
p : a b { action1 } c d { action2 }
```

équivaut à

```
p : a b x c d { action2 } ;
x : /* rien */ { action1 } ;
```

y compris pour ce qui concerne les `$`n dans `action2`.

7.7.3 Références dans la pile

On peut faire référence à `$0`, `$-1` etc. si on sait quel est le contexte de la pile de Yacc dans laquelle une règle est utilisée.

7.7.4 Les nœuds de type inconnu de Yacc

Avec les actions au milieu des règles et les références dans la pile, on ne peut pas spécifier le type de certains nœuds. Pour ceux là (*et ceux là seulement*) on peut définir leur type avec `$<type>`n.

7.7.5 `%token`

On peut déclarer les symboles terminaux avec `%token` au lieu de `%term`. (C'était même au départ l'unique manière de les déclarer, `%term` a été ajouté ensuite comme un synonyme.) Un *token* (traduction littérale en français : un *jeton*) est quelque chose qu'on ne peut pas découper. C'est une façon informelle de nommer les *lexèmes*, les mots qu'identifie l'analyseur lexical.

7.7.6 `%noassoc`

De même qu'on peut spécifier qu'un opérateur est associatif à gauche ou à droite, on peut aussi indiquer qu'il *n'est pas* associatif avec `%nonassoc`.

7.8 Le reste

Il existe une autre catégorie importante d'analyseurs syntaxiques qui ne sont pas traités dans le cours : les analyseurs descendants, dans lesquels on construit l'arbre à partir de la racine. C'est plus plus facile d'écrire un analyseur descendant qu'un analyseur ascendant, mais comme de toute façon on n'écrit pas les analyseurs ascendants (c'est Yacc qui le fait), on peut probablement en ignorer les détails.

Chapitre 8

L'analyseur syntaxique de Gcc

Ce court chapitre revient sur la question des grammaires. Nous y étudions la grammaire du langage C pour Yacc telle qu'elle apparait dans la version 2.95 du compilateur Gcc.

J'ai extrait cette grammaire du fichier source de gcc nommé `c-parse.y`; j'en ai retiré toutes les actions, très légèrement simplifié les règles et ajouté des numéros de lignes. On trouvera le résultat en annexe.

La suite de ce petit chapitre étudie (de façon incomplète) cette grammaire. Le but est double : d'une part cela conduit à regarder avec quelque détails une grammaire réaliste d'un compilateur authentique ; d'autre part cela nous sert aussi à explorer des aspects peu apparents du langage C et les extensions au langage que Gcc supportait dans sa version 2.95.

8.1 Les déclarations

Le type de valeur attaché aux lexèmes et aux noeuds de l'arbre syntaxique est défini par le `%union` des lignes 10 et 11. La valeur de la plupart des noeuds est du type `ttype` : un arbre qui représente l'arbre syntaxique. Cet arbre sera ensuite presque immédiatement traduit en un langage intermédiaire appelé *RTL* comme *Register Transfer Language*.

Les lignes 13–43 contiennent les noms qui ne sont pas des mots réservés : les choses sont un peu compliquées à cause des typedefs et de la multiplications des types possibles. L'analyseur lexical distingue les noms qui ne peuvent pas désigner un type (ce sont des *IDENTIFIERs*), ceux qui ont été définis avec `typedef` et peuvent donc désigner un type (ou pas) (ce sont des *TYPENAME*, ceux qui qualifient un type (comme `const` ou `volatile`) et les *storage class specifier* comme `static` ou `auto`.

Toutes les constantes, entières ou flottantes, seront signalées comme des *CONSTANT*, sauf les chaînes de caractères qui seront reconnues comme des *STRING*. Les points de suspensions sont un *ELLIPSIS*.

Les mots clefs qui ne définissent pas de type sont définis aux lignes 47–50. *ASM*, *TYPEOF* et *ALIGNOF* sont des extensions de gcc qui n'appartiennent pas au langage C, de même que les terminaux des lignes 49 et 50.

Les opérateurs (avec leurs précédences) sont définis aux lignes 60 à 74. *ASSIGN* représente les opérateurs qui combinent une opération et une affectation comme `+=` ou `«=`. Les noms des autres sont évidents.

8.2 Les règles de la grammaire

Quatre grands morceaux : programme, instructions, expressions, déclarations. Seules les déclarations sont vraiment compliquées.

8.2.1 Programme

Aux lignes 112–126 : un programme est une liste (peut-être vide) de déclaration de fonctions (`fndef`) et de déclarations de données (`datadef`), ou des extensions dont on ne parle pas.

8.2.2 Les instructions

A partir de la ligne 684 jusqu'à la ligne 834 on trouve les instructions. Les choses sont un peu compliquées à cause d'une extension de gcc qui permet de définir des étiquettes (avec le mot clef `label`).

Les blocs d'instructions sont définis aux lignes 729–733 : ce sont soit rien que des accolades (ligne 729), soit des déclarations et éventuellement des instructions (ligne 730), soit pas de déclaration et des instructions (ligne 732). La règle de la ligne 731 est là pour aider à la récupération d'erreurs.

Les différentes variétés d'instructions sont définies comme des `stmt` (pour *statement*) lignes 766–793,

Noter que les deux formes de `return`, avec et sans valeur, sont définis par deux règles différentes (780 et 781). L'extension `asm` utilise quatre règles et on découvre à la ligne 791 que gcc autorise le calcul des étiquettes sur lesquelles faire des `goto` (ce que ne permet pas C).

Les différentes sortes d'étiquettes sont aux lignes 799–803. Il y a ici aussi une extension de GCC à la ligne 800 : on peut étiqueter des `case` dans un `switch` avec des choses comme `case 'a'...'z'` pour reconnaître des intervalles de valeurs.

8.2.3 Les expressions

Les expressions sont définies avec des symboles intermédiaires pour forcer les niveaux de précédence.

Le premier niveau, avec la précédence la plus forte, est celui des `primary` (lignes 224 et suivantes) : identificateurs, constantes, expression entre parenthèses, appels de fonction (ligne 231), utilisation de tableaux (ligne 232) ou références à des champs de structures avec `.` (ligne 233) et `->` (ligne 234), `++` et finalement `--` postfixés.

Le deuxième niveau force la réduction des autres opérateurs unaires en `unary_expr` (lignes 177–193). Noter comment presque tous les opérateurs unaires simples sont réduits par la règle de la ligne 181, qui utilise le `unop` défini aux lignes 155–162 (mais pas l'`*`, qui peut s'appliquer à une expression qui contient un cast). Noter aussi comment les deux formes du `sizeof` (avec un type comme dans `sizeof(int)` ou avec une expression comme dans `sizeof 1`) se traduit par deux règles différentes aux lignes 188 et 189.

Le troisième niveau contient les conversions forcées (les *cast* en C), dans les noeuds de type `cast_expr` (lignes 196–200).

Le quatrième niveau contient presque tous les opérateurs binaires, sauf la virgule : ce sont les `expr_no_commas` (lignes 202–221). La virgule a un statut spécial, puisqu'elle peut apparaître à la fois dans une liste d'instruction (comme dans l'expression `i = 0, j = MAX`) et dans une liste d'arguments (comme dans `foo(1, 2)`).

Le niveau suivant est celui des *exprlist* (avec sa variante *nonull_exprlist*) (lignes 167–175) dans lequel on intègre les virgules, puis finalement l'*expr* de la ligne 164.

8.2.4 Les déclarations

La plus grande partie du reste de la grammaire est consacré aux déclaration de variables, avec éventuellement des valeurs initiales. Je ne détaille pas cette partie qui est vraiment complexe. La complexité des formes de déclarations du langage C est l'un des problèmes les plus fondamentaux du langage C.

8.3 Exercices

Ex. 8.1 — Comment déclarer en C un pointeur sur une fonction avec deux arguments entiers qui renvoie un nombre en virgule flottante ?

Ex. 8.2 — Comment déclarer en C un pointeur sur une fonction dont l'unique argument est une chaîne de caractère et qui renvoie un pointeur sur une fonction avec un argument flottant qui renvoie une pointeur sur une chaîne de caractère ?

Ex. 8.3 — Quel arbre syntaxique gcc produira-t-il quand il réduira le frag-

ment de code `return t[i++] * *p;` en instruction ?

Chapitre 9

La sémantique, la génération de code

Attention, la *sémantique des langages de programmation* désigne quand on l'utilise seul tout un sous-domaine de l'informatique très formel et passablement stérile. Dans le contexte de la compilation, il a un autre sens : c'est tout ce qu'il est nécessaire d'ajouter à l'information contenue dans l'arbre syntaxique pour pouvoir générer du code correct.

En pratique, on met dans la sémantique tout ce qu'on ne sait pas faire aisément avec l'analyseur syntaxique. Exemples : déclarations de variables, constances des expressions dans les initialisations...

9.1 Les grammaires attribuées

Je n'en parle (presque) pas. L'idée est d'avoir une manière synthétique de décrire les transmissions d'informations (notamment de type) dans l'arbre syntaxique. Il n'y a pas d'outil répandu comme Yacc pour le faire. Il y a une bonne description du travail dans le *Dragon Book*.

9.2 Les conversions

Le principal travail de la sémantique du C, ce sont les conversions.

Comparer `float x = 8/3;` avec `float x = 8/3.;`. Idem avec `int x = 8/3;` avec `int x = 8/3.;`.

Principe général du C : pour une opération entre deux types différents, on convertit le moins précis dans le plus précis pour faire l'opération.

Lors d'un appel de fonction, les floats sont passés comme des doubles, les

petits entiers (`char` et `short`) comme des `int`.

Les calculs sur les entiers de taille inférieure ou égale à `int` se font dans des `int` (ce qui signifie que c'est un peu coûteux d'utiliser des `short` : il va falloir convertir les opérandes en `int` pour faire l'opération puis convertir le résultat en `short`).

9.3 La génération de code, `ppcm`

Ce n'est pas bien compliqué de générer du code à partir de l'arbre syntaxique. Pour ancrer le propos dans le réel, je vais présenter cette partie sous la forme du commentaire d'un minuscule compilateur pour un sous-ensemble de C qui génère du code pour le processeur Intel 386. Le code est présenté (en partie) en annexe avec des lignes numérotées et est il est présent (en totalité) dans les documents associés.

Attention, la simplicité extrême de ce compilateur n'a été obtenue qu'en faisant l'impasse sur certaines améliorations élémentaires ; il ne faut pas l'utiliser pour autre chose que pour une démonstration.

9.3.1 Le langage source

Le langage source traité par le compilateur est un sous-ensemble du langage C.

La principale simplification est que le seul type de donnée est l'entier. La ménagerie des opérateurs de C est aussi réduite au minimum.

Il permet de définir une fonction vide avec :

```
main()
{
}
```

Il contient des boucles `while` comme dans :

```
fib_iter(n)
{
  int a, b, c;

  a = 0;
  b = 1;
  c = 0;
  while(c != n){
    c = c + 1;
    a = a + b;
    b = a - b;
  }
```

```
  return a;
}
```

On peut appeler des fonctions et utiliser les valeurs qu'elles renvoient comme dans :

```
main(){
  int c;

  while((c = getchar()) != -1)
    putchar(c);
}
```

ou bien

```
fib_rec(n)
{
  if (n != 0)
    if (n != 1)
      return fib_rec(n - 1)
             + fib_rec(n - 2);
  return n;
}
```

Comme on peut le voir dans ces exemples, les programmes du langage peuvent tous être compilés par un compilateur C ordinaire, sans erreur mais avec des *warnings* (des messages d'alertes) parce qu'on ne précise ni le type des valeurs renvoyées ni celui des arguments des fonctions (ce sont toujours des `int`).

9.3.2 La représentation des expressions

Une expression est représentée par une structure `expr`, définie dans le fichier `ppcm.h` (lignes 13-19) :

```
1 struct expr {
2   int position;                   /* en memoire, son index via %ebp */
3   char * nom;                     /* le nom de la variable (le cas echeant) */
4 };
```

Elle contient d'une part le nom de l'expression (si elle correspond à un argument ou une variable ; sinon elle correspond à une expression intermédiaire et le champs contient 0) et l'endroit où elle se trouve dans la pile, par rapport au frame pointer de la fonction. Il n'y a pas moyen d'avoir de variable globales.

Chaque structure `expr` utilisée est un élément du tableau du même nom défini dans le fichier `expr.c`, ligne 7. Deux fonctions permettent d'y accéder, `fairepr` et `exprvar`.

```
1  /* fairexpr  --  fabrique une expression (parametre, argument ou temporaire) */
2  struct expr *
3  fairexpr(char * nom){
4    register struct expr * e;
5
6    e = &expr[nexpr++];
7    e->position = posexpr;
8    e->nom = nom;
9    posexpr += incr;
10   return e;
11 }
```

La fonction `fairexpr` est utilisée pour fabriquer une nouvelle expression (fichier `expr.c`, lignes 10–20). On lui passe en argument le nom de l'expression si c'est un argument ou une variable locale, ou `NULL` si c'est une expression temporaire. Elle se contente d'initialiser les deux champs de prochain élément libre de `expr` et de mettre à jours la variable `posexpr` qui indique la position qu'aura la prochaine expression.

```
1  /* exprvar  ---  renvoie l'expression qui designe la variable */
2  struct expr *
3  exprvar(char * s){
4    register struct expr * e, * f;
5
6    for(e = & expr[0], f = e + nexpr; e < f; e += 1)
7      if (/* e->nom != NULL && */ e->nom == s)
8        return e;
9    fprintf(stderr, "Erreur, variable %s introuvable\n", s);
10   return & expr[0];
11 }
```

La fonction `exprvar` (fichier `expr.c`, lignes 22–32) sert à trouver dans le tableau `expr` l'entrée qui décrit une variable déjà déclarée : elle se contente de balayer le tableau et de renvoyer l'adresse de la structure qui la décrit. La raison pour laquelle on peut se contenter d'une comparaison entre pointeurs au lieu d'une comparaison de chaînes de caractères est donnée plus loin.

Il y a également une fonction `reinitexpr` (fichier `expr.c`, lignes 34–38) pour réinitialiser le tableau `expr` après une fonction.

9.3.3 L'analyseur lexical

L'analyseur lexical est défini à l'aide d'un outil que je n'ai pas encore présenté dans le cours : lex (ou flex). Tout se trouve dans le fichier `ppcm.l`, que lex va transformer en une fonction `yylex` définie dans un fichier nommé `lex.yy.c`.

Le gros du travail de reconnaissance est décrit par les lignes 7 à 20 du fichier `ppcm.l` : les mots clefs `if`, `else`, `while`, `int` et `return` ; les noms de variables

ou de fonctions, les constantes entières sous plusieurs formes, l'opérateur d'inégalité `!=` et les caractères spéciaux `-`, `+`, `*`, `/`, `%`, `=`, `(`, `)`, `;`, `{`, `}` et `,`.

La ligne 20 est utilisée pour ignorer les commentaires ordinaires du langage C ouverts avec `/*` et fermés avec `*/`. Elle est là surtout pour montrer un exemple non-trivial d'utilisation de lex et vous pouvez l'ignorer. (Les retours à la ligne dans les commentaires sont d'ailleurs ignorés par l'analyseur lexical, si bien que les numéros de ligne indiqués par les messages d'erreurs qui suivent seront erronés.)

Le reste du fichier `ppcm.l` contient une fonction `chaîne` utilisée pour stocker les noms de fonction et de variables de façon unique. Il y a simplement un tableau qui contient toutes les chaînes ; quand l'analyseur lexical reconnaît un nom de variable ou de fonction, il l'ajoute dans le tableau s'il n'y est pas déjà et dans tous les cas renvoie l'adresse indiquée par le tableau. C'est grâce à cela qu'on peut comparer deux noms avec une simple comparaison de pointeurs à la ligne 22 du fichier `expr.c`, au lieu d'avoir besoin d'utiliser `strcmp` : chaque identifieur n'apparaît qu'une seule fois dans la mémoire. Un programme sérieux utiliserait une table de hash-coding à cet endroit.

9.3.4 La génération de code

La génération de code proprement dite est faite dans le fichier `ppcm.y` à mesure que l'analyseur syntaxique construit l'arbre syntaxique. Je détaille sommairement la génération de code pour les expressions, pour les instructions, puis les prologues et épilogues de fonctions.

Les expressions

Le code pour les expressions va de la ligne 115 à la ligne 172 du fichier `ppcm.y`. Comme mentionné plus haut, la principale caractéristique est que chaque noeud de type `expr` a pour valeur l'adresse d'une structure `expr` qui indique où se trouvera sa valeur quand le code sera utilisé.

Si l'expression est une variable (ligne 115) on utilise la fonction `exprvar` (dans `expr.c`) pour trouver la structure `expr` qui la représente et on place son adresse comme valeur du nœud.

Si l'expression est une constante (ligne 119) on lui alloue un mot mémoire et on génère l'assembleur qui placera cette constante dans le mot mémoire. Ce n'est clairement pas la meilleur manière de générer du bon code : il vaudrait mieux stocker la valeur de la constante dans la structure `expr` et l'utiliser plus tard, mais cela compliquerait le générateur de code.

Si l'expression est une affectation (ligne 123), on fabrique l'assembleur pour recopier le contenu de l'adresse ou se trouvera la valeur de la partie droite à l'adresse où se trouve celle de la partie gauche de l'affectation. On peut noter que cela permet d'écrire des choses anormales comme `1 = a` ou `a + b = c`. Pour l'interdire, il suffirait ici de vérifier que `$1->nom` ne vaut pas `NULL` et désigne

donc bien une variable ou un argument.

Pour tous les autres opérateurs (− unaire, +, −, ×, / et modulo), le schéma est le même : on alloue un temporaire à l'expression (avec `fairexpr(NULL)`) et on fabrique l'assembleur qui calculera sa valeur.

Restent les appels de fonctions : on fabrique l'assembleur pour empiler leurs valeurs à mesure qu'on construit le nœud `listexpr` (lignes 170–172 (noter comment la liste est définie de telle manière que les arguments sont empilés en commençant par la fin). Une fois les arguments empilés, on fabrique l'appel de la fonction (ligne 158), le dépilage des arguments (lignes 159–160) et on alloue un temporaire pour y placer la valeur (peut-être) renvoyée (ligne 161) ; on termine avec l'assembleur pour y recopier la valeur renvoyée (ligne 162).

Évidence Il va sans dire (mais peut-être mieux encore en le disant) que le code est traité une fois, lors de sa compilation. Dans la fonction (stupide) suivante :

```
foo(){
  int x;

  x = 0;
  while(x != 1000000)
    x = x + 1;
}
```

il va y avoir un temporaire attribué pour contenir le résultat du calcul de l'expression `x+1`. (Le temporaire sera ensuite recopié dans `x`). La boucle est effectuée un million de fois (dans cet exemple), mais il n'y a qu'un seul temporaire attribué par le compilateur, quand il a traité l'expression `x+1`.

Les instructions

Le code spécifique pour les instructions se trouve aux lignes 82 à 113, et le début est vraiment facile. Si c'est une instruction vide (ligne 82) il n'y a bien sur rien à faire ; si c'est un bloc d'instructions (ligne 83), rien à faire (l'assembleur aura été généré pendant la construction des nœuds qui forment le bloc) ; si l'instruction se compose d'une expression (lignes 84–85), rien à faire non plus (l'assembleur aura été généré pendant la construction de l'expression).

Pour les `if`, le premier nœud construit est `ifdebut` (lignes 109–113) : la construction du nœud `expr` s'accompagne de la fabrication de l'assembleur pour calculer la valeur de l'expression testée. On fabrique l'assembleur pour comparer cette valeur avec 0 à la ligne 109, et celui qui saute sur le `else` si elle est égale à la ligne 110.

Ensuite, si le test n'a pas de branche `else` (ligne 86), l'assembleur de la branche *si-vrai* sera fabriqué lors de la construction du nœud `instr` et il suffit d'ajouter l'étiquette `else` (ligne 87).

En revanche, si le test possède une branche `else` (lignes 88 et suivantes), il faut produire l'assembleur pour sauter sur la fin et placer l'étiquette `else` juste après la construction du nœud `instr` de la branche *si-vrai* (lignes 89–91) ; ensuite la construction du nœud `instr` de la branche *si-faux* (ligne 92) fabriquera l'assembleur pour le traduire, avant qu'on ajoute l'étiquette de fin (ligne 93).

Ainsi le corps de la fonction (stupide)

```
foo(){
  int x;

  if (x)
    bar();
}
```

sera traduit par

```
                cmpl $0,-4(%ebp)    // comparer x avec 0
                je else0            // sauter s'il est égal
                call bar            // appeler bar sinon
                movl %eax,-8(%ebp)  // valeur renvoyée par bar
        else0:                      // fin du test.
```

En ce qui concerne les boucles, notre langage ne connaît que les boucles `while`, traitées par la règle des lignes 94 à 102. *Attention,* il ne s'agit que d'une seule règle avec des actions au milieu. Si on retirait les actions, la règle serait :

```
intr : YWHILE '(' expr ')' instr
```

avec le mot clef `while`, le test entre parenthèses puis l'instruction (peut-être composée) qui forme le corps de la boucle.

```
1           | YWHILE '('
2                  { printf("debut%d:\n", $<i>$ = label()); }
3             expr ')'
4                  { printf("\tcmpl $0,%d(%%ebp)\n", $4->position);
5                    printf("\tje fin%d\n", $<i>3); }
6             instr
7                  { printf("\tjmp debut%d\n", $<i>3);
8                    printf("fin%d:\n", $<i>3);
9                  }
```

Avant l'assembleur qui calcule la valeur du test, on place une étiquette de début de boucle (ligne 95). La construction du nœud `expr` (ligne 96) s'accompagne de la production de l'assembleur pour calculer la valeur du test. On ajoute l'assembleur pour tester cette valeur et sauter sur la fin si elle est égale à 0 (lignes 97–98). La construction du nœud `instr` pour le corps de la boucle (ligne 99)

s'accompagne de la production de l'assembleur pour l'exécuter ; à la fin, il ne reste plus qu'à ajouter l'assembleur pour sauter au début (ligne 100) et à placer l'étiquette de fin (ligne 101).

Ainsi le code de la fonction (stupide)

```
foo(){
  int x;

  while(x)
    bar();
}
```

sera traduit par

```
        debut0:              // debut de la boucle:
                cmpl $0,-4(%ebp)   // comparer x et 0
                je fin0            // sauter en dehors si x == 0
                call bar           // appeler bar sinon
                movl %eax,-8(%ebp) // recupérer la valeur renvoyée par bar
                jmp debut0         // recommencer
        fin0:
```

Finalement, il reste l'instruction `return`, traitée par la règle de la ligne 103. Il suffit de fabriquer l'assembleur pour placer la valeur de l'expression retournée dans le registre `%eax` (ligne 104) et de sauter sur l'épilogue de la fonction, qui porte un nom convenu.

9.3.5 Prologues et épilogues de fonctions

La définition des fonctions se fait avec une seule règle entrelardée d'actions, entre les lignes 44 et 66. Si on retire les actions, la règle devient :

```
fonction : YNOM '(' .listarg ')' '{' listvar listinstr '}'
```

Une fonction dans notre langage se compose du nom de la fonction, la liste des arguments (sans types ; ils sont tous entiers) entre parenthèses, puis le corps de la fonction avec une déclaration des variables puis une liste éventuelle d'instructions.

La liste des arguments et celle des variables sont tous les deux réduits en `.listnom` (une liste facultative de nom) définie lignes 76–80 : pour chaque nom rencontré, on appelle la fonction `fairexpr` qui initialise dans la mémoire du compilateur une structure `expr` pour indiquer là où se trouvera sa valeur. Avec les variables globales `posexpr` (comme *position de l'expression*) et `incr` (comme *incrément*), `fairexpr` placera correctement les arguments dans la pile, à partir de 8 octets au dessus du pointeur de frame, en commençant par le premier grâce

à la ligne 46. Les variables automatiques quant à elles seront placées à partir de 4 octets en dessous du pointeur de frame, en descendant.

Le résultat est que quand le compilateur aura traité :

```
foo(a, b, c){
  int x, y, z;
```

le tableau `expr` contiendra :

index	nom	position
0	a	8
1	b	12
2	c	16
3	x	-4
4	y	-8
5	z	-12

À ce point, il n'y a pas encore une seule ligne d'assembleur générée. Le compilateur fabrique maintenant les directives (ligne 50) qui précèdent la fonction. On souhaiterait avoir le prologue de la fonction, mais on ne peut pas le fabriquer parce qu'on ignore encore la quantité de mémoire qui sera nécessaire pour les expressions temporaires. En conséquence, le compilateur fabrique une étiquette qui marque le début du corps de la fonction, sur laquelle on pourra sauter quand on fabriquera le prologue (ligne 51).

Ensuite, lors de la construction du noeud `.listinstr` (ligne 54), le compilateur fabriquera l'assembleur qui traduit le corps de la fonction. Après l'accolade fermante, il placera l'étiquette qui marque l'épilogue (ligne 56, pour qu'on y saute lors d'un `return` éventuel), puis le code usuel de l'épilogue des fonctions (ligne 57) : libération de la pile, dépilage de l'ancien frame pointer, retour de d'appel.

Finalement, on peut maintenant construire le prologue puisqu'en traduisant le corps de la fonction, le compilateur a mesuré le nombre de temporaires nécessaires (ce nombre se traduit par la valeur de la variable globale `posexpr`). La ligne 59 produit donc l'étiquette qui marque le début de la fonction, La ligne 60 les deux lignes usuelles d'assembleur qui commencent le prologue (empiler l'ancien frame pointer, faire pointer le nouveau frame pointer sur la sauvegarde). La ligne 61 produit à son tour la ligne d'assembleur qui réserve l'espace sur la pile (en utilisant la valeur de `posexpr`) ; le compilateur émet finalement (ligne 62) l'assembleur pour sauter sur le début du corps de la fonction.

Ainsi, la compilation de la fonction (vide)

```
main(){
}
```

produit le code suivant :

```
  `             .text             // directives
```

```
        .align 16
         .globl main
debmain:                // debut (et fin) du corps de la fonction
finmain:                // EPILOGUE :
        movl %ebp,%esp  //   vider la pile
        popl %ebp       //   récupérer le vieux frame pointer
        ret             //   et revenir
main:                   // PROLOGUE
        pushl %ebp      //   sauver le vieux frame pointer
        movl %esp,%ebp  //   placer le nouveau frame pointer
        subl $0,%esp    //   réserver 0 octets sur la pile
        jmp debmain     //   et attaquer le corps de la fonction
```

9.3.6 Améliorations de ppcm

On trouvera dans les documents associés au cours des améliorations ponctuelles apportées à ppcm, que je présente ici.

Je n'ai pas intégré ces améliorations dans le programme principal parce que je pense qu'elles sont plus faciles à étudier quand il n'y en a qu'une seule d'ajoutée à la version simple du compilateur.

Pour les étudier, il est utile de savoir utiliser la commande `diff` (voir le manuel).

Opérateur de comparaison ==

Dans le répertoire `cmp`, il y a une modification (simple) pour rajouter l'opérateur d'égalité (==). La modification est élémentaire.

Traitement des constantes

Le répertoire `cste` contient une modification qui permet de ne pas placer immédiatement les constantes dans un mot mémoire. Ici, la valeur est placée dans la structure `expr` en attendant qu'elle soit utilisée ; quand c'est le cas, elle est placée directement dans l'assembleur (avec le mode d'adressage immédiat), sans utiliser de mémoire dans la pile.

Il est facile (mais pas fait) d'y rajouter le calcul des opérations dont les opérandes sont tous les deux des constantes.

Une autre forme de boucle

Pour montrer comment on peut réaliser d'autres formes de boucles, le répertoire `dirtyfor` contient l'ajout des boucles `for` du C, à grand coup de sauts inconditionnels dans le code généré.

Les connecteurs logiques

Dans le répertoire `andor`, on trouve la compilation des opérateurs `&&` et `||` de C.

Ces opérateurs sont un peu spéciaux parce que (comme dans la plupart des langages de programmation), ces expressions ne sont évaluées qu'autant que nécessaire pour déterminer la valeur de l'expression. Dans l'expression e_1 `&&` e_2, l'expression e_2 ne sera calculée que si e_1 est vraie (si e_1 est fausse, on sait déjà que l'expression est fausse et il n'y a pas besoin de calculer e_2). De même, dans e_1 `||` e_2, l'expression e_2 ne sera calculée que si e_1 est fausse (si e_1 est vraie, on sait déjà que l'expression est vraie et il n'y a pas besoin de calculer e_2).

Il s'agit d'une forme élémentaire de structure de contrôle : e_1 `&&` e_2 équivaut à `if (`e_1`)` e_2`;` et e_1 `||` e_2 équivaut à `if (!` e_1 `)` e_2 `;`.

Cette subtilité est nécessaire pour pouvoir écrire des choses comme :

```
if (npersonnes > 0 && ngateaux / npersonnes == 0)
  printf("Il n'y en aura pas pour tout le monde\n");
```

Si `npersonnes` vaut 0, il ne faut en aucun cas effectuer la division ! On rencontre aussi fréquemment cette organisation de test avec les tableaux :

```
int tab[MAX], i;
...
if (i < MAX && tab[i] != 0)
  ...
```

Si `i` est supérieur ou égal à `MAX`, il ne faut pas tester la valeur de `tab[i]`, sinon on risquerait d'accéder à un mot mémoire interdit et le processus s'arrêterait avec une erreur d'adressage.

Réutilisation des temporaires

Dans la version simple présentée ici, `ppcm` attribue un mot mémoire pour chacune des expressions temporaires (et des constantes) qu'il manipule. Un rapide examen du code montre qu'en réalité chaque temporaire ou constante n'est utilisé qu'une seule fois. Il est donc tentant de réutiliser ces mots mémoire.

Le répertoire `free` contient cette modification : la structure `expr` contient un nouveau champs qui sert de marqueur pour indiquer s'il s'agit d'un temporaire utilisé ou pas.

Chaque fois qu'on utilise un temporaire, on bascule ce marqueur à *inutilisé* (puisqu'on ne l'utilise qu'une fois).

Quand on a besoin d'un nouveau temporaire, la fonction `fairexpr` cherche un temporaire inutilisé et le réutilise plutôt que d'en allouer un nouveau.

Tableaux et pointeurs

Le répertoire `point` contient l'addition à `ppcm` des opérateurs d'indirection via des pointeurs et de l'opérateur `[]` pour accéder aux éléments d'un tableau.

Il a fallu que j'ajoute dans chaque noeud de l'arbre syntaxique qui pointe sur une structure `expr` un indicateur du nombre d'indirections nécessaires pour atteindre effectivement la valeur. En cas d'indirection on ne peut plus maintenant se contenter d'utiliser directement la valeur dans la pile. Il faut charger son adresse (ce qui est fait dans le registre `%ebx`) pour récupérer ou modifier la valeur de l'expression.

9.4 Exercices

Pour les exercices dont le corrigé est fourni avec le cours (les extensions de ppcm), il ne faut pas refaire le travail, mais simplement commenter les différences avec la version originale.

Ex. 9.1 — Quel arbre syntaxique le parseur de ppcm construira-t-il en compilant la fonction

```
main(){
  int c;

  while((c = getchar()) != -1)
    putchar(c);
}
```

Ex. 9.2 — Ajouter les opérateurs de comparaison >, >=, <, <= à ppcm.

Ex. 9.3 — Modifier ppcm pour qu'il effectue les opérations dont tous les opérandes sont constants au moment de la compilation.

Ex. 9.4 — Comparer le code produit par la compilation du code suivant avec la version simple et la version free de ppcm. Pour chacun d'entre eux, indiquer le rôle (ou les rôles) pour chaque temporaire allouée par ppcm.

```
fib_iter(n){
  int a, b, c;

  a = 0;
  b = 1;
  c = 0;
  while(c != n){
    c = c + 1;
    a = a + b;
    b = a - b;
  }
```

```
  return a;
}
```

Chapitre 10

Optimisation

Ce chapitre présente les optimisations qu'on peut attendre d'un compilateur. C'est important de savoir de quoi le compilateur est capable, parce que cela peut nous conduire à écrire le code d'une manière différente pour tirer profit de ses capacités ou pallier à ses déficiences.

10.1 Préliminaires

Attention, ce chapitre présente le code optimisé du C vers le C, mais il ne faut *jamais* écrire vos programmes directement sous la forme de code optimisé : c'est bien plus important d'avoir un programme juste qu'un programme rapide et un programme optimisé est très difficilement débuggable. (Si ce n'est pas votre opinion, placez la ligne

```
# define while if
```

au début de vos fichiers C : cela risque d'accélérer significativement vos programmes ; vous pouvez aussi économiser de la mémoire avec `# define struct union`.)

10.1.1 Pourquoi optimise-t-on ?

C'est souvent une bonne idée de générer du code améliorable puis de l'améliorer dans une seconde étape, plutôt que de générer rapidement du code optimal.

Séparer permet de simplifier.

On peut utiliser la compilation lente pour avoir du code rapide (pour la production) ou bien une compilation rapide pour du code lent (pour la mise au point).

10.1.2 Quels critères d'optimisation ?

Le terme *optimisation* est largement galvaudé en informatique. Au sens propre, *optimiser quelque chose* signifie qu'on trouver *la meilleure forme* pour cette chose. Dans notre discipline, on l'emploie fréquemment avec le sens bien plus restreint d'amélioration.

L'amélioration peut concerner de nombreux aspects : pour les compilateurs actuels, il s'agit le plus souvent de minimiser le temps d'exécution et parfois la quantité de mémoire utilisée (pour les systèmes embarqués). Ces deux types d'optimisation sont souvent contradictoires.

On distingue aussi l'optimisation pour le pire cas de l'optimisation pour le cas moyen. La première permet de limiter le temps maximum, alors que la seconde s'intéresse au cas le plus fréquent.

10.1.3 Sur quelle matière travaille-t-on ?

Souvent du pseudo-code pour pouvoir réutiliser.

Code deux adresses (genre assembleur Intel)

Code trois adresses (genre assembleur RISC)

Pour mémoire, il y a du code 1 adresse (tous les calculs se font avec le même registre, souvent le sommet de pile)

Gcc travaille sur le RTL (Register Transfer Language). Le RTL est une sorte d'assembleur pour une machine virtuelle avec un nombre infini de registres et une syntaxe à la Lisp. Gcc place tout dans des registres, sauf ce dont il doit pouvoir calculer l'adresse (les tableaux). Ne pas essayer de manipuler l'arbre syntaxique fabriqué par l'analyseur ; en revanche on peut manipuler le RTL.

10.1.4 Comment optimise-t-on ?

90 % du temps se passe (usuellement) dans 10 % du code (le cœur des boucles) ; c'est là qu'il faut travailler pour obtenir les résultats les meilleurs.

10.2 Définitions

Un optimiseur à la lucarne ne considère que quelques instructions à la fois ; il n'examine que des instructions voisines et donc ne peut pas collecter des informations globales sur le fonctionnement du code. En anglais, on appelle cela un *peep-hole optimizer* (un optimiseur à travers le trou de la serrure en français). Un optimiseur de ce type est relativement facile à fabriquer.

On appelle bloc de base une suite d'instructions dont on a la garantie qu'elles seront toutes exécutées en séquence (il ne contient pas d'instruction de saut, sauf à la fin ; il ne ne contient pas d'étiquette sur laquelle sauter). Les optimiseurs

peuvent donc utiliser les informations glanées sur l'effet des premières instructions du bloc dans les instructions suivantes.

On considère que les expressions (ou les registres) ont une *durée de vie* : elle commence quand l'expression est calculée (on dit alors qu'elle est *définie*; sa mort intervient à la suite de la dernière utilisation de l'expression.

10.3 Optimisations indépendantes

Cette section examine les techniques d'optimisation les plus courantes une par une. Dans l'optimiseur, il y aura interactions entre ces techniques, considérées dans la section suivante.

10.3.1 Le pliage des constantes

Quand on a des valeurs constantes dans le programme, le compilateur peut effectuer les calculs directement sur ces valeurs au moment de la compilation et remplacer les expressions qui les contiennent par le résultat du calcul. Par exemple, les trois instructions assembleurs :

```
movl $1,%eax
sall $10,%eax
movl %eax,-16(%ebp)
```

peuvent être remplacées par

```
movl $1024,-16(%ebp)
```

A peu près tous les compilateurs font cette optimisation ; ça nous permet d'écrire du code plus lisible (si on peut faire confiance au compilateur). Par exemple le code précédent est obtenu à partir de

```
# define bit(n) (1 << (n))
    ...
  x = bit(10);
```

De même, pour définir 5 mégas, je préfère `5 * 1024 * 1024` à `5242880`.

Un autre exemple fréquent d'apparition des constantes est de la forme :

```
if (foo < &tableau_global[MAX]) ...
```

Puisque le tableau est global, il est à une adresse fixe. `MAX` est constant, donc l'adresse de `&tableau_global[MAX]` est une constante qui peut être calculée avant l'exécution. Cette optimisation est moins fréquente, parce qu'il faut que le compilateur et l'éditeur de lien coopèrent : le compilateur ne sait pas où dans la mémoire l'éditeur de lien placera le tableau.

Tous les compilateurs ne savent pas exploiter les propriétés des opérations arithmétiques pour regrouper les constantes ; par exemple reconnaître que `1+x+1` est équivalent à `x+2`.

Simplifications algébriques

Certaines opérations avec des constantes donnent des résultats connus : addition ou soustraction de 0, multiplication par 0 ou par 1, division par 1.

On ne peut pas appliquer systématiquement les simplifications algébriques ; il faut s'assurer que la suppression d'une expression ne modifie pas le programme. Par exemple dans l'expression `0*printf("foo\n")` on connaît sa valeur (c'est 0) mais il ne faut pas supprimer l'appel de printf !

Propagation des constantes

Quand une instruction place une constante dans une variable, il y a moyen de continuer à utiliser sa valeur tant que la variable n'est pas redéfinie. Ceci n'est pas tout à fait pareil que les opérations sur les constantes (du point de vue du compilateur). Par exemple `x = 23; y = 2 * x;` s'optimise facilement en `x = 23; y = 46;`.

De même, on peut imaginer une boucle formulée comme :

```
for(p = q = &tableau_global[base]; p < q + MAX; p++)
  *p = 0;
```

(écrite, assez maladroitement, dans l'idée d'éviter le calcul de `&tableau_global[base]` à chaque tour). La propagation des constantes permettra de remplacer `q+MAX` par la valeur.

10.3.2 Les instructions de sauts

On peut générer des instruction de saut sans se poser de question, puis laisser l'optimisation retirer les sauts inutiles.

Saut sur la prochaine instruction

Un instruction de saut sur l'instruction suivante est bien évidemment inutile et peut être retirée.

Un exemple où le générateur de code produit un saut sur l'instruction suivante : pour compiler l'instruction `return x;` il va produire des instructions pour placer la valeur renvoyée à l'endroit convenu (dans le registre `%eax` puis sauter sur l'épilogue de la fonction :

```
movl x,%eax
jump epilogue
```

Si le return est la dernière instruction du corps de la fonction, alors le saut sur l'épilogue est en fait un saut sur la prochaine instruction.

les sauts par dessus des sauts

Le générateur de code peut produire des sauts conditionnels qui sautent seulement par dessus une instruction de saut ; par exemple `if (test()) ; else bar(); }` sera traduit par ppcm en

```
(1)         call test
(2)         movl %eax,-4(%ebp)
(3)         cmpl $0,-4(%ebp)
(4)         je else0
(5)         jmp fin0
(6) else0:
(7)         call bar
(8)         movl %eax,-8(%ebp)
(9) fin0:
```

Il est bien sur souhaitable de remplacer le lignes 4–6 par une seule instruction de saut, avec le test inversé :

```
(5)         jne fin0
```

Les sauts sur des sauts

On peut couramment rencontrer des sauts sur d'autres instructions de saut, par exemple avec des boucles imbriquées. Ppcm traduira le corps de la fonction :

```
foo(){
  while(test1()){
    corps1();
    while(test2())
      corps2();
  }
}
```

par l'assembleur :

```
 (1) debut0:                     // premier while
 (2)         call test1          // test1()
 (3)         movl %eax,-4(%ebp)
 (4)         cmpl $0,-4(%ebp)    //  test1() == 0
 (5)         je fin0
 (6)         call corps1         // corps1()
 (7)         movl %eax,-8(%ebp)
```

```
 (8) debut1:                     // deuxième while
 (9)         call test2          // test2()
(10)         movl %eax,-12(%ebp)
(11)         cmpl $0,-12(%ebp)   // test2() == 0
(12)         je fin1
(13)         call corps2         // corps2()
(14)         movl %eax,-16(%ebp)
(15)         jmp debut1
(16) fin1:
(17)         jmp debut0
(18) fin0:
```

Le saut de la ligne 12 peut bien sur être remplacé par `jump debut0`.

Il est aussi possible (mais plus complexe) de détecter qu'un saut conditionnel renvoie sur un autre saut conditionnel équivalent, comme dans dans l'exemple :

```
if (x >= 10)
  if (x > 0)
    ...
```

Si l'exemple semble artificiel, considérer le fragment de code suivant

```
# define abs(n) ((n) < 0 ? -(n) : (n))
...
  if (x > Min && y > Min)
    dist = abs(x) + abs(y); // distance Manhattan
  else
    dist = sqrt(x*x + y*y); // distance pythagoricienne
```

10.3.3 Ôter le code qui ne sert pas

On appelle le code inutilisé du code *mort*. Le compilateur peut (dans certains cas) le détecter à la compilation. Pour un exemple trivial :

```
foo(){
  bar();
  return 0;
  joe();
}
```

L'appel de `joe();` suit le `return` et ne sera donc jamais effectué.

La détection du code du code mort est facile dans le cas ordinaire : c'est un bloc de base qu'on ne peut pas atteindre. En revanche pour détecter qu'une boucle ne peut pas être atteinte, il faut construire le graphe de tous les enchaînements de blocs de base et en trouver les parties disjointes (ce qui est beaucoup plus lourd).

10.3.4 Utilisation des registres

La qualité du code généré par un compilateur dépend pour une grande partie de l'allocation des registres. Il y a toutes sortes de méthodes heuristiques qui permettent d'effectuer cette affectation d'une façon presque optimale *si le code correspond au comportement par défaut* (en général : qu'un test est plus souvent faux que vrai, que les boucles sont exécutées un certain nombre de fois). Sur ce point, je renvoie le lecteur au *Dragon Book* sans détailler.

Il est possible dans le langage C d'indiquer au compilateur les variables les plus utilisées avec le mot clef `register`, dans l'idée d'aider le compilateur à placer les bonnes variables dans des registres. Le compilateur gcc ignore purement et simplement ces indications, sans doute parce que sont ses auteurs sont certains que leurs choix d'affectation des registres sont meilleurs que ceux de l'auteur du programme. Cela me semble très criticable comme point de vue ; j'ai cependant cessé d'utiliser `register` dans mes propres programmes.

Une optimisation bien plus accessible sur les registres est d'éliminer les chargements et déchargements inutiles, qui sont nombreux dans le code produit par un générateur de code naïf. Par exemple, ppcm traduira le corps de :

```
foo(a, b, c){
  a + b + c;
}
```

par

```
(1)        movl 8(%ebp),%eax     // a dans %eax
(2)        addl    12(%ebp),%eax // a+b dans %eax
(3)        movl %eax,-4(%ebp)    // sauver a+b

(4)        movl -4(%ebp),%eax    // a+b dans %eax
(5)        addl    16(%ebp),%eax // a+b+c dans %eax
(6)        movl %eax,-8(%ebp)    // sauver a+b+c
```

Un optimiseur à la lucarne détectera aisément que le chargement de la ligne 4 est inutile. C'est un peu plus difficile de constater que la valeur de `-4%ebp` n'est utilisé nulle part et que le déchargement de la ligne 3 est lui aussi inutile.

Placer les adresses mémoires dans des registres avant d'opérer dessus peut aussi être une source d'optimisation.

10.3.5 Inliner des fonctions

Les optimiseurs peuvent couramment choisir de remplacer un appel de fonction par le corps de la fonction lui-même. Cela présente deux avantages : d'une part on se dispense du coût de l'appel fonction, d'autre part cela ouvre la voie à de nouvelles optimisations (par exemple des propagations de constantes).

En C, on peut indiquer lors de la définition d'une fonction que ses appels doivent être remplacés par sa définition avec le mot clef `inline`. Par exemple, on peut faire à peu près indifféremment :

```
# define ABS(n) (n < 0 ? -n : n)
static int
abs(int n){
  if (n < 0)
    return -n;
  return n;
}
```

(En fait la définition de `ABS` manque de parenthèses et l'appel de `ABS(-x)` aura des effets surprenants à première vue. A la différence de la macro, la fonction `abs` ne fonctionne pas avec des nombres flottants.)

Attention, l'abus des fonctions *inline* fait grossir le code, ce qui dégrade l'efficacité du cache et peut conduire facilement qu'on ne le pense à des pertes de performances.

10.3.6 Les sous expressions communes

On a fréquemment des sous-expressions qui apparaissent plusieurs fois. Le compilateur peut dans ce cas réutiliser la valeur calculée lors de la première utilisation de l'expression. Pour un exemple élémentaire :

```
  x = a + b + c;
  y = a + b + d;
```

l'expression `a+b` apparaît deux fois et l'optimiseur doit la reformuler de manière a ne faire que trois additions, et non quatre comme dans la version originale :

```
  t = a + b;
  x = t + c;
  y = t + d;
```

Quand on programme dans un style ordinaire, les sous-expressions communes apparaissent tout particulièrement dans les manipulations de tableaux. Étant donné un tableau déclaré avec

```
  char tab[X][Y][Z];
```

quand on fait référence à un de ses éléments `tab[a][b][c]`, le compilateur va calculer son adresse avec une série de multiplications et d'addition `tab + a * Y * Z + b * Z + c`. Si on fait référence aux cases de deux tableaux, comme dans :

```
(1)         char from[X][Y][Z], to[X][Y][Z];
(2)         ...
(3)         to[i][j][k] = from[i][j][k];
```

la séquence d'opérations pour calculer les adresse des éléments de `from` et de `to` est la même. Un bon optimiseur ré-écrira la ligne 3 dans l'équivalent de

```
(3.1)  tmp = i * Y * Z + j * Z + k;
(3.2)  *(to + tmp) = *(from + tmp);
```

ou comme :

```
(3.1)  tmp = (i * Y + j) * Z + k;
(3.2)  *(to + tmp) = *(from + tmp);
```

10.3.7 La réduction de force

Il s'agit de remplacer des opérations coûteuses par d'autres équivalentes. En anglais, on appelle cela de la *strength reduction*

On peut remplacer une multiplication par une puissance de 2 par un décalage ou la division d'un nombre flottant par une constante par une multiplication par l'inverse de la constante. Il faut prendre garde dans ce dernier cas au problème de précision des calculs que ces manipulations peuvent affecter.

Une opération ordinaire que nous avons utilisée dans la programmation en assembleur consiste à remplacer une comparaison avec 0 `cmpl $0,%eax` par le test équivalent `andl %eax,%eax` et la mise à 0 d'un registre comme `movl $0,%eax` par `xorl %eax,%eax`.

10.3.8 Sortir des opérations des boucles

Quand l'optimiseur peut déterminer qu'une expression calculée dans une boucle ne change pas de valeur entre deux tours, il peut sortir le calcul de la boucle pour l'exécuter une seule fois avant d'y entrer. Par exemple dans :

```
while(i < nelements - 1)
  t[i++] = 0;
```

si le compilateur peut déterminer que `nelements` ne change pas de valeur dans la boucle, alors il peut sortir le calcul de nelements - 1 de la boucle, pour donner l'équivalent de

```
tmp1 = nelements - 1;
while( i < tmp1)
  t[i++] = 0;"
```

10.3.9 Réduction de force dans les boucles

Souvent, il est possible de remplacer dans une boucle le calcul lent d'une expression par une modification rapide de la valeur de l'expression au tour précédent. Ainsi, il est intéressant que le compilateur puisse récrire :

```
int tab[Max];

for(i = 0; i < Max; i++)
  tab[i] = i * 5;
```

comme

```
for(i = tmp = 0; i < Max; i++, tmp += 5)
  tab[i] = tmp;
```

ce qui permet de remplacer une multiplication par une addition, moins coûteuse. On peut noter que la multiplication par 4 qu'implique et l'addition qu'implique le calcul de l'adresse de `tab[i]` est susceptible de recevoir le même traitement.

10.3.10 Dérouler les boucles

Dans certains cas, il peut être intéressant que le compilateur déroule les boucles, en remplaçant la boucle par la répétition de son corps. Par exemple il peut remplacer avantageusement

```
for(i = 0; i < 4; i++)
  to[i] = from[i];
```

par

```
i = 0;
to[i] = from[i];
i++;
to[i] = from[i];
i++;
to[i] = from[i];
i++;
to[i] = from[i];
i++;
```

Ceci évite les tests et les instructions de saut, mais surtout permet, après propagation des constantes, d'obtenir :

```
to[0] = from[0];
to[1] = from[1];
to[2] = from[2];
to[3] = from[3];
i = 4;
```

Il est aussi possible de dérouler une boucle même quand on ne peut pas déterminer le nombre de fois qu'elle tournera. Cela revient par exemple à remplacer

```
for(i = 0; i < n; i++)
  to[i] = from[i];
```

par (en déroulant deux fois) :

```
for(i = 0; i < n - 1; i++){
  to[i] = from[i];
  i++;
  to[i] = from[i];
}
if (i < n){
  to[i] = from[i];
  i++;
}
```

Cela permet (au moins) d'éviter une instruction de saut et un test pour chaque déroulement et ouvre la voie à de nouvelles propagations de constantes et élimination de sous-expressions communes.

Attention, l'abus du déroulement des boucles fait grossir le code, dégrade l'exploitation des caches et peut facilement induire des pertes de performances.

10.3.11 Modifier l'ordre des calculs

Sur les processeurs actuels, il existe plusieurs unités de calcul indépendantes, susceptibles d'effectuer chacune un calcul en même temps (on appelle cela un processeur *multithreadé*. Le processeur charge plusieurs instructions en même temps (souvent quatre) et commence leur exécution à toutes les quatre. Si par exemple un calcul sur l'UAL 2 dépend du résultat du calcul sur l'UAL 1, le travail est interrompu sur l'UAL 2 jusqu'à ce qu'il soit disponible. On a des problèmes du même genre avec les unités de calcul en *pipe-line*.

Un bon optimiseur tiendra compte de la structure des unités de calcul de manière à organiser les instructions indépendantes pour qu'elles soient exécutées en même temps.

La réorganisation des instructions indépendantes permet aussi de diminuer l'utilisation des registres. Une bonne heuristique est de commencer par calculer la partie la plus simple de l'expression.

10.3.12 Divers

Gcc tente d'optimiser l'utilisation des caches pour la mémoire en alignant le code et les données dans la pile sur ce qu'il suppose être des débuts des blocs de mémoire stockés dans les caches (les *lignes de cache*).

Gcc permet de retarder le dépilement des arguments des fonctions empilées avant un appel, et de ne pas utiliser de frame pointer.

Il ne garde qu'une seule fois les constantes qui apparaissent à plusieurs endroits différents dans le programme (notamment pour les chaînes de caractères).

Il permet au programmeur d'indiquer la valeur probable d'un test, de manière à améliorer la localité du code et à insérer des instructions de pré-chargement de la mémoire sur les processeurs qui le permettent. On utilise souvent ceci avec les macros `likely` et `unlikely` :

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

La macro `likely` indique que l'expression `x` aura le plus souvent la valeur 1 (vrai) et `unlikely` qu'elle aura probablement la valeur 0 (faux).

10.3.13 Les problèmes de précision

Il est bien sur hors de question pour l'optimiseur de produire du code rapide mais faux, ou que l'accélération des calculs conduise à des pertes de précision dans les calculs. Certains programmes fonctionnent aussi moins bien quand les calculs sont *trop* précis. Dans ce cas il n'est pas possible de conserver dans un registre de précision `double` une expression dont la précision n'est que `float`.

10.4 Tout ensemble : deux exemples

Considérons du code de mise au point conservé (mais inactivé) dans les sources du programme, sous la forme d'une macro `DEBUG` :

```
# define FAUX (0 == 1)
# define DEBUG if(FAUX)printf
  ...
  DEBUG("La variable i vaut %d\n", i);
  ...
```

Les étapes d'optimisation qui nous permettent de conserver ce code dans nos programmes sans avoir aucun coût supplémentaire sont les suivantes.

- Le compilateur identifie que l'expression `0 == 1` est constante et la remplace par sa valeur (`0`).
- Il reconnaît que le test est toujours faux et remplace son saut conditionnel par un saut sans condition.
- Il détecte que l'appel à `printf` est du code mort et le retire (le code n'est mort qu'après la transformation de saut de l'étape précédente).
- Finalement, puisque la chaîne de caractère n'est utilisée nulle part, elle est éliminée.

Considérons la fonction :

```
void
copier(int to[], int from[]){
  int i;

  for(i = 0; i < N; i++)
    to[i] = from[i];
}
```

Le corps est traduit par un générateur de code naïf en quelque chose comme :

```
    movl $0,%eax       // i = 0
loop:
    cmpl $N,%eax       // i < N
    jge  fin
    movl %eax,%ebx     // &from[i] dans %ebx
    sall $2,%ebx
    addl from,%ebx
    movl %eax,%ecx     // &to[i] dans %ecx
    sall $2,%ecx
    addl to,%ecx
    movl (%ebx),(%ecx) // to[i[] = from[i]
    incl %eax
    jump loop
fin:
```

On peut sortir une instruction de la boucle en réorganisant le code :

```
    movl $0,%eax       // i = 0
    jump in
loop:
    movl %eax,%ebx     // &from[i] dans %ebx
    sall $2,%ebx
    addl from,%ebx
    movl %eax,%ecx     // &to[i] dans %ecx
    sall $2,%ecx
    addl to,%ecx
    movl (%ebx),(%ecx) // to[i[] = from[i]
    incl %eax
in:
    cmpl $N,%eax       // i < N
    jlt  loop
fin:
```

Après élimination de la sous-expression commune :

```
    xorl %eax,%eax     // i = 0
    jump in
loop:
    movl %eax,%ebx
    sall $2,%ebx
    movl %ebx,%ecx
    addl from,%ebx     // &from[i] dans %ebx
    addl to,%ecx       // &to[i] dans %ecx
    movl (%ebx),(%ecx) // to[i[] = from[i]
    incl %eax          // i++
in:
    cmpl $N,%eax       // i < N
    jlt  loop
fin:
```

Calcul incrémental de $4i$:

```
    xorl %eax,%eax     // i = 0
    xorl %edx,%edx     // 4i = 0
    jump in
loop:
    movl %edx,%ebx
    movl %ebx,%ecx
    addl from,%ebx     // &from[i] dans %ebx
    addl to,%ecx       // &to[i] dans %ecx
    movl (%ebx),(%ecx) // to[i[] = from[i]
    incl %eax          // i++
    addl $4,%edx       // 4i += 4
in:
    cmpl $4*N,%edx     // 4i < 4N
    jlt  loop
fin:
```

Suppression de la variable i qui ne sert plus à rien :

```
    xorl %edx,%edx     // 4i = 0
    jump in
loop:
    movl %edx,%ebx
    movl %ebx,%ecx
    addl from,%ebx     // &from[i] dans %ebx
    addl to,%ecx       // &to[i] dans %ecx
    movl (%ebx),(%ecx) // to[i[] = from[i]
    addl $4,%edx       // 4i += 4
in:
    cmpl $4*N,%edx     // 4i < 4N
    jlt  loop
```

```
fin:
```

Utilisation du mode d'adressage indirect indexé

```
    xorl %edx,%edx     // 4i = 0
    movl from,%ebx
    movl to,%edx
    jump in
loop:
    movl (%edx,%ebx),(%edx,%ecx) // to[i[] = from[i]
    addl $4,%edx       // 4i += 4
in:
    cmpl $4*N,%edx     // 4i < 4N
    jlt  loop
fin:
```

Attribution de registres aux constantes

```
    xorl %edx,%edx     // 4i = 0
    movl from,%ebx
    movl to,%edx
    movl $4*N,%eax     // %eax = 4*N
    movl $4,%esi       // %esi = 4
    jump in
loop:
    movl (%edx,%ebx),(%edx,%ecx) // to[i[] = from[i]
    addl %esi,%edx     // 4i += 4
in:
    cmpl %eax,%edx     // 4i < 4N
    jlt  loop
fin:
```

Cette version est plus ou moins l'équivalent du code C

```
  for(p = &to, q = from, r = &to[N]; p < r)
    *p++ = *to++;
```

Je répète encore une fois qu'il ne faut pas programmer de cette façon ; mieux vaut garder un programme lisible et laisser travailler l'optimiseur, ou se contenter d'un programme un peu pluslent qu'on peut mettre relire.

10.4.1 Le problème des modifications

Tous les appels de fonctions et les références à travers les pointeurs ont pu modifier toutes les variables qui ne sont pas locales.

Le mot clef `const` permet de limiter les dégâts en signalant au compilateur ce qui n'est pas modifié. Par exemple :

```
int strcpy(char *, const char *);
```

pour indiquer que les caractères pointés par le premier argument sont modifiés, mais pas ceux pointés par le second. Attention, `const` est d'un maniement très délicat :

```
int foo(const char **);
```

indique que les caractères ne sont pas modifiés, mais le pointeur sur les caractères peut l'être, alors que

```
int foo(char const * *);
```

indique que le pointeur sur les caractères n'est pas modifiés mais que les caractères, eux, peuvent l'être. Si ni l'un ni l'autre ne le sont, il faut écrire :

```
int foo (const char const * *)
```

On peut même dire :

```
int foo (const char const * const *)
```

ce qui n'a pas grand sens puisque l'argument est une copie; on se moque de savoir si elle est modifiée ou pas.

10.5 Optimisations de gcc

Les options -O0, -O o -O1, -O2, -O3, -Os, -funroll-loops, -funroll-all-loops

Renvoyer à la page de documentation ou la reprendre.

10.6 Exercices

Ex. 10.1 — (facile) Dans l'assembleur suivant, produit par gcc en compilant la fonction pgcd, quels sont les blocs de base ?

```
pgcd:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        jmp     .L2
.L4:
        movl    8(%ebp), %eax
        cmpl    12(%ebp), %eax
        jge     .L3
        movl    12(%ebp), %eax
        movl    %eax, -4(%ebp)
```

```
        movl    8(%ebp), %eax
        movl    %eax, 12(%ebp)
        movl    -4(%ebp), %eax
        movl    %eax, 8(%ebp)
.L3:
        movl    12(%ebp), %eax
        subl    %eax, 8(%ebp)
.L2:
        cmpl    $0, 8(%ebp)
        jne     .L4
        movl    12(%ebp), %eax
        leave
        ret
```

Ex. 10.2 — (assez facile) Dans la fonction copier (section 10.4), fixer la constante N à 10000 et compiler avec les options `-O2 -funroll-all-loops`. Comparer avec le code obtenu avec la seule option `-O2` et commenter les différences. (un peu plus difficile) Quel gain en terme de nombre d'instructions exécutées l'optimisation permet elle d'obtenir ? (encore un peu plus difficile) Comparer les vitesses d'exécution des deux versions du code obtenu et expliquer.

Ex. 10.3 — Expliquer le code produit par gcc, avec et sans optimisation, pour traduire la fonction `foo`.

```
enum {
  N = 1000 * 1000,
};

void
foo(void){
  char from[N], to[N];
  int i;

  init(from);
  for(i = 0; i < N; i++)
    to[i] = from[i];
  manipule(from, to);
}
```

(pas de correction) Comparer la vitesse du code produit avec ce qui se passe quand on remplace la boucle `for` par un appel à la fonction `memcpy`.

Ex. 10.4 — En compilant le code suivant, à partir de quel niveau d'optimisation gcc détecte-t-il que la fonction `joe` ne sera jamais appelée. (Dans ce cas, il ne placera par d'appel de la fonction dans le code généré.)

```
void
```

```
bar(int x){
  if (x)
    ceci();
  else if (!x)
    cela();
  else
    joe();
}
```

Ex. 10.5 — Même question pour

```
void
bar(int x){
  if (x)
    ceci();
  else if (!x)
    cela();
  else
    for(;;)
      joe();
}
```

(pas encore de corrigé).

Ex. 10.6 — Compiler le code suivant en utilisant gcc avec et sans optimisations. Identifier les optimisations apportées.

```
int fib_iter(int n){
  int a = 0, b = 1, c = 0;
  while(c != n){
    c = c + 1;
    a = a + b;
    b = a - b;
  }
  return a;
```

Chapitre 11

L'analyse lexicale : le retour

Ce chapitre revient sur les analyseurs lexicaux traités dans un chapitre précédent. Il montre l'équivalence qui existe entre les automates finis et les expressions régulières, puis présente brièvement `lex`, un outil de prototypage rapide d'analyseurs lexicaux.

11.1 Les expressions régulières

Les expressions régulières sont une façon de décrire un ensemble de chaînes de caractères (le terme correct en français est *« expressions rationnelles »*, mais il est très peu utilisé). Elles apparaissent à de nombreux endroits dans le système Unix et la manière d'exploiter leurs capacités fait souvent la différence entre un utilisateur averti et un novice.

11.1.1 Les expressions régulières dans l'univers Unix

Usage courant : extraire avec `egrep` les lignes importantes d'un (gros) fichier. Par exemple on peut extraire toutes les lignes d'un fichier de logs qui contiennent le mot `important` avec :

```
egrep 'important' fichier
```

où bien trouver la valeur de la variable `HOME` dans l'environnement avec

```
printenv | egrep HOME=
```

On peut réaliser des recherches beaucoup plus sophistiquées ; par exemple extraire les lignes qui contiennent soit `foo`, soit `bar` avec `'foo|bar'` ou bien celles qui contiennent `foobar` ou `foo et bar` avec `'foo( et )?bar'`.

La commande `sed` permet de manipuler des lignes sur la base des expressions régulières ; on l'utilise fréquemment pour fabriquer des commandes. Par exemple

pour retirer le '.bak' à la fin des noms des fichiers du répertoire courant, on peut prendre la liste des fichiers donnée par la commande `ls`, utiliser sed pour fabriquer des lignes de commande et les passer à l'interprète de commande :

```
ls *.bak | sed 's/\(.*\).bak/mv "&" "\1"/' | sh
```

Ou bien pour remplacer le (premier) blanc souligné des noms de fichiers par un espace :

```
ls *_* | sed 's/\(.*\)_\(.*\)/mv "&" "\1 \2"/' | sh
```

La commande `ed` permet de modifier les fichiers sur place. Par exemple, pour remplacer tous les `badname` par `GoodName` dans tous les fichiers C du répertoire courant :

```
for i in *.[ch] do ed - "$i" << EOF ; done
g/\([^a-zA-Z0-9_]\)badname\([^a-zA-Z0-9_]\)/s//\1GoodName\2/g
g/^badname\([^a-zA-Z0-9_]\)/s//GoodName\1/g
g/\([^a-zA-Z0-9_]\)badname$/s//\1GoodName/g
g/^badname$/s//GoodName/g
wq
EOF
```

La structure des quatre premières commandes pour `ed` sont

```
g/expr1/s//expr2/g
```

qui indique, dans toutes les lignes où une chaîne correspond à l'expression régulière `expr1`, de remplacer toutes les chaines qui y correspondent par l'expression régulière 2. La première commande traite le cas général où `badname` apparaît au milieu de la ligne, la deuxième celle où elle apparaît au début de la ligne, la troisième celle où elle se trouve en fin de ligne et la quatrième celle où elle se trouve toute seule sur la ligne.

La commande `awk` permet de combiner ces traitements avec des calculs arithmétiques. Par exemple pour calculer une moyenne :

```
awk '/Result: / { sum += $2; nel += 1 }
     END { print "moyenne", sum / nel }'
```

Pour calculer la moyenne *et* l'écart type :

```
awk '/Result: / { sum += res[NR] = $2; nel += 1 }
 END { moy = sum / nel;
       for (i in res)
         sumecarre += (res[i] - moy) * (res[i] - moy);
       print "moyenne", moy, "ecart type", sqrt(sumecarre / nel)
 }'
```

Les langages `perl` et `python` permettent un accès aisé aux expressions régulières. Un bon programmeur perl n'a pas besoin de connaître `awk` et `sed` ou les subtilités de la programmation shell : la maîtrise de perl permet de tout faire dans le langage.

En C, les expressions régulières permettent une analyse bien plus fine des entrées que le pauvre format de `scanf` avec ses mystères. Voir la documentation des fonctions `regcomp` et `regexec`.

11.1.2 Les expressions régulières de base

À la base, il n'y a que cinq constructeurs pour fabriquer les expressions régulières : le caractère, la concaténation, l'alternative, l'optionnel et la répétition. Les expressions régulières dans les commandes Unix comprennent de nombreuses autres facilités d'expressions, mais reposent presque exclusivement sur ces mécanismes de base.

Le caractère

La brique de base est une expression régulière qui désigne *un* caractère. Par exemple `egrep 'X'` extraira toutes les lignes qui contiennent un `X`.

La concaténation

On peut lier les briques entre elles : une expression régulière r_1 suivie d'une expression régulière r_2 donne une nouvelle expression régulière $r_1 r_2$ qui reconnaîtra les chaînes dont la première partie est reconnue par r_1 et la deuxième par r_2.

On utilise cette construction pour spécifier les expressions régulières qui contiennent plus d'un seul caractère. Par exemple, dans l'expression régulière `'bar'`, on a concaténé l'expression `'b'`, l'expression `'a'` et l'expression `'r'`.

On peut bien sur utiliser ce mécanisme pour concaténer des expressions régulières plus complexes.

L'alternative

L'alternative se note usuellement avec la barre verticale `|`, comme dans $r_1|r_2$: elle indique que l'expression régulière acceptera n'importe quelle chaîne acceptée soit par r_1, soit par r_2. Par exemple `'a|b|c'` reconnaîtra n'importe lequel des trois premiers caractères de l'alphabet, alors que `'pentium|((4|5)86)'` reconnaîtra `486`, `586` ou `pentium` (noter les parenthèses dans cet exemple, qui permettent d'éviter les ambigüités sur l'ordre dans lequel interpréter les concaténations et les alternatives).

Les expressions régulières d'Unix possèdent de nombreuses versions abrégées de cet opérateur, qui permettent d'écrire les expressions régulières d'une

façon plus aisée. La plus courante utilise les crochets quand les branches de l'alternative ne contiennent qu'un caractère : `'[aeiou]'` permet de spécifier *n'importe quelle voyelle* et est beaucoup plus lisible que `'(a|e|i|o|u)'`. De même, `'[0-9]'` indique *n'importe quel chiffre décimal* et `'[A-Za-z]'` *n'importe quel symbole alphabétique de l'ASCII.*

Parce que l'alphabet est *fini* (c'est à dire contient un nombre limité de caractères), on peut aussi utiliser une version *tout sauf ...* des crochets. Par exemple `'[^ABC]'` désigne *n'importe quel caractère sauf* `A`, `B` *et* `C`. De même le point `'.'` désigne *n'importe quel caractère* (sauf le retour à la ligne dans certains outils).

Certaines implémentations des expressions régulières ont également des constructions spécifiques pour désigner les différentes formes d'espace, les caractères qui peuvent appartenir à un mot, etc. Voir les pages de manuel des commandes pour les détails.

Il est important de comprendre que ces modes d'expression sont du *glaçage syntaxique* (en anglais *syntactic sugar*) : ils permettent d'exprimer de manière plus concise la même chose qu'avec les opérateurs de base, mais rien de plus.

L'option

L'opérateur d'option permet d'indiquer qu'on peut soit avoir une expression régulière, soit rien. Par exemple, `'foo( et )?bar'` indique *soit la chaîne* `'foo et bar'` (si l'expression optionnelle `' et '` est présente, *soit la chaîne* `'foobar'` (si elle est absente).

Certaines présentations (plus formelles) des expressions régulières utilisent un autre mécanisme équivalent : une expression régulière peut désigner la chaîne vide, notée ϵ. On écrirait alors l'expression régulière du dernier exemple `'foo(( et )|`ϵ`)bar'`. Un problème de cette présentation est qu'on ne trouve pas le caractère ϵ sur un clavier ordinaire.

La répétition

Il y a finalement un opérateur qui permet d'indiquer la répétition de l'expression régulière sur laquelle il s'applique, n'importe quel nombre de fois, y compris zéro. Il se note avec une étoile `'*'` et s'appelle la *fermeture de Kline*. Par exemple `'fo*bar'` reconnaîtra `fbar` (zéro fois `o`) et `fobar` (une fois `o`) et `foobar` (deux fois `o`) et `fooobar` (trois fois `o`) et ainsi de suite pour n'importe quel nombre de `o` entre le `f` et le `b`.

On combine fréquemment le point '.' qui désigne n'importe quel caractère avec l'étoile '*' pour indiquer *n'importe quoi, y compris rien*. Par exemple la commande `egrep 'foo.*bar'` imprime les lignes (de son entrée standard ou des fichiers arguments) qui contiennent à la fois `foo` puis `bar`. Pour voir aussi les lignes qui contiennent `bar` puis `foo`, on pourrait employer `'(foo.*bar)|(bar.*foo)'`.

Souvent, sous Unix, l'opérateur `+` indique que l'expression régulière sur laquelle il s'applique doit apparaître au moins une fois. Ici aussi, il s'agit d'une

simple facilité d'expression, car il va de soi que r_i`+` est équivalent à r_i`(`r_i`*)` et que r_i`*` est une autre façon d'écrire r_i`+?`.

11.1.3 Une extension Unix importante

Une extension Unix qui enrichit les expressions régulières (mais aussi qui les complique) est la possibilité de faire référence à une chaîne de caractère qui a déjà été rencontrée dans la partie gauche de l'expression régulière. Avec la contre oblique suivie d'un nombre i, on désigne la chaîne qui a rempli la $i$ème expression entre parenthèses. Ainsi `egrep '(.)\1'` permet de sélectionner toutes les lignes dans lesquelles le même caractère est redoublé.

Cette extension s'utilise largement pour les substitutions ; par exemple considérons la commande pour échanger les mots d'une ligne :

```
sed -e 's/\(.*\) et \(.*\)/\2 et \1/'
```

L'argument spécifie (avec `s` comme *substitute*) qu'il faut remplacer une expression régulière (encadrée par des obliques) par une chaîne (encadrée elle aussi par des obliques). L'expression régulière se compose de *n'importe quoi* (entre parenthèses) suivi de `et` suivi d'un deuxième *n'importe quoi* (lui aussi entre parenthèses). Ce qui le remplacera sera le deuxième n'importe quoi (`\2`) suivi de `et` suivi du premier n'importe quoi (`\1`).

11.2 Les automates finis déterministes

Les automates sont une façon graphique de se représenter un certain nombre de tâches, dont notamment le travail d'un analyseur lexical.

Un automate fini peut se représenter avec des *états*, qu'on peut voir comme les noeuds d'un graphe, et des *transitions* qu'on peut voir comme les arcs qui relient les noeuds du graphe entre eux, chacune étiquetée par un ou plusieurs symboles de l'alphabet d'entrée.

L'automate démarre dans l'état de départ, puis lit l'un après l'autre les caractères de la chaîne à traiter ; pour chaque caractère, il suit la transition vers un autre état qui devient le nouvel état courant. (S'il n'y a pas de transition étiquetée par le caractère courant, alors la chaîne ne correspond pas à celles reconnues par l'automate.) Quand la chaîne est terminée, si l'automate se trouve dans un des états marqués comme *« acceptation »*, cela signifie que la chaîne est reconnue par l'automate. On peut voir un tel automate élémentaire sur la figure 11.1.

Pour un exemple moins trivial, voyons un automate (erroné) pour reconnaître les commentaires du langage C sur la figure 11.2.

L'automate démarre dans l'état de gauche, qui indique qu'on est en dehors d'un commentaire. Pour tous les caractères sauf l'oblique (`/`), il suit la transition

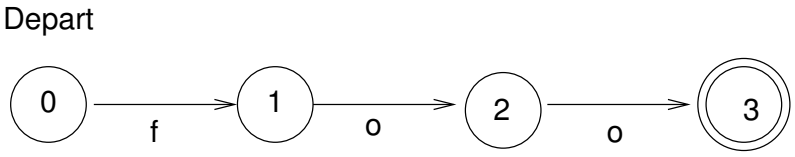


FIGURE 11.1 – Un automate élémentaire pour reconnaître la chaîne `foo`. Comme indiqué, l'automate démarre dans l'état 0. Quand il lit un `f`, il passe dans l'état 1. S'il lit ensuite un `o`, il passe dans l'état 2, d'où un autre `o` l'amène dans l'état 3 qui est un état d'acceptation. Toute autre entrée fait échouer l'automate : ce n'est pas `foo` qui a été lu.

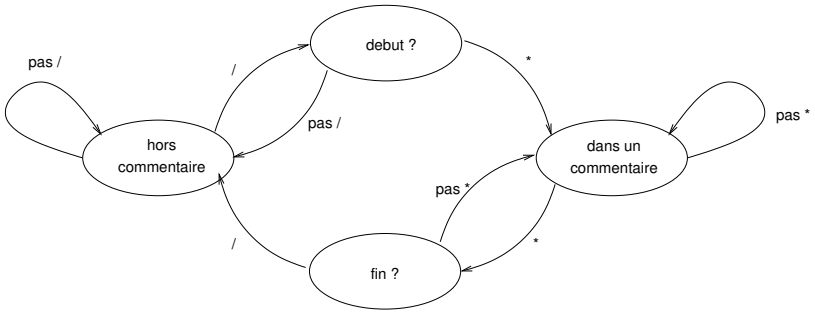


FIGURE 11.2 – Un automate qui tente de reconnaître les commentaires du langage C. Il ne traitera pas correctement la chaîne `/***/`.

qui le laisse dans le même état ; quand il rencontre une oblique, alors il passe dans l'état du haut qui indique qu'on est *peut-être* au début d'un commentaire (si on rencontre `/*`), mais peut-être pas (comme dans `a/2`). La distinction entre les deux cas se fait dans l'état du haut : si le caractère suivant est une *, alors on entre vraiment dans un commentaire et on passe dans l'état de droite ; en revanche, n'importe quoi d'autre fait revenir dans l'état qui indique qu'on sort d'un commentaire. La fin du commentaire avec `*/` fait passer de l'état de droite à l'état de gauche avec un mécanisme identique, sauf que les transitions se font d'abord sur `*` puis sur `/`.

Cet automate ne traitera pas correctement les commentaires qui contiennent un nombre impair d'étoiles avant la fermeture du commentaire, un cas qui se présente fréquemment dans les fichiers de ceux qui balisent leurs programmes avec des lignes remplies d'étoiles (personellement, je trouve cela de mauvais goût ; je préfère ponctuer mon fichier avec de espaces vierges comme dans les textes ordinaires ; j'estime que cela met mieux la structure du programme en évidence et facilite sa lecture).

L'exemple plus simple d'un tel échec se produit avec `/***/` : sur la première

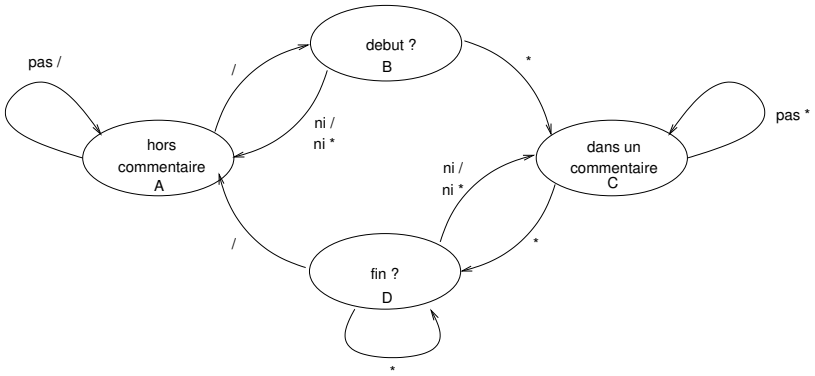


FIGURE 11.3 – L'automate de la figure 11.2 corrigé : maintenant, les commentaires C sont correctement reconnus, même quand ils se terminent par un nombre impair d'étoiles. J'ai également ajouté des noms aux états pour les nommer plus aisément.

oblique, l'automate passe de l'état de gauche à l'état du haut, puis de l'état du haut à l'état de droite sur la première étoile. La deuxième étoile fait passer de l'état de droite à celui du bas (c'est peut-être la fin d'un commentaire), et la troisième étoile ramène dans l'état de droite (en définitive, ce n'était pas la fin d'un commentaire) d'où l'automate ne sortira plus.

Un des avantages qu'il y a à représenter les automates sous forme de graphe, c'est qu'une fois que la cause de l'échec est identifiée, elle est facile à corriger : il suffit de rajouter à l'état du bas une transition vers lui-même sur le caractère étoile. Cet automate corrigé est présenté sur la figure 11.3.

Ex. 11.1 — Modifier l'automate de la figure 11.3 de manière à lui permettre de reconnaître aussi les commentaires à la C++, qui commencent par deux obliques et se prolongent jusqu'à la fin de la ligne.

Un autre avantage des automates de cette sorte est qu'ils sont faciles à transformer en programme. Par exemple, l'automate de la figure 11.3 peut facilement se traduire par le fragment de code :

```
enum { A, B, C, D }; // les états

void
automate(){
  int c;
  int etat = A;

  while((c = getchar()) != EOF){
    if (etat == A){
      if (c == '/')
```

```
        etat = B;
      /* else etat = A; */
    } else if (etat == B){
      if (c == '*')
        etat = C;
      else
        etat = A;
    } else if (etat == C){
       if (c == '*')
         etat = D;
       /* else etat == C; */
    } else if (etat == D){
       if (c == '/')
         etat = A;
       else if (c != *)
         etat = C;
       /* else etat = D; */
    }
  }
}
```

L'état courant est indiqué par une variable (nommée `état`) initialisée à l'état de départ. La fonction ne contient qu'une boucle qui lit un caractère et modifie l'état de départ en fonction de l'état courant et du caractère lu.

J'ai rajouté dans le code, en commentaire pour mémoire, les transitions qui maintiennent dans un état. Il serait facile d'y ajouter ce qu'il faut à la fonction pour n'imprimer que les commentaires, ou au contraire retirer les commentaires d'un programme.

On peut faire la même chose avec du code encore plus simple et des données un peu plus complexes, en construisant une table des transitions ; si on se limite aux codes de caractères sur 8 bits (ce qui n'est sans doute plus une très bonne idée en 2010, alors que le couple Unicode-UTF est en train de s'imposer), on peut faire :

```
enum {
  A = 0, B = 1, C = 2, D = 3,  // les états
  Netat = 4,
  Ncar = 256,
};

int trans[Netat][Ncar];

void
inittrans(){
  int i;
```

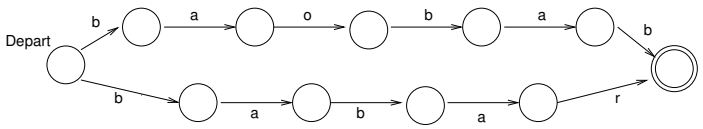


FIGURE 11.4 – Un automate non déterministe pour reconnaître les mots `baobab` ou `babar`.

```
  for(i = 0; i < Ncar; i++){ // par défaut
    etat[A][i] = etat[B][i] = A;
    etat[C][i] = etat[D][i] = C;
  }
  etat[A]['/'] = B;
  etat[B]['*'] = etat[C]['*'] = C;
  etat[D]['/'] = A;
}

void
automate()
  int c;
  int etat;

  while((c = getchar()) != EOF)
    etat = trans[etat][c];
}
```

11.3 Les automates finis non déterministes

Les automates finis que nous avons vus dans la section précédente sont dits *déterministes* parce que dans un état donné, le caractère lu ne permet de choisir qu'une seule transition : c'est ce qui permet de les traduire si facilement en code compact. Nous voyons ici des automates qui ne présentent pas cette caractéristique.

11.3.1 Les transitions multiples

Il est parfois plus facile de considérer des automates qui permettent d'atteindre plusieurs états différents avec le même caractère, comme celui, élémentaire, de la figure 11.4. Dans cet automate, à partir de l'état de départ, il y a *deux* transitions étiquetée avec `b` : ceci le rend *non déterministe* parce que dans la fonction de parcours on ne peut pas décider après n'avoir lu que le premier caractère laquelle des deux transitions il convient de suivre.

Pour transformer cet automate en programme, le premier réflexe d'un pro-

grammeur qui à intégré la récursivité consiste à utiliser le *backtracking* : tenter de prendre une des branches et si elle échoue revenir en arrière et essayer l'autre. Cela complique et ralentit considérablement le programme. (Au pire, au lieu d'avoir un temps de traitement proportionnel au nombre de caractères lus, on va en obtenir un proportionnel au nombre de transitions non déterministes par noeud à la puissance du nombre de caractères lus.)

Il existe une autre méthode bien plus facile à mettre en oeuvre et bien plus efficace, qui consiste à maintenir, au lieu d'un état courant, une *liste de tous les états courants possibles.* Pour ce faire, j'ai besoin de listes d'entiers (qui coderont les numéros d'états) ; je peux les réaliser de la façon simple suivante, parce qu'il ne sera nécessaire que d'ajouter des éléments dans la liste :

```
typedef struct List List;

enum {
  MaxNel = 1024,  // nombre maximum d'éléments
};

struct List {
  int nel;
  int el[MaxNel]; // il vaudrait mieux allouer dynamiquement
};

/* ajouter  --  ajouter un élément dans la liste s'il n'y est pas encore */
static inline void
ajouter(int val, List * l){
  int i;

  for(i = 0; i < l->nel; i++)
    if (l->el[i] == val)
      return; // déja dans la liste

  assert(l->nel < MaxNel);
  l->el[l->nel++] = val;
}
```

Pour représenter l'automate, on utilise une structure transition avec les états de départ et d'arrivée et le caractère qui permet de suivre la transition. L'automate se représente avec sa liste de transition, le numéro de l'état de départ et la liste des états d'arrivée.

```
typedef struct Trans Trans;
struct Trans {
  int  src;     // état de départ
  char car;     // caractère lu
  int  dst;     // état d'arrivée
```

```
};

Trans trans[Netat][MaxTrans]; // les transitions
int ntrans;                   // le nombre de transitions
int depart;                   // numéro de l'état de départ
List arrivee                  // les etats d'arrivee
```

Le code de la fonction de parcours de l'automate peut être le suivant (sans aucune amélioration) :

```
void
automate{}{
  List courant, prochain;
  int i, j;
  int etat;

  courant.nel = 0;
  ajouter(depart, &courant);
  while((c = getchar()) != EOF){
    prochain.nel = 0;
    for(i = 0; i < courant.nel; i++){
      etat = courant.el[i];
      for(j = 0; j < ntrans; j++)
        if (trans[j].src == courant && trans[j].car == car)
          ajouter(trans[j].dst, &prochain);
   if (prochain.nel == 0)
      return 0;
    courant = prochain;
  }
  return member(courant, arrivee);
}
```

La boucle `while` lit chaque caractère. Pour ce caractère, la première boucle `for` balaye la liste des états courants possibles. Pour chacun d'eux, la seconde boucle `for` examine toutes les transitions qui en émanent ; si elle est étiquetée par le caractère lu, la destination est ajoutée dans la liste des nouveaux états possibles si elle n'y était pas encore.

La méthode fonctionne parce que la seule chose qui nous intéresse est l'état courant, sans se soucier du chemin employé pour l'atteindre. Elle n'est pas sans rapport avec la programmation dynamique.

11.3.2 Les transitions epsilon

Il existe une seconde manière pour un automate fini d'être non déterministe. Il est parfois pratique d'utiliser des transitions que l'automate est libre de prendre ou pas, sans lire de caractère. On les appelle des *transitions epsilon*

(parce qu'elle sont ordinairement étiquetées avec le caractère ϵ). L'automate est alors non déterministe parce qu'il n'y a pas moyen de déterminer, simplement en regardant le prochain caractère si une transition epsilon doit être prise ou pas.

Les transitions epsilon, sont très utiles pour lier entre eux des automates, comme montré dans la figure 11.5.

Le parcours d'un automate avec des transitions epsilon n'est pas beaucoup plus complexe qu'avec un automate doté de transitions multiples. On commence par définir une fonction de fermeture qui rajoute à une liste d'états ceux qu'on peut atteindre avec une transition epsilon :

```
void
epsilonfermer(List * l){
  int i, j, etat;

  for(i = 0; i < l->nel; i++){
    etat = l->el[i];
    for(j = 0; j < ntrans; j++)
      if (trans[j].src == etat && trans[j].car == EOF)
        ajouter(trans[j].dst, l);
  }
}
```

La fonction se contente d'ajouter à une liste d'états future ceux qu'on peut atteindre à partir des états de la liste présente avec une ou plusieurs transitions epsilon. J'ai utilisé la valeur `EOF` pour coder epsilon, puisque je suis certain que ce n'est pas le code d'un caractère valide.

Pour compléter la liste avec les états qu'on atteint après plusieurs transitions epsilon, il est important que la fonction `ajouter` place le nouvel élément à la fin de la liste. De cette manière, les transitions epsilon qui partent des états ajoutés seront examinées aussi.

Finalement, il suffit de modifier la fonction de parcours pour effectuer la fermeture sur chaque nouvelle liste d'états construits en remplaçant la ligne :

```
    courant = prochain;
```

par

```
    courant = prochain;
    epsilonfermer(&courant);
```

11.3.3 Les automates finis déterministes ou pas

Le parcours d'un automate non déterministe ressemble à son traitement par un interprète. Il est possible aussi d'effectuer sa *compilation* en énumérant tous

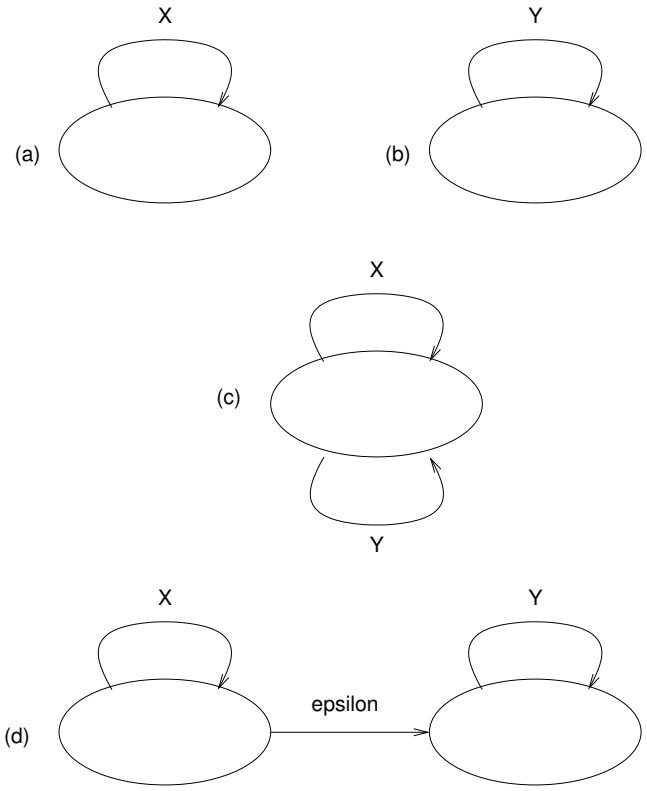


FIGURE 11.5 – L'automate (a) reconnaît n'importe quel nombre de `X`, l'automate (b) reconnaît n'importe quel nombre de `Y`. Pour reconnaître n'importe quel nombre de X suivi de n'importe quel nombre de Y, la tentative naïve de fusion de ces deux automates en (c) ne fonctionne pas, puisqu'elle reconnaît n'importe quel *mélange* de X et de Y. En revanche, l'utilisation d'une transition epsilon en (d) permet d'enchaîner facilement les deux automates.

les groupes d'états dans lesquels il est possible de se trouver et en plaçant des transitions entre ces groupes d'états. L'automate ainsi construit est un automate déterministe. Un algorithme pour effectuer ce travail que j'appelle la *détermi-nisation* d'un automate se trouve dans le *Dragon Book*.

Cette construction démontre un résultat important et un peu étonnant au premier abord : les automates non déterministes ne sont pas plus puissants que les automates déterministes : tout ce qu'il est possible de décrire avec l'un peut aussi se décrire avec l'autre.

(Pour le folklore) On peut renverser un automate : il suffit de transformer l'état de départ en état d'acceptation, d'ajouter un nouvel état de départ et de placer des transitions epsilon depuis cet état de départ vers les anciens états d'acceptation. Si on prend un automate, qu'on le renverse, qu'on déterminise le résultat, qu'on le renverse de nouveau et qu'on déterminise le résultat, alors on obtient un automate équivalent à celui de départ, mais qui est minimal en nombre d'état et de transitions. Je ne connais pas de source écrite pour cette méthode qui m'a été indiquée par Harald Wertz. Elle est sans doute moins efficace mais beaucoup plus élégante que la méthode de minimisation des automates indiquée dans le *Dragon Book*.

11.4 Des expressions régulières aux automates

Les automates sont une façon pratique de raisonner sur certains problèmes et ils permettent de structurer des programmes puissants et rapides, mais ils présentent un problème : s'il est aisé de les dessiner, en revanche ils ne sont pas faciles à décrire à un ordinateur avec les outils usuels d'interaction.

Je montre dans cette section que les automates finis et les expressions régulières sont équivalents : ce qu'on peut décrire avec une expression régulière peut aussi être décrit avec un automate. Or les expressions régulières sont (relativement) aisées à décrire avec un clavier.

11.4.1 État d'acceptation unique

Nous aurons besoin dans les étapes qui suivent d'avoir des automates qui n'ont qu'un seul état d'acceptation. Ceci est facile à construire à partir d'un automate avec plusieurs états d'acceptation. Il suffit d'ajouter un nouvel état qui sera l'état d'acceptation et de placer des transitions epsilon entre les anciens états d'acceptation et ce nouvel état. Je représente un tel automate dans les figures qui suivent par une forme indéterminée dont n'émergent que l'état de départ et l'état d'acceptation, comme dans la figure 11.6.

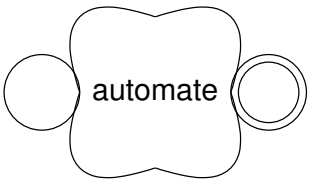


FIGURE 11.6 – Une représentation d'un automate lorsque les seules choses importantes sont son état de départ et son (unique) état d'acceptation. Le détail des états et des transitions intermédiaires n'apparaît pas sur la figure.

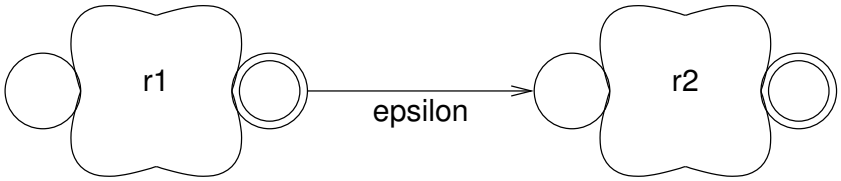


FIGURE 11.7 – Construction de l'automate équivalent à $r_1 r_2$ à partir des automates équivalents à r_1 et r_2.

11.4.2 La brique de base

Il va sans dire que les expressions régulières qui se composent d'un seul caractères sont équivalentes à un automate avec un état de départ et un état d'acceptation joints par une unique transition étiquetée par ce caractère.

11.4.3 Les opérateurs des expressions régulières

Étant donné deux automates qui sont équivalents à deux expressions régulières r_1 et r_2, on construit un automate équivalent à l'expression régulière $r_1 r_2$ en ajoutant une transition epsilon entre l'état de d'acceptation du premier vers l'état de départ du second, comme sur la figure 11.7

Étant donné deux automates qui sont équivalents à deux expressions régulières r_1 et r_2, on construit un automate équivalent à l'expression régulière $r_1 | r_2$ en ajoutant un nouvel état de départ et un nouvel état d'arrivée, deux transitions epsilon de l'état de départ vers les anciens états de départ et deux transitions epsilon depuis les anciens états d'arrivée vers le nouvel état d'arrivée, comme dans la figure 11.8.

Étant donné un automate équivalent à l'expression régulière r, on peut facilement construire l'automate équivalent à l'expression régulière `r?` en ajoutant un nouvel état de départ et un nouvel état d'acceptation relié à l'ancien état de départ et à l'ancien état d'acceptation par des transitions epsilon, avec entre elles une autre transition epsilon qui permet d'éviter de passer par l'automate,

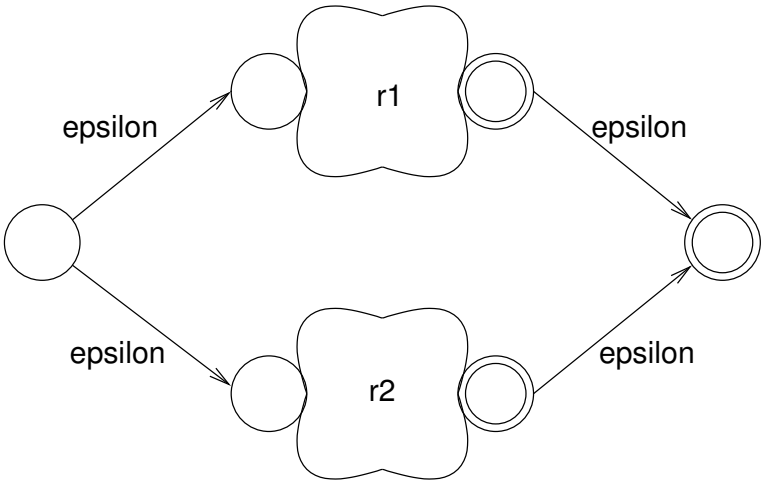


FIGURE 11.8 – Construction de l'automate équivalent à $r_1 | r_2$ à partir des automates équivalents à r_1 et r_2.

comme dans la figure 11.9

Finalement, pour la fermeture de Kline, il suffit de d'ajouter à la figure précédente une transition epsilon qui permet de revenir dans l'état de départ une fois qu'on a atteint l'état d'acceptation, comme dans la figure 11.10.

11.4.4 C'est tout

Parce que nous avons montré comment chaque mécanisme de construction des expressions régulières peut se traduire en un mécanisme équivalent de construction des automates, nous avons démontré que tout ce que peuvent les automates, les expressions régulières le peuvent aussi.

En pratique, les mécanismes exposés ici sont souvent trop lourds pour les expressions régulières réellement manipulées : il a été nécessaire d'être prudent pour que tous les cas soient traités correctement. Si, comme c'est souvent le cas, l'état d'acceptation n'a pas de transition sortante, il est possible d'utiliser les états d'acceptation des automates constituants et ainsi d'éviter l'ajout d'un nouvel état d'acceptation et de transitions vide. De même, si l'état de départ n'a pas de transition entrante, l'ajout d'un nouvel état de départ est inutile : on peut fusionner les états de départ des automates initiaux.

Nous n'avons pas montré comment construire des expressions régulières à partir d'un automate, et donc il est possible d'imaginer que les expressions régulières permettent d'exprimer des choses qui sont hors de portée des automates. Ce n'est pas le cas, mais je vous demande de me croire sur parole sur ce point. J'invite les incrédules à consulter l'ouvrage de J. AHO et R. SETHI *Les concepts*

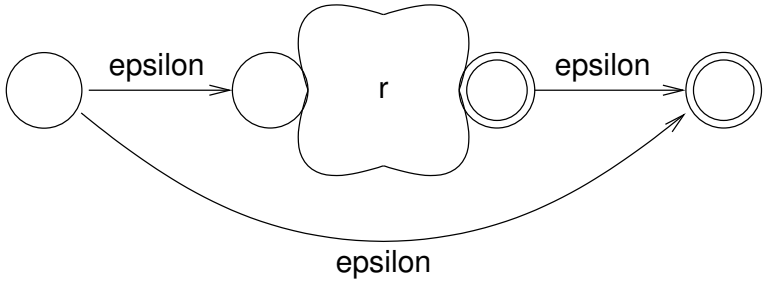


FIGURE 11.9 – Construction de l'automate équivalent à $r?$ à partir de l'automate équivalent à r. La transition du bas permet de passer directement de l'état de départ à l'état d'arrivée sans avoir reconnu l'expression régulière r.

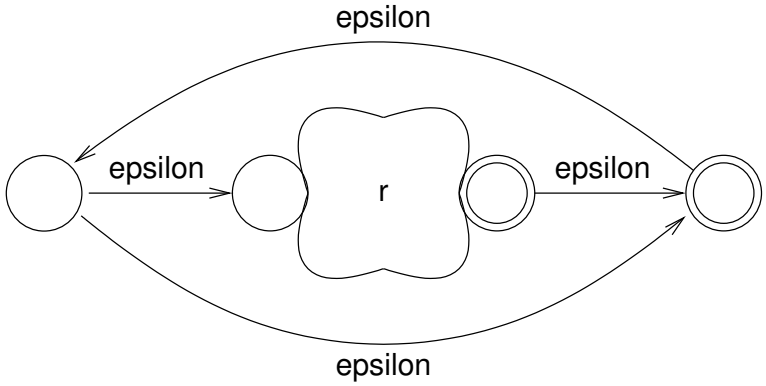


FIGURE 11.10 – Construction de l'automate équivalent à $r*$ à partir de l'automate équivalent à r. La transition du haut permet de revenir dans l'état de départ après avoir reconnu l'expression régulière r.

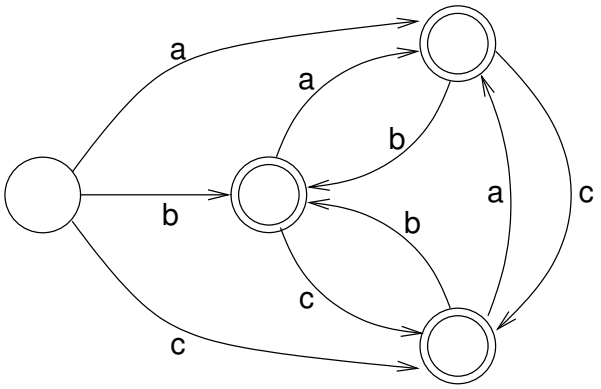


FIGURE 11.11 – Cet automate accepte n'importe quelle séquence de `a`, `b` et `c` à condition que le même caractère n'apparaisse pas deux fois de suite.

fondamentaux de l'informatique traduit chez InterEditions.

Ex. 11.2 — (assez facile) Traduire l'automate qui reconnaît les commentaires du langage C en expression régulière.

Ex. 11.3 — (difficile) L'automate de la figure 11.11 accepte toutes les suites (non vides) de `a`, `b` et `c` dans lesquelles il n'y a pas deux fois le même caractère de suite. Trouver une expression régulière équivalente. (D'après AHO et sc Ullman, *Aspects fondamentaux de l'informatique*

11.5 La limite des expressions régulières

Ex. 11.4 — (instructif) Dessiner un automate fini, déterministe ou pas, qui reconnaît un nombre quelconque de A suivi du *même* nombre de B.

11.6 Lex et Flex

On trouve sous Unix une commande `lex` qu'on peut utiliser pour construire automatiquement un analyseur lexical à partir d'expressions régulières. La version du projet Gnu de cette commande s'appelle `flex`.

Le principal avantage de `lex`, c'est qu'il a été conçu pour s'interfacer aisément avec les analyseurs lexicaux fabriqués par `yacc`. Pour le reste, les analyseurs lexicaux qu'il produit sont gros et pas particulièrement rapides : on peut l'utiliser comme un outil de prototypage rapide, mais il est souvent peu judicieux de s'en servir en production dans les systèmes critiques.

Chapitre 12

Projet et évaluation

Ce court chapitre présente le projet à réaliser ; il est aussi destiné à vous permettre de donner votre évaluation du cours.

12.1 Projet

Vous devez entrer en contact avec moi afin de nous mettre d'accord avec un projet que vous devrez réaliser pour la validation du cours.

Le contenu du projet est relativement ouvert. Il peut parfaitement s'intégrer dans quelque chose que vous réalisez par ailleurs, par exemple pour validation d'un autre cours.

Le projet *doit* contenir l'écriture d'une grammaire non triviale pour `yacc`, dont le parseur produit des données qui sont ensuite utilisées, ou bien une modification significative d'une grammaire complexe.

Si vous n'avez pas d'idée, indiquez moi ce qui vous intéresse ; j'en aurai peut-être une.

Quelques exemples de projets réalisés les années précédentes :
- un assembleur pour l'ordinateur en papier
- une ré-implémentation de la commande `make` (simplifiée).
- une modification de ppcm pour lui faire produire du code directement exécutable en mémoire à la place de l'assembleur.
- diverses manipulations sur des fichiers XML.

12.2 Évaluation

Merci de prendre aussi le temps de répondre aux questions suivantes pour m'aider à évaluer la façon dont le cours s'est passé de votre point de vue.

Notez que l'idée est d'essayer d'améliorer les choses, aussi les réponses exces-

sives sont à peu près inutiles. Par exemple avec *« Le chapitre le plus intéressant ? Aucun. Le chapitre le moins intéressant ? Tous. »*, on ne peut pas faire grand chose.

- Le contenu du cours a-t-il correspondu à ce que vous attendiez ? (si non, de quelle manière)
- Qu'est-ce qui vous a le plus surpris (en bien) ?
- Qu'est-ce qui vous a le plus déçu ?
- Quels étaient les chapitres les plus difficiles ?
- Quels étaient les chapitres les plus faciles ?
- Quels étaient les chapitres les plus intéressants ?
- Quels étaient les chapitres les moins intéressants ?
- Que me suggéreriez-vous de modifier dans l'ordre des chapitres ?
- Qu'est-ce qui est en trop dans le cours ?
- Qu'est-ce qui manque dans le cours ?
- Comment étaient les exercices du point de vue quantité (pas assez, trop).
- Comment étaient les exercices du point de vue difficulté (trop durs, trop faciles) ?
- Que donneriez-vous comme conseil à un étudiant qui va suivre le cours ?
- Que me donneriez-vous comme conseil en ce qui concerne le cours ?
- Si vous deviez mettre une note globale au cours, entre 0 et 20, laquelle mettriez-vous ?
- Quelles questions manque-t-il pour évaluer correctement le cours (et bien évidemment, quelle réponse vous y apporteriez) ?

Chapitre 13

Documments annexes

13.1 Les parseurs à précédence d'opérateur

13.1.1 Le fichier `src/ea-oper0.c`

```
/* oper0.c
 Un parseur a precedence d'operateurs avec deux piles en guise de premier analys
 Version sans parentheses

 Exercice facile : ajouter l'operateur %
*/
# include <stdio.h>
# include <ctype.h>
# include <assert.h>
# include <string.h>

enum {
  Num = 1,                      // renvoye par l'analyseur lexical
  Executer = -1, Empiler = 1,   // comparaison de precedence
  MaxStack = 1024,              // taille maximum des piles
};

char * operator = "+-*/";       // la liste des operateurs
int yylval;                     // la valeur du lexeme

/* yylex  --  lit le prochain mot sur stdin,
  place sa valeur dans yylval,
  renvoie son type,
  proteste pour les caracteres invalides
 */
```

```
int
yylex(void){
  int c;
  int t;

 redo:
  while(isspace(c = getchar()))
    if (c == '\n')
      return 0;

  if (c == EOF)
    return 0;

  if (isdigit(c)){
    ungetc(c, stdin);
    t = scanf("%d", &yylval);
    assert(t == 1);
    return Num;
  }

  if (strchr(operator, c) == 0){
    fprintf(stderr, "Caractere %c (\\0%o) inattendu\n", c, c);
    goto redo;
  }
  return c;
}


int operande[MaxStack];
int operateur[MaxStack];
int noperande, noperateur;

/* preccmp  --  prec. de l'operateur gauche - prec. de l'operateur droit */
int
preccmp(int gop, int dop){
  assert(gop != 0);
  if (dop == 0)                 // EOF : executer ce qui reste.
    return Executer;

  if (gop == dop)               // le meme : executer
    return Executer;

  if (gop == '+' || gop == '-'){ // + ou -
    if (dop == '+' || dop == '-')
      return Executer; // puis + ou - : executer
    else
```

```
71        return Empiler; //  puis * ou / : empiler
72    }
73
74    return -1; // dans tous les autres cas, executer
75  }
76
77  void
78  executer(int op){
79    int t;
80
81    switch(op){
82    default:
83      fprintf(stderr, "Operateur impossible, code %c (\\0%o)\n", op, op);
84      return;
85    case '+':
86      t = operande[--noperande];
87      t += operande[--noperande];
88      operande[noperande++] = t;
89      return;
90    case '-':
91      t = operande[--noperande];
92      t = operande[--noperande] - t;
93      operande[noperande++] = t;
94      return;
95    case '*':
96      t = operande[--noperande];
97      t *=operande[--noperande];
98      operande[noperande++] = t;
99      return;
100   case '/':
101     t = operande[--noperande];
102     t = operande[--noperande] / t;
103     operande[noperande++] = t;
104     return;
105   }
106 }
107
108 void
109 analyser(void){
110   int mot;
111
112   noperateur = noperande = 0;
113   do {
114     mot = yylex();
115
```

```
116     if (mot == Num){
117       assert(noperande < MaxStack);
118       operande[noperande++] = yylval;
119 
120     } else {
121       while(noperateur > 0 && preccmp(operateur[noperateur - 1], mot) < 0)
122         executer(operateur[--noperateur]);
123       assert(noperateur < MaxStack);
124       operateur[noperateur++] = mot;
125     }
126   } while(mot != 0);
127 
128   if (noperateur != 1 || noperande != 1 || operateur[0] != 0)
129     fprintf(stderr, "Erreur de syntaxe\n");
130   else
131     printf("%d\n", operande[0]);
132 }
133 
134 
135 int
136 main(){
137   analyser();
138   return 0;
139 }
```

13.1.2 Le fichier src/eb-oper1.c

```
1  /* oper1.c
2   Un parseur a precedence d'operateurs avec deux piles en guise de premier analys
3   Version avec les parentheses
4 
5   Exercice facile : ajouter l'operateur %
6  */
7  # include <stdio.h>
8  # include <ctype.h>
9  # include <assert.h>
10 # include <string.h>
11 
12 enum {
13   Num = 1,                      // renvoye par l'analyseur lexical
14   Executer = -1, Empiler = 1,   // comparaison de precedence
15   MaxStack = 1024,              // taille maximum des piles
16 };
17 
18 char * operator = "+-*/()";     // la liste des operateurs
```

```
19 int yylval;                       // la valeur du lexeme
20
21 /* yylex  --  lit le prochain mot sur stdin,
22   place sa valeur dans yylval,
23   renvoie son type,
24   proteste pour les caracteres invalides
25  */
26 int
27 yylex(void){
28   int c;
29   int t;
30
31  redo:
32   while(isspace(c = getchar()))
33     if (c == '\n')
34       return 0;
35
36   if (c == EOF)
37     return 0;
38
39   if (isdigit(c)){
40     ungetc(c, stdin);
41     t = scanf("%d", &yylval);
42     assert(t == 1);
43     return Num;
44   }
45
46   if (strchr(operator, c) == 0){
47     fprintf(stderr, "Caractere %c (\\0%o) inattendu\n", c, c);
48     goto redo;
49   }
50   return c;
51 }

52
53 int operande[MaxStack];
54 int operateur[MaxStack];
55 int noperande, noperateur;
56
57 /* preccmp  --  prec. de l'operateur gauche - prec. de l'operateur droit */
58 int
59 preccmp(int gop, int dop){
60   assert(gop != 0);
61   if (dop == 0)                  // EOF : executer ce qui reste.
62     return Executer;
63
```

```
64   if (gop == ')')                // toujours executer la parenthese fermante
65     return Executer;
66
67   if (dop == ')'){               // avec une nouvelle fermante :
68     if (gop == '(')
69       return Empiler;            //   l'empiler sur son ouvrante
70     else
71       return Executer;           //   et executer sur tous les autres.
72   }
73
74   if (dop == '(')                // toujours empiler les nouvelles ouvrantes
75     return Empiler;
76
77   if (gop == dop)                // le meme : executer
78     return Executer;
79
80   if (gop == '+' || gop == '-'){ // + ou -
81     if (dop == '+' || dop == '-')
82       return Executer;           // puis + ou - : executer
83     else
84       return Empiler;            //  puis * ou / : empiler
85   }
86
87   return Executer; // dans tous les autres cas, executer
88 }
89
90 void
91 executer(int op){
92   int t;
93
94   switch(op){
95   default:
96       fprintf(stderr, "Operateur impossible, code %c (\\0%o)\n", op, op);
97     return;
98   case '+':
99     t = operande[--noperande];
100    t += operande[--noperande];
101    operande[noperande++] = t;
102    return;
103  case '-':
104    t = operande[--noperande];
105    t = operande[--noperande] - t;
106    operande[noperande++] = t;
107    return;
108  case '*':
```

```
109     t = operande[--noperande];
110     t *=operande[--noperande];
111     operande[noperande++] = t;
112     return;
113   case '/':
114     t = operande[--noperande];
115     t = operande[--noperande] / t;
116     operande[noperande++] = t;
117     return;
118   case '(':
119     fprintf(stderr, "Ouvrante sans fermante\n");
120     return;
121   case ')':
122     if (noperateur == 0){
123       fprintf(stderr, "Fermante sans ouvrante\n");
124       return;
125     }
126     t = operateur[--noperateur];
127     if (t != '('){
128       fprintf(stderr, "Cas impossible avec la parenthese fermante\n");
129       return;
130     }
131   }
132 }
133
134 void
135 analyser(void){
136   int mot;
137
138   noperateur = noperande = 0;
139   do {
140     mot = yylex();
141
142     if (mot == Num){
143       assert(noperande < MaxStack);
144       operande[noperande++] = yylval;
145
146     } else {
147       while(noperateur > 0 && preccmp(operateur[noperateur - 1], mot) < 0)
148         executer(operateur[--noperateur]);
149       assert(noperateur < MaxStack);
150       operateur[noperateur++] = mot;
151     }
152   } while(mot != 0);
153
154   if (noperateur != 1 || noperande != 1 || operateur[0] != 0)
```

```
155     fprintf(stderr, "Erreur de syntaxe\n");
156   else
157     printf("%d\n", operande[0]);
158 }
159 
160 
161 int
162 main(){
163   analyser();
164   return 0;
165 }
```

13.2 Les petits calculateurs avec Yacc

13.2.1 Le fichier src/ed-1-calc-y

```
1  /* 1-calc.y
2   Une calculette elementaire avec Yacc
3 
4   Les expressions arithmetiques avec les quatre operateurs et les parentheses
5   */
6  %{
7  # define YYDEBUG 1              /* Pour avoir du code de mise au point */
8  int yydebug;
9 
10 int yylex(void);
11 int yyerror(char *);
12 %}
13 
14 %term NBRE                     /* Les symboles renvoyes par yylex */
15 
16 %left '+' '-'                  /* Precedence et associativite */
17 %left '*' '/'
18 %right EXP
19 %left FORT
20 
21 %start exprs                   /* le symbole de depart */
22 
23 %%
24 exprs : /* rien */
25                 { printf("? "); }
26       | exprs expr '\n'
27                 { printf("%d\n? ", $2); }
28       ;
29 
```

```
30 expr : NBRE
31                 { $$ = $1; }
32      | expr '+' expr
33                 { $$ = $1 + $3; }
34      | expr '-' expr
35                 { $$ = $1 - $3; }
36      | expr EXP expr
37                 { int i;
38                   $$ = 1;
39                   for(i = 0; i < $3; i++)
40                     $$ *= $1;
41                 }
42      | expr '*' expr
43                 { $$ = $1 * $3; }
44      | expr '/' expr
45                 { $$ = $1 / $3; }
46      | '-' expr          %prec FORT
47                 { $$ = - $2; }
48      | '(' expr ')'
49                 { $$ = $2; }
50      ;
51 %%
52 # include <stdio.h>
53 # include <assert.h>
54
55 static int ligne = 1;           /* numero de ligne courante */
56
57 int
58 main(){
59   yyparse();
60   puts("Bye");
61   return 0;
62 }
63
64 /* yyerror  --  appele par yyparse en cas d'erreur */
65 int
66 yyerror(char * s){
67   fprintf(stderr, "%d: %s\n", ligne, s);
68   return 0;
69 }

70
71 /* yylex  --  appele par yyparse, lit un mot, pose sa valeur dans yylval
72    et renvoie son type */
73 int
74 yylex(){
```

```
75   int c;
76
77  re:
78   switch(c = getchar()){
79   default:
80     fprintf(stderr, "'%c' : caractere pas prevu\n", c);
81     exit(1);
82
83   case ' ': case '\t':
84     goto re;
85
86   case EOF:
87     return 0;
88
89   case '0': case '1': case '2': case '3': case '4':
90   case '5': case '6': case '7': case '8': case '9':
91     ungetc(c, stdin);
92     assert(scanf("%d", &yylval) == 1);
93     return NBRE;
94
95   case '\n':
96     ligne += 1;
97     /* puis */
98   case '+': case '-': case '*': case '/': case '(': case ')':
99     return c;
100
101  case '^':
102    return EXP;
103  }
104 }
```

13.2.2 Le fichier src/ed-3-calc.y

```
1  /* 3-calc.y
2   Une calculette elementaire avec Yacc
3   Construction explicite de l'arbre syntaxique en memoire
4   */
5  %{
6  # define YYDEBUG 1              /* Pour avoir du code de mise au point */
7
8  typedef struct Noeud Noeud;
9
10 Noeud * noeud(int, Noeud *, Noeud *);
11 Noeud * feuille(int, int);
12 void print(Noeud *);
13
```

```
14 int yylex(void);
15 int yyerror(char *);
16 %}
17
18 %union {
19   int i;
20   Noeud * n;
21 };
22
23 %token <i> NBRE                 /* Les symbole renvoyes par yylex */
24 %type <n> expr                  /* Type de valeur attache au noeuds expr */
25
26 %left <i> ADD                   /* Precedence et associativite */
27 %left <i> MUL
28 %left '^'
29 %left UMOINS
30
31 %start exprs                    /* le symbole de depart */
32
33 %%
34 exprs : /* rien */
35                 { printf("? "); }
36       | exprs expr '\n'
37                 { print($2);
38                   printf("? "); }
39       ;
40
41 expr : NBRE
42                 { $$ = feuille(NBRE, $1); }
43      | expr ADD expr
44                 { $$ = noeud($2, $1, $3); }
45      | expr '^' expr
46                 { $$ = noeud('^', $1, $3); }
47      | expr MUL  expr
48                 { $$ = noeud($2, $1, $3); }
49      | '-' expr          %prec UMOINS
50                 { $$ = noeud(UMOINS, $2, 0); }
51      | '(' expr ')'
52                 { $$ = $2; }
53      ;
54 %%
55 # include <stdio.h>
56 # include <assert.h>
57
58 static int ligne = 1;           /* numero de ligne courante */
59
```

```
60 int yydebug = 0;                  /* different de 0 pour la mise au point */
61
62 int
63 main(){
64   yyparse();
65   puts("Bye");
66   return 0;
67 }
68
69 /* yyerror  --  appeler par yyparse en cas d'erreur */
70 int
71 yyerror(char * s){
72   fprintf(stderr, "%d: %s\n", ligne, s);
73   return 0;
74 }
75
76 /* yylex  --  appele par yyparse, lit un mot, pose sa valeur dans yylval
77    et renvoie son type */
78 int
79 yylex(){
80   int c;
81   int r;
82
83  re:
84   switch(c = getchar()){
85   default:
86     fprintf(stderr, "'%c' : caractere pas prevu\n", c);
87     goto re;
88
89   case ' ': case '\t':
90     goto re;
91
92   case EOF:
93     return 0;
94
95   case '0': case '1': case '2': case '3': case '4':
96   case '5': case '6': case '7': case '8': case '9':
97     ungetc(c, stdin);
98     assert(scanf("%d", &r) == 1);
99     yylval.i = r;
100    return NBRE;
101
102  case '\n':
103    ligne += 1;
104    return c;
105
```

```
106   case '+': case '-':
107     yylval.i = c;
108     return ADD;
109
110   case '*': case '/':
111     yylval.i = c;
112     return MUL;
113
114   case '(': case ')': case '^':
115     return c;
116   }
117 }
118
119 /*
120  Representation d'un noeud de l'arbre qui represente l'expression
121 */
122 struct Noeud {
123   int type;
124
125   int val;                      /* valeur pour les feuilles */
126   Noeud * gauche, * droit;      /* enfants pour les autres */
127 };
128
129 /* nouvo  --  alloue un nouveau noeud */
130 static Noeud *
131 nouvo(void){
132   Noeud * n;
133   void * malloc();
134
135   n = malloc(sizeof n[0]);
136   if (n == 0){
137     fprintf(stderr, "Plus de memoire\n");
138     exit(1);
139   }
140   return n;
141 }
142 /* feuille  --  fabrique une feuille de l'arbre */
143 Noeud *
144 feuille(int t, int v){
145   Noeud * n;
146
147   n = nouvo();
148   n->type = t;
149   n->val = v;
150   return n;
151 }
```

```
152
153 /* noeud  --  fabrique un noeud interne de l'abre */
154 Noeud *
155 noeud(int t, Noeud * g, Noeud * d){
156   Noeud * n;
157
158   n = nouvo();
159   n->type = t;
160   n->gauche = g;
161   n->droit = d;
162   return n;
163 }
164
165 /* puissance  --  calcule (mal) n a la puissance p */
166 static int
167 puissance(int n, int p){
168   int r, i;
169
170   r = 1;
171   for(i = 0; i < p; i++)
172     r *= n;
173   return r;
174 }
175
176 /* eval  --  renvoie la valeur attachee a un noeud */
177 static int
178 eval(Noeud * n){
179   switch(n->type){
180   default:
181     fprintf(stderr, "eval : noeud de type %d inconnu\n", n->type);
182     exit(1);
183   case NBRE:
184     return n->val;
185   case UMOINS:
186     return - eval(n->gauche);
187   case '+':
188     return eval(n->gauche) + eval(n->droit);
189   case '-':
190     return eval(n->gauche) - eval(n->droit);
191   case '*':
192     return eval(n->gauche) * eval(n->droit);
193   case '/':
194     return  eval(n->gauche) / eval(n->droit);
195   case '^':
196     return puissance(eval(n->gauche), eval(n->droit));
197   }
```

```
198 }
199
200 /* parenthese  --  ecrit une expression completement parenthesee */
201 static void
202 parenthese(Noeud * n){
203   switch(n->type){
204   default:
205     fprintf(stderr, "eval : noeud de type %d inconnu\n", n->type);
206     exit(1);
207
208   case NBRE:
209     printf("%d", n->val);
210     break;
211
212   case UMOINS:
213     printf("-(");
214     parenthese(n->gauche);
215     putchar(')');
216     break;
217
218   case '+':  case '-':  case '*':  case '/':  case '^':
219     putchar('(');
220     parenthese(n->gauche);
221     putchar(n->type);
222     parenthese(n->droit);
223     putchar(')');
224   }
225 }
226
227 /* freenoeuds  --  libere la memoire attachee a un noeud et ses enfants */
228 static void
229 freenoeuds(Noeud * n){
230   switch(n->type){
231   default:
232     fprintf(stderr, "freenoeuds : noeud de type %d inconnu\n", n->type);
233     exit(1);
234
235   case NBRE:
236     break;
237
238   case UMOINS:
239     freenoeuds(n->gauche);
240     break;
241
242   case '+':  case '-':  case '*':  case '/':  case '^':
```

```
243     freenoeuds(n->gauche);
244     freenoeuds(n->droit);
245   }
246   free(n);
247 }
248 
249 /* print  --  imprime l'expression parenthesee et sa valeur */
250 void
251 print(Noeud * n){
252   parenthese(n);                /* expression parenthesee */
253 
254   printf(" = %d\n", eval(n));   /* valeur */
255 
256   freenoeuds(n);
257 }
```

13.2.3 Le fichier src/ee-2-calc.y

```
1  /* 2-calc.y
2   Une calculette elementaire avec Yacc
3   Noeuds de types float,
4   Recuperation des erreurs
5   */
6  %{
7  # define YYDEBUG 1              /* Pour avoir du code de mise au point */
8  
9  int yylex(void);
10 int yyerror(char *);
11 %}
12 
13 %union {                       /* le type des valeurs attachees aux noeuds */
14   float f;
15 }
16 
17 %term <f> NBRE                 /* Les symboles renvoyes par yylex */
18 %type <f> expr                 /* Type de valeur attache aux noeuds expr */
19 
20 %left '+' '-'                  /* Precedence et associativite */
21 %left '*' '/'
22 %left EXP
23 %left FORT
24 
25 %start exprs                   /* le symbole de depart */
26 
27 %%
28 exprs : /* rien */
```

```
29                  { printf("? "); }
30        | exprs expr '\n'
31                  { printf("%g\n? ", $2); }
32        | exprs error '\n'
33        ;
34
35  expr : NBRE
36                  { $$ = $1; }
37        | expr '+' expr
38                  { $$ = $1 + $3; }
39        | expr '-' expr
40                  { $$ = $1 - $3; }
41        | expr EXP expr
42                  { int i;
43                    $$ = 1;
44                    for(i = 0; i < $3; i++)
45                      $$ *= $1;
46                  }
47        | expr '*' expr
48                  { $$ = $1 * $3; }
49        | expr '/' expr
50                  { $$ = $1 / $3; }
51        | '-' expr          %prec FORT
52                  { $$ = - $2; }
53        | '(' expr ')'
54                  { $$ = $2; }
55        ;
56  %%
57  # include <stdio.h>
58  # include <assert.h>
59
60  static int ligne = 1;           /* numero de ligne courante */
61
62  int yydebug = 0;                /* different de 0 pour la mise au point */
63
64  int
65  main(){
66    yyparse();
67    puts("Bye");
68    return 0;
69  }
70  /* yyerror  --  appeler par yyparse en cas d'erreur */
71  int
72  yyerror(char * s){
73    fprintf(stderr, "%d: %s\n", ligne, s);
74    return 0;
```

```
}

/* yylex  --  appele par yyparse, lit un mot, pose sa valeur dans yylval
   et renvoie son type */
int
yylex(){
  int c;

 re:
  switch(c = getchar()){
  default:
    fprintf(stderr, "'%c' : caractere pas prevu\n", c);
    goto re;

  case ' ': case '\t':
    goto re;

  case EOF:
    return 0;

  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    ungetc(c, stdin);
    assert(scanf("%f", &yylval.f) == 1);
    return NBRE;

  case '\n':
    ligne += 1;
    /* puis */
  case '+': case '-': case '*': case '/': case '(': case ')':
    return c;

  case '^':
    return EXP;
  }
}
```

13.3 La grammaire C de gcc

```
/*
 La grammaire du langage en C, extrait des sources de gcc (v. 2.95.2),
 fichier c-parse.y, sans les actions, un peu simplifie.

 jm, fevrier 2000
 */
```

```
7
8  %start program
9
10 %union {long itype; tree ttype; enum tree_code code;
11         char *filename; int lineno; int ends_in_label; }
12
13 /* All identifiers that are not reserved words
14    and are not declared typedefs in the current block */
15 %token IDENTIFIER
16
17 /* All identifiers that are declared typedefs in the current block.
18    In some contexts, they are treated just like IDENTIFIER,
19    but they can also serve as typespecs in declarations.
   */
20 %token TYPENAME
21
22 /* Reserved words that specify storage class.
23    yylval contains an IDENTIFIER_NODE which indicates which one.
   */
24 %token SCSPEC
25
26 /* Reserved words that specify type.
27    yylval contains an IDENTIFIER_NODE which indicates which one.
   */
28 %token TYPESPEC
29
30 /* Reserved words that qualify type: "const", "volatile", or "restrict".
31    yylval contains an IDENTIFIER_NODE which indicates which one.
   */
32 %token TYPE_QUAL
33
34 /* Character or numeric constants.
35    yylval is the node for the constant.  */
36 %token CONSTANT
37
38 /* String constants in raw form.
39    yylval is a STRING_CST node.  */
40 %token STRING
41
42 /* "...", used for functions with variable arglists.  */
43 %token ELLIPSIS
44
45 /* the reserved words */
46 /* SCO include files test "ASM", so use something else. */
47 %token SIZEOF ENUM STRUCT UNION IF ELSE WHILE DO FOR SWITCH CASE DEFAULT
48 %token BREAK CONTINUE RETURN GOTO ASM_KEYWORD TYPEOF ALIGNOF
```

```
%token ATTRIBUTE EXTENSION LABEL
%token REALPART IMAGPART

/* Add precedence rules to solve dangling else s/r conflict */
%nonassoc IF
%nonassoc ELSE

/* Define the operator tokens and their precedences.
   The value is an integer because, if used, it is the tree code
   to use in the expression made from the operator.  */

%right <code> ASSIGN '='
%right <code> '?' ':'
%left <code> OROR
%left <code> ANDAND
%left <code> '|'
%left <code> '^'
%left <code> '&'
%left <code> EQCOMPARE
%left <code> ARITHCOMPARE
%left <code> LSHIFT RSHIFT
%left <code> '+' '-'
%left <code> '*' '/' '%'
%right <code> UNARY PLUSPLUS MINUSMINUS
%left HYPERUNARY
%left <code> POINTSAT '.' '(' '['

%type <code> unop

%type <ttype> identifier IDENTIFIER TYPENAME CONSTANT expr nonnull_exprlist expr
%type <ttype> expr_no_commas cast_expr unary_expr primary string STRING
%type <ttype> typed_declspecs reserved_declspecs
%type <ttype> typed_typespecs reserved_typespecquals
%type <ttype> declmods typespec typespecqual_reserved
%type <ttype> typed_declspecs_no_prefix_attr reserved_declspecs_no_prefix_attr
%type <ttype> declmods_no_prefix_attr
%type <ttype> SCSPEC TYPESPEC TYPE_QUAL nonempty_type_quals maybe_type_qual
%type <ttype> initdecls notype_initdecls initdcl notype_initdcl
%type <ttype> init maybeasm
%type <ttype> asm_operands nonnull_asm_operands asm_operand asm_clobbers
%type <ttype> maybe_attribute attributes attribute attribute_list attrib
%type <ttype> any_word

%type <ttype> compstmt

%type <ttype> declarator
```

```
%type <ttype> notype_declarator after_type_declarator
%type <ttype> parm_declarator

%type <ttype> structsp component_decl_list component_decl_list2
%type <ttype> component_decl components component_declarator
%type <ttype> enumlist enumerator
%type <ttype> struct_head union_head enum_head
%type <ttype> typename absdcl absdcl1 type_quals
%type <ttype> xexpr parms parm identifiers

%type <ttype> parmlist parmlist_1 parmlist_2
%type <ttype> parmlist_or_identifiers parmlist_or_identifiers_1
%type <ttype> identifiers_or_typenames

%type <ends_in_label> lineno_stmt_or_label lineno_stmt_or_labels stmt_or_label


%%
program: /* empty */
        | extdefs
        ;

extdefs:
        extdef
        | extdefs extdef
        ;

extdef:
        fndef
        | datadef
        | ASM_KEYWORD '(' expr ')' ';'
        | EXTENSION extdef
        ;

datadef:
          notype_initdecls ';'
        | declmods notype_initdecls ';'
        | typed_declspecs initdecls ';'
        | declmods ';'
        | typed_declspecs ';'
        | error ';'
        | error '}'
        | ';'
        ;

fndef:
```

```
140           typed_declspecs declarator
141           old_style_parm_decls compstmt_or_error
142         | typed_declspecs declarator error
143         | declmods notype_declarator
144           old_style_parm_decls compstmt_or_error
145         | declmods notype_declarator error
146         | notype_declarator old_style_parm_decls compstmt_or_error
147         | notype_declarator error
148         ;

149
150 identifier:
151         IDENTIFIER
152         | TYPENAME
153         ;
154
155 unop:     '&'
156         | '-'
157         | '+'
158         | PLUSPLUS
159         | MINUSMINUS
160         | '~'
161         | '!'
162         ;
163
164 expr:   nonnull_exprlist
165         ;
166
167 exprlist:
168           /* empty */
169         | nonnull_exprlist
170         ;
171
172 nonnull_exprlist:
173         expr_no_commas
174         | nonnull_exprlist ',' expr_no_commas
175         ;
176
177 unary_expr:
178         primary
179         | '*' cast_expr   %prec UNARY
180         | EXTENSION cast_expr     %prec UNARY
181         | unop cast_expr  %prec UNARY
182         /* Refer to the address of a label as a pointer.
    */
183         | ANDAND identifier
```

```
/* This seems to be impossible on some machines, so let's turn it off.
   You can use __builtin_next_arg to find the anonymous stack args.
        | '&' ELLIPSIS
*/
        | SIZEOF unary_expr  %prec UNARY
        | SIZEOF '(' typename ')'  %prec HYPERUNARY
        | ALIGNOF unary_expr  %prec UNARY
        | ALIGNOF '(' typename ')'  %prec HYPERUNARY
        | REALPART cast_expr %prec UNARY
        | IMAGPART cast_expr %prec UNARY
        ;

cast_expr:
        unary_expr
        | '(' typename ')' cast_expr  %prec UNARY
        | '(' typename ')' '{' initlist_maybe_comma '}'  %prec UNARY
        ;


expr_no_commas:
          cast_expr
        | expr_no_commas '+' expr_no_commas
        | expr_no_commas '-' expr_no_commas
        | expr_no_commas '*' expr_no_commas
        | expr_no_commas '/' expr_no_commas
        | expr_no_commas '%' expr_no_commas
        | expr_no_commas LSHIFT expr_no_commas
        | expr_no_commas RSHIFT expr_no_commas
        | expr_no_commas ARITHCOMPARE expr_no_commas
        | expr_no_commas EQCOMPARE expr_no_commas
        | expr_no_commas '&' expr_no_commas
        | expr_no_commas '|' expr_no_commas
        | expr_no_commas '^' expr_no_commas
        | expr_no_commas ANDAND expr_no_commas
        | expr_no_commas OROR expr_no_commas
        | expr_no_commas '?' expr ':' expr_no_commas
        | expr_no_commas '?' ':' expr_no_commas
        | expr_no_commas '=' expr_no_commas
        | expr_no_commas ASSIGN expr_no_commas
        ;

primary:
        IDENTIFIER
        | CONSTANT
        | string
        | '(' expr ')'
```

```
229         | '(' error ')'
230         | '(' compstmt ')'
231         | primary '(' exprlist ')'   %prec '.'
232         | primary '[' expr ']'   %prec '.'
233         | primary '.' identifier
234         | primary POINTSAT identifier
235         | primary PLUSPLUS
236         | primary MINUSMINUS
237         ;
238
239 /* Produces a STRING_CST with perhaps more STRING_CSTs chained onto it.
    */
240 string:
241           STRING
242         | string STRING
243         ;


244
245
246 old_style_parm_decls:
247         /* empty */
248         | datadecls
249         | datadecls ELLIPSIS
250                 /* ... is used here to indicate a varargs function.
    */
251         ;
252
253 /* The following are analogous to lineno_decl, decls and decl
254    except that they do not allow nested functions.
255    They are used for old-style parm decls.  */
256 lineno_datadecl:
257           datadecl
258         ;
259
260 datadecls:
261         lineno_datadecl
262         | errstmt
263         | datadecls lineno_datadecl
264         | lineno_datadecl errstmt
265         ;
266
267 /* We don't allow prefix attributes here because they cause reduce/reduce
268    conflicts: we can't know whether we're parsing a function decl with
269    attribute suffix, or function defn with attribute prefix on first old
270    style parm.  */
271 datadecl:
```

```
272         typed_declspecs_no_prefix_attr initdecls ';'
273         | declmods_no_prefix_attr notype_initdecls ';'
274         | typed_declspecs_no_prefix_attr ';'
275         | declmods_no_prefix_attr ';'
276         ;
277
278 /* This combination which saves a lineno before a decl
279    is the normal thing to use, rather than decl itself.
280    This is to avoid shift/reduce conflicts in contexts
281    where statement labels are allowed.  */
282 lineno_decl:
283           decl
284         ;
285
286 decls:
287         lineno_decl
288         | errstmt
289         | decls lineno_decl
290         | lineno_decl errstmt
291         ;
292
293 decl:
294         typed_declspecs initdecls ';'
295         | declmods notype_initdecls ';'
296         | typed_declspecs nested_function
297         | declmods notype_nested_function
298         | typed_declspecs ';'
299         | declmods ';'
300         | EXTENSION decl
301         ;
302
303 /* Declspecs which contain at least one type specifier or typedef name.
304    (Just `const' or `volatile' is not enough.)
305    A typedef'd name following these is taken as a name to be declared.
306    Declspecs have a non-NULL TREE_VALUE, attributes do not.
   */
307
308 typed_declspecs:
309           typespec reserved_declspecs
310         | declmods typespec reserved_declspecs
311         ;
312
313 reserved_declspecs:  /* empty */
314         | reserved_declspecs typespecqual_reserved
315         | reserved_declspecs SCSPEC
316         | reserved_declspecs attributes
```

```
317         ;
318
319 typed_declspecs_no_prefix_attr:
320           typespec reserved_declspecs_no_prefix_attr
321         | declmods_no_prefix_attr typespec reserved_declspecs_no_prefix_attr
322         ;
323
324 reserved_declspecs_no_prefix_attr:
325           /* empty */
326         | reserved_declspecs_no_prefix_attr typespecqual_reserved
327         | reserved_declspecs_no_prefix_attr SCSPEC
328         ;
329
330 /* List of just storage classes, type modifiers, and prefix attributes.
331    A declaration can start with just this, but then it cannot be used
332    to redeclare a typedef-name.
333    Declspecs have a non-NULL TREE_VALUE, attributes do not.
   */
334
335 declmods:
336           declmods_no_prefix_attr
337         | attributes
338         | declmods declmods_no_prefix_attr
339         | declmods attributes
340         ;
341
342 declmods_no_prefix_attr:
343           TYPE_QUAL
344         | SCSPEC
345         | declmods_no_prefix_attr TYPE_QUAL
346         | declmods_no_prefix_attr SCSPEC
347         ;
348
349
350 /* Used instead of declspecs where storage classes are not allowed
351    (that is, for typenames and structure components).
352    Don't accept a typedef-name if anything but a modifier precedes it.
   */
353
354 typed_typespecs:
355           typespec reserved_typespecquals
356         | nonempty_type_quals typespec reserved_typespecquals
357         ;
358
359 reserved_typespecquals:  /* empty */
360         | reserved_typespecquals typespecqual_reserved
```

```
361         ;
362
363 /* A typespec (but not a type qualifier).
364    Once we have seen one of these in a declaration,
365    if a typedef name appears then it is being redeclared.
    */
366
367 typespec: TYPESPEC
368         | structsp
369         | TYPENAME
370         | TYPEOF '(' expr ')'
371         | TYPEOF '(' typename ')'
372         ;
373
374 /* A typespec that is a reserved word, or a type qualifier.
    */
375
376 typespecqual_reserved: TYPESPEC
377         | TYPE_QUAL
378         | structsp
379         ;
380
381 initdecls:
382         initdcl
383         | initdecls ',' initdcl
384         ;
385
386 notype_initdecls:
387         notype_initdcl
388         | notype_initdecls ',' initdcl
389         ;
390
391 maybeasm:
392           /* empty */
393         | ASM_KEYWORD '(' string ')'
394         ;
395
396 initdcl:
397           declarator maybeasm maybe_attribute '=' init
398         | declarator maybeasm maybe_attribute
399         ;
400
401 notype_initdcl:
402           notype_declarator maybeasm maybe_attribute '=' init
403         | notype_declarator maybeasm maybe_attribute
404         ;
```

```
405
406 maybe_attribute:
407       /* empty */
408         | attributes
409         ;
410
411 attributes:
412       attribute
413         | attributes attribute
414         ;
415
416 attribute:
417       ATTRIBUTE '(' '(' attribute_list ')' ')'
418         ;
419
420 attribute_list:
421       attrib
422         | attribute_list ',' attrib
423         ;
424
425 attrib:
426     /* empty */
427         | any_word
428         | any_word '(' IDENTIFIER ')'
429         | any_word '(' IDENTIFIER ',' nonnull_exprlist ')'
430         | any_word '(' exprlist ')'
431         ;
432
433 /* This still leaves out most reserved keywords,
434    shouldn't we include them?  */
435
436 any_word:
437           identifier
438         | SCSPEC
439         | TYPESPEC
440         | TYPE_QUAL
441         ;

442
443 /* Initializers.  `init' is the entry point.  */
444
445 init:
446         expr_no_commas
447         | '{' initlist_maybe_comma '}'
448         | error
449         ;
```

```
450
451 /* 'initlist_maybe_comma' is the guts of an initializer in braces.
    */
452 initlist_maybe_comma:
453           /* empty */
454         | initlist1 maybecomma
455         ;
456
457 initlist1:
458           initelt
459         | initlist1 ',' initelt
460         ;
461
462 /* 'initelt' is a single element of an initializer.
463    It may use braces.  */
464 initelt:
465           designator_list '=' initval
466         | designator initval
467         | identifier ':'
468           initval
469         | initval
470         ;
471
472 initval:
473           '{' initlist_maybe_comma '}'
474         | expr_no_commas
475         | error
476         ;
477
478 designator_list:
479           designator
480         | designator_list designator
481         ;
482
483 designator:
484           '.' identifier
485         /* These are for labeled elements.  The syntax for an array element
486            initializer conflicts with the syntax for an Objective-C message,
487            so don't include these productions in the Objective-C grammar.
    */
488         | '[' expr_no_commas ELLIPSIS expr_no_commas ']'
489         | '[' expr_no_commas ']'
490         ;
491
492 nested_function:
```

```
493           declarator old_style_parm_decls
494 /* This used to use compstmt_or_error.
495    That caused a bug with input `f(g) int g {}',
496    where the use of YYERROR1 above caused an error
497    which then was handled by compstmt_or_error.
498    There followed a repeated execution of that same rule,
499    which called YYERROR1 again, and so on.  */
500           compstmt
501         ;
502
503 notype_nested_function:
504           notype_declarator old_style_parm_decls compstmt
505         ;
506
507 /* Any kind of declarator (thus, all declarators allowed
508    after an explicit typespec).  */
509
510 declarator:
511           after_type_declarator
512         | notype_declarator
513         ;
514
515 /* A declarator that is allowed only after an explicit typespec.
   */
516
517 after_type_declarator:
518           '(' after_type_declarator ')'
519         | after_type_declarator '(' parmlist_or_identifiers
   %prec '.'
520         | after_type_declarator '[' expr ']'  %prec '.'
521         | after_type_declarator '[' ']'  %prec '.'
522         | '*' type_quals after_type_declarator  %prec UNARY
523         | attributes after_type_declarator
524         | TYPENAME
525         ;
526
527 /* Kinds of declarator that can appear in a parameter list
528    in addition to notype_declarator.  This is like after_type_declarator
529    but does not allow a typedef name in parentheses as an identifier
530    (because it would conflict with a function with that typedef as arg).
   */
531
532 parm_declarator:
533           parm_declarator '(' parmlist_or_identifiers  %prec '.'
534         | parm_declarator '[' '*' ']'  %prec '.'
535         | parm_declarator '[' expr ']'  %prec '.'
```

```
536         | parm_declarator '[' ']'  %prec '.'
537         | '*' type_quals parm_declarator  %prec UNARY
538         | attributes parm_declarator
539         | TYPENAME
540         ;
541
542 /* A declarator allowed whether or not there has been
543    an explicit typespec.  These cannot redeclare a typedef-name.
    */
544
545 notype_declarator:
546           notype_declarator '(' parmlist_or_identifiers  %prec '.'
547         | '(' notype_declarator ')'
548         | '*' type_quals notype_declarator  %prec UNARY
549         | notype_declarator '[' '*' ']'  %prec '.'
550         | notype_declarator '[' expr ']'  %prec '.'
551         | notype_declarator '[' ']'  %prec '.'
552         | attributes notype_declarator
553         | IDENTIFIER
554         ;
555
556 struct_head:
557           STRUCT
558         | STRUCT attributes
559         ;
560
561 union_head:
562           UNION
563         | UNION attributes
564         ;
565
566 enum_head:
567           ENUM
568         | ENUM attributes
569         ;
570
571 structsp:
572           struct_head identifier '{' component_decl_list '}' maybe_attribute
573         | struct_head '{' component_decl_list '}' maybe_attribute
574         | struct_head identifier
575         | union_head identifier '{' component_decl_list '}' maybe_attribute
576         | union_head '{' component_decl_list '}' maybe_attribute
577         | union_head identifier
578         | enum_head identifier '{' enumlist maybecomma_warn '}' maybe_attribute
579         | enum_head '{'  enumlist maybecomma_warn '}' maybe_attribute
580         | enum_head identifier
```

```
581         ;
582
583 maybecomma:
584           /* empty */
585         | ','
586         ;
587
588 maybecomma_warn:
589           /* empty */
590         | ','
591         ;
592
593 component_decl_list:
594           component_decl_list2
595         | component_decl_list2 component_decl
596         ;
597
598 component_decl_list2:   /* empty */
599         | component_decl_list2 component_decl ';'
600         | component_decl_list2 ';'
601         ;
602
603 /* There is a shift-reduce conflict here, because `components' may
604    start with a `typename'.  It happens that shifting (the default resolution)
605    does the right thing, because it treats the `typename' as part of
606    a `typed_typespecs'.
607
608    It is possible that this same technique would allow the distinction
609    between `notype_initdecls' and `initdecls' to be eliminated.
610    But I am being cautious and not trying it.  */
611
612 component_decl:
613           typed_typespecs components
614         | typed_typespecs
615         | nonempty_type_quals components
616         | nonempty_type_quals
617         | error
618         | EXTENSION component_decl
619         ;
620
621 components:
622           component_declarator
623         | components ',' component_declarator
624         ;
625
626 component_declarator:
```

```
627           declarator maybe_attribute
628         | declarator ':' expr_no_commas maybe_attribute
629         | ':' expr_no_commas maybe_attribute
630         ;
631
632 /* We chain the enumerators in reverse order.
633    They are put in forward order where enumlist is used.
634    (The order used to be significant, but no longer is so.
635    However, we still maintain the order, just to be clean.)
    */
636
637 enumlist:
638           enumerator
639         | enumlist ',' enumerator
640         | error
641         ;
642
643 enumerator:
644           identifier
645         | identifier '=' expr_no_commas
646         ;
647
648 typename:
649         typed_typespecs absdcl
650         | nonempty_type_quals absdcl
651         ;
652
653 absdcl:   /* an absolute declarator */
654         /* empty */
655         | absdcl1
656         ;
657
658 nonempty_type_quals:
659           TYPE_QUAL
660         | nonempty_type_quals TYPE_QUAL
661         ;
662
663 type_quals:
664           /* empty */
665         | type_quals TYPE_QUAL
666         ;
667
668 absdcl1:  /* a nonempty absolute declarator */
669           '(' absdcl1 ')'
670                              /* `(typedef)1' is `int'.  */
671         | '*' type_quals absdcl1  %prec UNARY
```

```
672         | '*' type_quals  %prec UNARY
673         | absdcl1 '(' parmlist  %prec '.'
674         | absdcl1 '[' expr ']'  %prec '.'
675         | absdcl1 '[' ']'  %prec '.'
676         | '(' parmlist  %prec '.'
677         | '[' expr ']'  %prec '.'
678         | '[' ']'  %prec '.'
679         /* ??? It appears we have to support attributes here, however
680            using prefix_attributes is wrong.  */
681         | attributes absdcl1
682         ;
683
684 /* at least one statement, the first of which parses without error.
    */
685 /* stmts is used only after decls, so an invalid first statement
686    is actually regarded as an invalid decl and part of the decls.
    */
687
688 stmts:
689         stmt_or_labels
690         ;
691
692 stmt_or_labels:
693           stmt_or_label
694         | stmt_or_labels stmt_or_label
695         | stmt_or_labels errstmt
696         ;
697
698 xstmts:
699         /* empty */
700         | stmts
701         ;
702
703 errstmt:  error ';'
704         ;
705
706 /* Read zero or more forward-declarations for labels
707    that nested functions can jump to.  */
708 maybe_label_decls:
709           /* empty */
710         | label_decls
711         ;
712
713 label_decls:
714           label_decl
```

```
715         | label_decls label_decl
716         ;
717
718 label_decl:
719           LABEL identifiers_or_typenames ';'
720         ;
721
722 /* This is the body of a function definition.
723    It causes syntax errors to ignore to the next openbrace.
   */
724 compstmt_or_error:
725           compstmt
726         | error compstmt
727         ;
728
729 compstmt: '{ '}'
730         | '{ maybe_label_decls decls xstmts '}'
731         | '{ maybe_label_decls error '}'
732         | '{ maybe_label_decls stmts '}'
733         ;

734
735 /* Value is number of statements counted as of the closeparen.
   */
736 simple_if:
737           if_prefix lineno_labeled_stmt
738 /* Make sure c_expand_end_cond is run once
739    for each call to c_expand_start_cond.
740    Otherwise a crash is likely.  */
741         | if_prefix error
742         ;
743
744 if_prefix:
745           IF '(' expr ')'
746         ;
747
748 /* This is a subroutine of stmt.
749    It is used twice, once for valid DO statements
750    and once for catching errors in parsing the end test.
   */
751 do_stmt_start:
752           DO lineno_labeled_stmt WHILE
753         ;
754
755 lineno_labeled_stmt:
756           stmt
```

```
757         | label lineno_labeled_stmt
758         ;
759
760 stmt_or_label:
761           stmt
762         | label
763         ;
764
765 /* Parse a single real statement, not including any labels.
    */
766 stmt:
767           compstmt
768         | expr ';'
769         | simple_if ELSE lineno_labeled_stmt
770         | simple_if                     %prec IF
771         | simple_if ELSE error
772         | WHILE '(' expr ')' lineno_labeled_stmt
773         | do_stmt_start '(' expr ')' ';'
774 /* This rule is needed to make sure we end every loop we start.
    */
775         | do_stmt_start error
776         | FOR '(' xexpr ';' xexpr ';' xexpr ')' lineno_labeled_stmt
777         | SWITCH '(' expr ')' lineno_labeled_stmt
778         | BREAK ';'
779         | CONTINUE ';'
780         | RETURN ';'
781         | RETURN expr ';'
782         | ASM_KEYWORD maybe_type_qual '(' expr ')' ';'
783         /* This is the case with just output operands.  */
784         | ASM_KEYWORD maybe_type_qual '(' expr ':' asm_operands ')' ';'
785         /* This is the case with input operands as well.
    */
786         | ASM_KEYWORD maybe_type_qual '(' expr ':' asm_operands ':' asm_operands
787         /* This is the case with clobbered registers as well.
    */
788         | ASM_KEYWORD maybe_type_qual '(' expr ':' asm_operands ':'
789           asm_operands ':' asm_clobbers ')' ';'
790         | GOTO identifier ';'
791         | GOTO '*' expr ';'
792         | ';'
793         ;

794
795 /* Any kind of label, including jump labels and case labels.
796    ANSI C accepts labels only before statements, but we allow them
797    also at the end of a compound statement.  */
```

```
798
799 label:    CASE expr_no_commas ':'
800         | CASE expr_no_commas ELLIPSIS expr_no_commas ':'
801         | DEFAULT ':'
802         | identifier ':' maybe_attribute
803         ;
804 /* Either a type-qualifier or nothing.  First thing in an 'asm' statement.
    */
805 maybe_type_qual:
806         /* empty */
807         | TYPE_QUAL
808         ;
809
810 xexpr:
811         /* empty */
812         | expr
813         ;
814
815 /* These are the operands other than the first string and colon
816    in  asm ("addextend %2,%1": "=dm" (x), "0" (y), "g" (*x))
    */
817 asm_operands: /* empty */
818         | nonnull_asm_operands
819         ;
820
821 nonnull_asm_operands:
822           asm_operand
823         | nonnull_asm_operands ',' asm_operand
824         ;
825
826 asm_operand:
827           STRING '(' expr ')'
828         ;
829
830 asm_clobbers:
831           string
832         | asm_clobbers ',' string
833         ;


834
835 /* This is what appears inside the parens in a function declarator.
836    Its value is a list of ..._TYPE nodes.  */
837 parmlist:
838           parmlist_1
839         ;
840
```

```
parmlist_1:
          parmlist_2 ')'
        | parms ';'
          parmlist_1
        | error ')'
        ;

/* This is what appears inside the parens in a function declarator.
   Is value is represented in the format that grokdeclarator expects.
*/
parmlist_2:  /* empty */
        | ELLIPSIS
        | parms
        | parms ',' ELLIPSIS
        ;

parms:
        parm
        | parms ',' parm
        ;

/* A single parameter declaration or parameter type name,
   as found in a parmlist.  */
parm:
          typed_declspecs parm_declarator maybe_attribute
        | typed_declspecs notype_declarator maybe_attribute
        | typed_declspecs absdcl maybe_attribute
        | declmods notype_declarator maybe_attribute
        | declmods absdcl maybe_attribute
        ;

/* This is used in a function definition
   where either a parmlist or an identifier list is ok.
   Its value is a list of ..._TYPE nodes or a list of identifiers.
*/
parmlist_or_identifiers:
          parmlist_or_identifiers_1
        ;

parmlist_or_identifiers_1:
          parmlist_1
        | identifiers ')'
        ;

/* A nonempty list of identifiers.  */
identifiers:
```

```
885         IDENTIFIER
886         | identifiers ',' IDENTIFIER
887         ;
888
889 /* A nonempty list of identifiers, including typenames.  */
890 identifiers_or_typenames:
891         identifier
892         | identifiers_or_typenames ',' identifier
893         ;
894 %%
```

13.4 Les sources de ppcm

13.4.1 Le fichier de déclarations ppcm.h

```
1  /* ppcm.h
2   constantes arbitraires, variables globales, prototypes de fonctions
3   */
4  # define MAXNOM 3000            /* nbre max. de noms (variables+fonctions) */
5  # define MAXEXPR 1000           /* nbre max. d'expresions par fonction */
6
7  /*
8   frame : argn ... arg1 @retour fp locale1 locale2 ... localeN ->adresses basses
9   */
10 extern int posexpr;             /* index via %ebp de la prochaine expression */
11 extern int incr;                /* la suivante ds la pile : +4 ou -4 */
12
13 /*
14  dans expr.c
15  */
16 struct expr {
17   int position;                 /* en memoire, son index via %ebp */
18   char * nom;                   /* le nom de la variable (le cas echeant) */
19 };
20 struct expr * fairexpr(char *);
21 struct expr * exprvar(char *);
22 void reinitexpr(void);
```

13.4.2 Le fichier expr.c

```
1  /* expr.c
2   gestion des expressions, de la memoire et des registres de ppcm
3   */
4  # include <stdio.h>
5  # include "ppcm.h"
```

```
6
7  struct expr expr[MAXEXPR];      /* les expressions */
8  int nexpr;                      /* le nombre d'expressions */
9
10 /* fairexpr  --  fabrique une expression (parametre, argument ou temporaire) */
11 struct expr *
12 fairexpr(char * nom){
13   register struct expr * e;
14
15   e = &expr[nexpr++];
16   e->position = posexpr;
17   e->nom = nom;
18   posexpr += incr;
19   return e;
20 }
21
22 /* exprvar  ---  renvoie l'expression qui designe la variable */
23 struct expr *
24 exprvar(char * s){
25   register struct expr * e, * f;
26
27   for(e = & expr[0], f = e + nexpr; e < f; e += 1)
28     if (/* e->nom != NULL && */ e->nom == s)
29       return e;
30   fprintf(stderr, "Erreur, variable %s introuvable\n", s);
31   return & expr[0];
32 }
33
34 /* reinitexpr  --  a la fin d'une fonction, ni expression ni registre */
35 void
36 reinitexpr(){
37   nexpr = 0;
38 }
```

13.4.3 L'analyseur lexical dans le fichier `pccm.l`

```
1   /* ppcm.l
2    analyseur lexical. representation unique des chaines de caracteres
3    */
4   char * chaine();
5   int lineno = 1;
6  %%
7  "if"                    { return YIF; }
8  "while"                 { return YWHILE; }
9  "else"                  { return YELSE; }
10 "int"                   { return YINT; }
```

```
"return"                  { return YRETURN; }
[a-zA-Z_][a-zA-Z0-9_]*    { yylval.c = chaine(yytext); return YNOM; }
'.'                       { yylval.i = yytext[1]; return YNUM; }
[0-9]+                    { yylval.i = atoi(yytext); return YNUM; }
[-+*/%=();{},]            { return yytext[0]; }
!=                        { return YNEQ; }
[ \t\f]                   ;
\n                        lineno += 1;
.                         { fprintf(stderr, "yylex : (%c)\n", yytext[0]); }
"/*"([^*]|("*"+[^/*]))*"*"+"/"   { ; /* Commentaire */ }
%%
static char * chaines[MAXNOM];
static int nchaines = 0;

/* chaine  --  renvoie une representation unique de la chaine argument */
char *
chaine(char * s){
  register char **p, **f;

  for(p = & chaines[0], f = p + nchaines; p < f; p += 1)
    if (strcmp(*p, s) == 0)
      return *p;
  if (nchaines == MAXNOM){
    fprintf(stderr, "Pas plus de %d noms\n", MAXNOM);
    exit(1);
  }
  if ((*p = (char *)malloc(strlen(s) + 1)) == NULL)
    nomem();
  strcpy(*p, s);
  nchaines += 1;
  return *p;
}

yywrap(){ return 1; }
```

13.4.4 Le grammaire et la génération de code dans le fichier ppcm.y

```
/* ppcm.y
 parseur de ppcm : le code est genere au fur et a mesure du parsing
 */
%{
# include <stdio.h>

char * chainop;                 /* assembleur pour l'operateur courant */
int posexpr, incr;              /* deplacement ds la pile pour les variables */
```

```
9  char * fonction;                /* le nom de la fonction courante */
10
11 # include "ppcm.h"
12
13 # define YYDEBUG 1
14 %}
15
16 %union {
17   int i;                        /* constantes, etiquettes et nbre d'arg. */
18   char * c;                     /* variables et fonctions */
19   struct expr *  e;             /* expressions */
20 };
21
22 %token <c> YNOM
23 %token <i> YNUM
24 %token YINT YIF YELSE YWHILE YRETURN
25
26 %type <i> ifdebut .listexpr
27 %type <e> expr
28
29 %right '='
30 %nonassoc YNEQ
31 %left '+' '-'
32 %left '*' '/' '%'
33 %left FORT
34
35 %%
36 programme : /* rien */          /* point d'entree : rien que des fonctions */
37         | programme fonction
38         ;
39
40 .listinstr : /* rien */         /* liste d'instructions */
41         | .listinstr instr
42         ;
43
44 fonction
45         : YNOM '('
46                 { posexpr = 8; incr = 4; }
47           .listarg ')' '{'
48                 { posexpr = -4; incr = -4; }
49           .listvar
50                 { printf(".text\n\t.align 16\n.globl %s\n", $1);
51                   printf("deb%s:\n", $1);
52                   fonction = $1;
53                 }
54           .listinstr '}'
```

```
55                  {                   /* epilogue */
56                    printf("fin%s:\n", $1);
57                    printf("\tmovl %%ebp,%%esp\n\tpopl %%ebp\n\tret\n");
58                                        /* proloque */
59                    printf("%s:\n", $1);
60                    printf("\tpushl %%ebp\n\tmovl %%esp,%%ebp\n");
61                    printf("\tsubl $%d,%%esp\n", -posexpr-4);
62                    printf("\tjmp deb%s\n", $1);
63
64                    reinitexpr();
65                  }
66          ;
67                                        /* liste des arguments a fournir a l'appel */
68  .listarg : /* rien */
69           |  listnom
70           ;
71                                        /* liste des variables de la fonction */
72  .listvar : /* rien */
73           | YINT  listnom ';'
74           ;
75                                       /* la liste des noms : arguments ou variables */
76  listnom : YNOM
77                  { fairexpr($1); }
78           | listnom ',' YNOM
79                  { fairexpr($3); }
80           ;
81
82  instr   : ';'                       /* Toutes les instructions */
83           | '{' .listinstr '}'
84           | expr ';'
85                  { ; }
86           | ifdebut instr
87                  { printf("else%d:\n", $1); }
88           | ifdebut instr YELSE
89                  { printf("\tjmp fin%d\n", $1);
90                    printf("else%d:\n", $1);
91                  }
92             instr
93                  { printf("fin%d:\n", $1); }
94           | YWHILE '('
95                  { printf("debut%d:\n", $<i>$ = label()); }
96             expr ')'
97                  { printf("\tcmpl $0,%d(%%ebp)\n", $4->position);
98                    printf("\tje fin%d\n", $<i>3); }
99             instr
100                 { printf("\tjmp debut%d\n", $<i>3);
```

```
101                     printf("fin%d:\n", $<i>3);
102                   }
103           | YRETURN expr ';'
104                   { printf("\tmovl %d(%%ebp),%%eax\n", $2->position);
105                     printf("\tjmp fin%s\n", fonction);
106                   }
107           ;
108
109 ifdebut : YIF '(' expr ')'
110                   { printf("\tcmpl $0,%d(%%ebp)\n", $3->position);
111                     printf("\tje else%d\n", $$ = label());
112                   }
113           ;
114                                       /* toutes les expressions */
115 expr    : YNOM
116                   { $$ = exprvar($1); }
117           | '(' expr ')'
118                   { $$ = $2; }
119           | YNUM
120                   { $$ = fairexpr(NULL);
121                     printf("\tmovl $%d,%d(%%ebp)\n", $1, $$->position);
122                   }
123           | YNOM '=' expr
124                   { printf("\tmovl %d(%%ebp),%%eax\n", $3->position);
125                     printf("\tmovl %%eax,%d(%%ebp)\n", exprvar($1)->position);
126                     $$ = $3;
127                   }
128           | '-' expr %prec FORT
129                   { printf("\tmovl %d(%%ebp),%%eax\n", $2->position);
130                     printf("\tnegl %%eax\n");
131                     $$ = fairexpr(NULL);
132                     printf("\tmovl %%eax,%d(%%ebp)\n", $$->position);
133                   }
134           | expr '-' expr
135                   { chainop = "subl";
136                   bin:
137                     printf("\tmovl %d(%%ebp),%%eax\n", $1->position);
138                     printf("\t%s\t%d(%%ebp),%%eax\n", chainop, $3->position);
139                     $$ = fairexpr(NULL);
140                     printf("\tmovl %%eax,%d(%%ebp)\n", $$->position);
141                   }
142           | expr '+' expr
143                   { chainop = "addl"; goto bin; }
144           | expr '*' expr
145                   { chainop = "imull"; goto bin; }
146           | expr YNEQ expr
```

```
147                 { chainop = "subl"; goto bin; }
148         | expr '/' expr
149                 { chainop = "%eax";
150            div: printf("\tmovl %d(%%ebp),%%eax\n\tcltd\n", $1->position);
151                  printf("\tidivl\t%d(%%ebp),%%eax\n", $3->position);
152                  $$ = fairexpr(NULL);
153                  printf("\tmovl %s,%d(%%ebp)\n", chainop, $$->position);
154                 }
155         | expr '%' expr
156                 { chainop = "%edx"; goto div; }
157         | YNOM '(' .listexpr ')'
158                 { printf("\tcall %s\n", $1);
159                   if ($3 != 0)
160                     printf("\taddl $%d,%%esp\n", $3 * 4);
161                   $$ = fairexpr(NULL);
162                   printf("\tmovl %%eax,%d(%%ebp)\n", $$->position);
163                 }
164         ;
165                                 /* liste d'expressions (appel de fonction) */
166 .listexpr : /* rien */
167                 { $$ = 0; }
168         | expr
169                 { $$ = 1; printf("\tpushl %d(%%ebp)\n", $1->position); }
170         | expr ',' .listexpr
171                 { $$ = $3 + 1; printf("\tpush %d(%%ebp)\n", $1->position); }
172         ;
173 %%
174 int
175 main(){
176   yyparse();
177   return 0;
178 }
179
180 /* label  -- renvoie un nouveau numero d'etiquette a chaque appel */
181 label(){
182   static int foo = 0;
183   return foo++;
184 }
185
186 # include "lex.yy.c"            /* yylex et sa clique */
187
188 yyerror(char * message){
189   extern int lineno;
190   extern char * yytext;
191
192   fprintf(stderr, "%d: %s at %s\n", lineno, message, yytext);
```

```
193 }
194
195 nomem(){
196   fprintf(stderr, "Pas assez de memoire\n");
197   exit(1);
198 }
```