Cours de compilation

Luc Maranget

Qu'est-ce qu'un compilateur ?

Cette question est en fait bien de nature plus pratique que théorique. Étant acquis que l'on sait déjà ce qu'est un langage de programmation et un programme.

On constate en pratique deux façons d'exécuter ses programmes. La première technique est dite *interprétation*, un programme (l'interpréteur) lit votre programme à vous et l'exécute. Cette technique tend à disparaître pour les langages d'usage général, mais elle survit bien dans des contextes plus spécialisés. Par exemple, la vaste majorité des imprimantes fabriquent les pages qu'elles impriment en interprétant un langage (le Postscript), les commandes que vous tapez en Unix sont interprétées (et exécutées) par le *shell* qui n'est rien d'autre qu'un interpréteur, on peut aussi citer *JavaScript*, interprété par les brouteurs (*browsers*) web, etc. L'autre technique consiste à compiler. En pratique, un *compilateur* lit votre programme et le transforme en un exécutable, c'est à dire quelque chose que la machine peut exécuter directement, disons une suite d'entiers que le processeur de votre machine comprend comme des instructions.

L'expérience montre l'avantage principal de la compilation sur l'interprétation : les programmes s'exécutent plus rapidement. Cela s'explique par ce que certaines opérations sont faites par le compilateur et ne seront donc plus à faire lors de l'exécution. Pour bien comprendre prenons un programme qui contient l'affectation d'un entier `123` à une variable `x`. L'interpréteur doit lire les trois caractères, les transformer en un entier machine, puis ranger cet entier dans `x`. Le compilateur va, quant à lui, lire les caractères, les transformer en un entier machine, puis produire le code qui range l'entier dans la variable `x`. Dès lors, à l'exécution le programme compilé se contente de ranger l'entier machine dans `x`, soit grosso-modo une seule instruction machine, tandis que l'interpréteur doit exécuter des dizaines, voire des centaines d'instructions pour arriver au même résultat.

Bien sûr, dans la réalité, les choses sont moins nettes, (par exemple, si l'affectation précédente est dans une boucle, un interpréteur ne lira généralement les trois caractère qu'une seule fois) mais l'idée à retenir est que le compilateur réalise par avance certaines des opérations demandées par l'exécution du programme. Cette idée va assez loin, des opérations qui semblent élémentaires telles que ≪ lire le contenu d'une variable ≫, pouvant se décomposer en des opérations effectuées à la compilation (ici, associer le nom de la variable à disons un emplacement dans la mémoire) et des opérations effectuées à l'exécution (ici, lire le contenu de l'emplacement mémoire).

De façon parfois un peu abusive, on étend le sens du mot compilateur (présenté ci-dessus comme un traducteur d'un langage de programmation vers des instructions machines) pour l'appliquer à n'importe quel traducteur. Typiquement, toutefois on s'attend à ce que le langage d'entrée, ou langage *source*, soit de plus haut-niveau que le langage de sortie, ou langage *cible*. Cette idée de haut-niveau signifie que le langage source contient des construction synthétiques, faciles à comprendre par un homme, tandis que le langage cible exprime des opération élémentaires, faciles à réaliser par une machine. Par exemple, on peut légitiment considérer qu'un traducteur qui transforme la construction de filtrage de Caml (le `match`) en cascades de tests simples (des `if`) est bien un compilateur. En poussant un peu le raisonnement dans ses retranchements, un traducteur de Pascal vers C peut être vu comme un compilateur, car Pascal contient nombre de constructions qui n'existent pas en C (par exemple, les constructions d'ensemble ou les procédures locales).

Table des matières

Chapitre 1

L'environnement des compilateurs

1.1 Qu'est-ce exactement qu'un compilateur ?

La présentation introductive des compilateurs est extrêmement simplifiée. La chaîne qui va du programme source à l'exécutable comprend un certain nombre de tâches qui regardent plus le système d'exploitation de la machine (le format exact des fichiers exécutables par exemple) que le processus de traduction du langage source vers des instructions machines. La compilation proprement dite est ce dernier processus, les autres tâches sont déléguées à des programmes spécialisés. Cette section n'entend pas décrire en détail les techniques mises en œuvre par ces outils, mais vous permettre d'en comprendre les principes.

1.1.1 Assembleur

Il est bien plus simple de définir une suite d'instructions de la machine par des symboles (par exemple de représenter une addition par « `add` » que par un code entier quelconque). Dans la pratique les programmes en langage machine se présentent sous la forme de fichiers de texte obéissant à des conventions simples. Ces fichiers sont donc écrits dans un langage particulier dit *assembleur*, qui sera lui même traduit en suite d'entiers par un programme particulier dit aussi assembleur. Cette transformation n'offre pas d'intérêt particulier, seulement des difficultés techniques qu'il est opportun de laisser régler par le fabricant de la machine ou le concepteur du système d'exploitation.

Soit, en première approximation, le compilateur transforme le langage source en assembleur, expression humainement compréhensible des instructions de la machine. Sous Unix, on peut produire un fichier assembleur en donnant une option au compilateur, généralement « `-S` ». Considérons le fichier suivant `bonjour.c` :

```
#include <stdio.h>

int main (int argc, char **argv) {
  printf("bonjour\n") ;
  return 0 ;
}
```

Sur un PC quelconque, on le compile ainsi :

```
# cc -S bonjour.c
```

Et on obtient le fichier d'assembleur suivant (quelques détails sont omis !) :

```
.LC0:
        .string "bonjour\n"
.text
        .align 4
.globl main
```

```
        .type    main,@function
main:
        subl    $24, %esp
        pushl   $.LC0
        call    printf
        addl    $28, %esp
        xorl    %eax, %eax
        ret
```

Chaque ligne de ce fichier s'interprète ainsi :

- comme une instruction du processeur (par ex. `subl $24, %esp`, qui doit être une soustraction),
- comme une directive donnée à l'assembleur (par ex. `.align 4` ou `.data`), débutant par un point,
- ou comme une étiquette, nom symbolique que l'on donne à une adresse mémoire, suffixée par « `:` ». Par exemple, l'étiquette `main` est définie comme égale au début du code de la fonction homonyme.

1.1.2 Édition de liens

Jusqu'ici j'ai prétendu que le travail de l'assembleur est de transformer le source d'assemblage en exécutable. Mais en fait j'ai menti. Il est impossible de procéder aussi directement : dans `bonjour.s` on peut remarquer l'instruction `call printf`, qui réalise l'appel de la fonction `printf`. L'instruction `call` prend en argument une adresse qui est celle du début du code de la fonction appelée ; logiquement cette adresse est présente dans le code assembleur sous forme d'un symbole.

Or la fonction `printf` ne fait pas partie de notre programme, son code est *ailleurs*. De fait cette fonction fait partie de la librairie standard de C et elle est présente sous forme compilée quelque part. L'assembleur utilisé seul ne peut traduire le symbole `printf` en une adresse. Son produit sera bien une suite d'entiers représentant des instructions, supplémentée par des indications sur les symboles utilisés (et non résolus), ainsi que sur les symboles définis (par exemple ici `main`). On appelle ce produit le *code objet* et les fichiers qui le contiennent portent généralement l'extension « `.o` ».

C'est un autre programme, dit *éditeur de liens* qui prend tous les fichiers objets et fabrique l'exécutable. L'éditeur de lien s'occupe essentiellement de mettre tous les fichiers objets les uns derrière les autres et de résoudre les références symboliques entre ces fichiers. En Unix, l'éditeur de liens est normalement le programme `ld`.

1.1.3 Chargement dynamique

Une fois encore j'ai menti par souci de simplicité.

Il y a encore deux éléments à considérer. Tout d'abord, lorsque l'on compile (sans les lier) de nombreux fichiers, on obtient logiquement de nombreux fichiers objets (les « `.o` »). C'est en particulier le cas pour la librairie standard. Il n'est pas très pratique de manipuler tous ces fichiers, on les regroupe donc dans une *bibliothèque* (en anglais *library*), que l'on appelle parfois aussi archive. En Unix c'est la commande `ar` qui fabrique ces bibliothèques, leur suffixe est traditionnellement « `.a` » et leur nom commence généralement par « `lib` ». Ainsi, la libraire standard de C (dite aussi *libc*) est généralement un fichier `libc.a`, qui contient le code de toutes les fonctions de la librairie standard.

Mon second mensonge est plus grave, la description de l'édition de liens de la section précédente présente l'édition de liens traditionnelle, dite également statique. Cette technique présente l'inconvénient, lorsque l'on utilise une librairie (et on en utilise forcément) de copier le code des fonctions de librairie dans l'exécutable. Dès lors, tous les programmes C qui utilisent `printf` contiennent une copie du code de `printf`, une copie du code de toutes les fonctions appelées par `printf`, etc. Par conséquent les fichiers exécutables deviennent systématiquement assez gros.

Une technique plus maligne dite chargement *dynamique*, consiste à reporter le chargement en mémoire du code des bibliothèques lors de l'exécution du programme. En gros cela revient, à la compilation, à remplacer l'édition de liens statique par l'ajout de code et d'informations suffisants pour aller chercher les symboles non encore résolus. L'inconvénient majeur de cette technique est que la compilation et l'exécution doivent de dérouler dans des environnements offrant les mêmes bibliothèques, enfin pour le moins des bibliothèques compatibles. Cela complique notablement la distribution de code exécutable et s'applique *a fortiori* à l'exécution des *applets* : l'environnement hôte doit fournir une machine virtuelle et des libraires compatibles avec celles de l'environnement de compilation.

1.1.4 En résumé

Le simple appel ≪ `cc bonjour.c` ≫ invoque en fait plusieurs programmes dont un seulement est le compilateur proprement dit. On peut voir ce qui se passe en donnant l'option ≪ `-v` ≫ à `cc`. En simplifiant un peu, on a :

```
# cc -v bonjour.c
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/2.96/specs
gcc version 2.96 20000731 (Red Hat Linux 7.1 2.96-98)
 /usr/lib/gcc-lib/i386-redhat-linux/2.96/cpp0 ... bonjour.c /tmp/cc9oRNcK.i
 /usr/lib/gcc-lib/i386-redhat-linux/2.96/cc1 /tmp/cc9oRNcK.i -o /tmp/cck4wLml.s
 as -V -Qy -o /tmp/ccNUF1EX.o /tmp/cck4wLml.s
 /usr/lib/gcc-lib/i386-redhat-linux/2.96/collect2 ...
```

On distingue les appels à l'assembleur (`as`) et à l'éditeur de liens (`collect2`). Le compilateur proprement dit est `cc1`.

On notera au passage que `java` pousse très loin l'idée du chargement dynamique, la tentative de chargement d'un code objet (`.class` en Java) plus vieux que son source (d'extension `.java`) allant jusqu'à provoquer une recompilation. Enfin, Caml, le langage de ce cours, suit plus ou moins le principe simple de l'édition de liens statique. Les fichiers objets portant cette fois l'extension `.cmo`.

Au passage encore, le fait que tant `javac` que `ocamlc` produisent du bytecode et non pas du code natif ne change pas grand chose à l'affaire.

1.2 La chaîne de compilation

Nous avons donc défini un compilateur au sens de ce cours comme un traducteur d'un langage source (de programmation) vers l'assembleur. On décompose cette traduction en traductions plus élémentaires. Tout ceci est idéalement représenté par un schéma (cf. figure 1.1). Dans ce dessin, les flèches correspondent à des transformations et les boites à des résultats. La colonne de gauche correspond à la partie avant du compilateur (*front-end*) et la colonne de droite à la partie arrière (*back-end*).

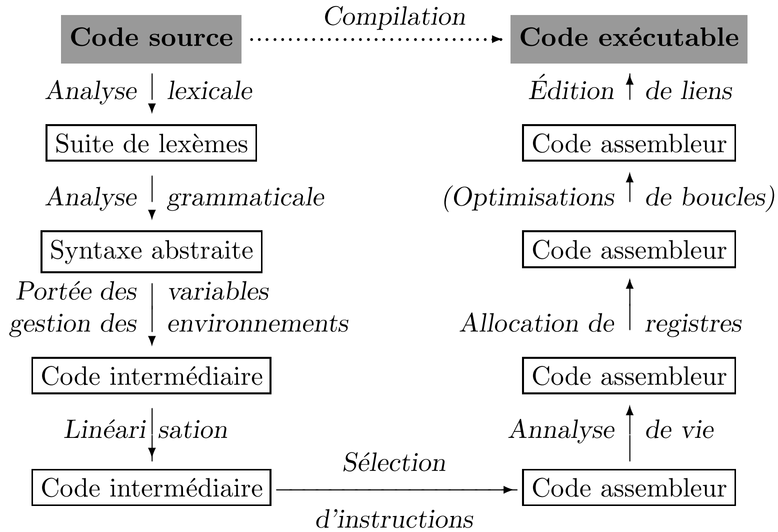
En première approximation ces deux parties sont indépendantes, le front-end dépendant du langage de programmation et le back-end de la machine ciblée.

Dans la pratique de ce cours, la majeure partie du front-end va dépendre du langage source, car il implémente réellement la sémantique du langage compilé. Tandis que la plus grande partie du back-end peut sera réalisée dans un style générique, les sous-parties clairement dépendantes de la machine cible étant bien isolées.

1.3 Gestion des compilation et recompilations

Pour des raisons diverses, on en vient rapidement à répartir le source d'un programme en plusieurs fichiers. Une des plus convaincantes de ces raisons découle du principe d'abstraction. On construit un programme compliqué par l'assemblage de briques individuellement simples. C'est précisément ce que nous allons faire dans ce cours en écrivant un compilateur, dont les briques de bases (ou *phases*) seront reparties dans divers fichiers source. C'est pourquoi je prendrai l'exemple d'un gros programme écrit en Caml, langage d'implémentation de notre cours.

Figure 1.1 – Les phases d'un compilateur



Une brique s'appelle aussi une *unité de compilation*, ou parfois un *module*, ce qui est légèrement impropre. Le principe d'abstraction conduit à séparer une brique B de programme en deux : d'une part un fichier d'implémentation (extension `.ml`) et d'autre part un fichier d'interface (extension `.mli`) qui contient les informations suffisantes pour compiler d'autres briques du programme qui utilisent B. Ces informations sont essentiellement les noms des fonctions de B utilisables de l'extérieur (dites fonctions *exportées*), leurs types, et des commentaires qui disent comment les utiliser. Pour des raisons d'efficacité les fichiers `.mli` sont compilés en des fichiers objets bien particuliers qui portent l'extension `.cmi`. Il en résulte une première dépendance : la compilation du fichier *nom*`.ml` étant l'occasion de vérifier que le module *Nom* définit bien ce qu'il affirme exporter dans *nom*`.mli`, il importe de compiler l'interface avant l'implémentation.

La séparation des unités de compilation en implémentation et interface mène à la propriété de *compilation séparée* : pour compiler `b.ml` qui utilise des fonctions de `a.ml` il n'est pas nécessaire d'avoir compilé `a.ml`, la compilation préalable de `a.mli` suffit. Si l'on se place maintenant dans le cadre d'un développement d'un programme `zyva`, dont le source est réparti en trois unités de compilation, `A`, `B` et `Zyva`. Supposons en outre que `Zyva` utilise des fonctions de `A` et de `B`, tandis que `B` utilise des fonctions de `A`. Pour fabrique le programme `zyva`, on pourra donc enchaîner les commandes suivantes :

`# ocamlc a.mli`	*produire* `a.cmi`
`# ocamlc -c a.ml`	*l'option* `-c` *évite l'édition de liens, production de* `a.cmo`
`# ocamlc b.mli`	
`# ocamlc -c b.ml`	
`# ocamlc -c zyva.ml`	
`# ocamlc -o zyva a.cmo b.cmo zyva.cmo`	*édition de liens*

(On aurait pu tout aussi bien compiler `a.mli` et `b.mli` en premiers.)

Mieux, si nous modifions `a.ml` et seulement lui, il suffit pour recréer `zyva` de procéder ainsi :

`# ocamlc -c a.ml`	*inclus la vérification de la nouvelle implémentation par rapport à l'interface*
`# ocamlc -o zyva a.cmo b.cmo zyva.cmo`	*édition de liens*

Dans la pratique il hors de question de gérer compilation et recompilations soi-même, on dispose pour ce faire d'un outil : `make`.

Je présente maintenant quelques principes, qui devraient suffire pour comprendre les exemples donnés en TP. Dans le principe `make` est un outil simple : à l'aide d'une description des règles de production de fichiers et de leur dépendances, contenue dans un fichier de nom `Makefile`, l'outil `make` invoque les commandes nécessaires à la production d'un fichier passé en argument. Soit ici, si on a écrit le `Makefile` idoine, la commande « `make zyva` » reconstruira `zyva` pour nous. Pour ce faire il analyse le graphe des dépendances entre fichiers et invoque les commandes nécessaires (typiquement des appels au compilateur), il s'agit là d'un bête tri topologique que vous connaissez déjà. Dans le cadre de notre exemple, il nous faut donc à la fois définir les règles de production et les dépendances. Voici quelques explications sur ce qu'on peut mettre dans le `Makefile`.

— Le cas de l'edition de liens est assez simple :

  ```
  zyva: a.cmo b.cmo zyva.cmo
          ocamlc -o zyva a.cmo b.cmo zyva.cmo
  ```

  La première ligne décrit les dépendances, la seconde indique comment fabriquer `zyva`. Attention, la seconde ligne doit obligatoirement commencer par une tabulation.

— Pour les diverses compilations, les règles de production s'expriment par des règles dites implicites, dont voici la syntaxe :

  ```
  .ml.cmo:
           ocamlc -c $<
  .mli.cmi:
           ocamlc  $<
  ```

  C'est à lire comme : en cas de besoin d'un fichier *nom*`.cmo` dépend de et se construit avec un fichier *nom*`.ml` (qui doit donc exister ou pouvoir être construit), avec la commande `ocamlc -c` *nom*`.ml`. La variable spéciale `$<` représente le nom du source. Les autres dépendances sont écrites à part, ici on aura :

  ```
  a.cmo: a.cmi
  b.cmo: a.cmi b.cmi
  zyva.cmo: a.cmi b.cmi
  ```

  À lire par exemple comme `a.cmo` doit être refait après `a.cmi`.

On dispose d'un outil spécifique pour produire ces dépendances, `ocamldep`, qui prend en argument les noms des fichiers sources. Généralement, on met les dépendances dans un fichier nommé `.depend`, fichier qui est inclus dans le `Makefile` ainsi :

```
include .depend
```

Il est pratique d'appeler `ocamldep` à l'aide de `make`, on utilise donc une cible `depend` correspondant à la règle suivante :

```
depend:
        ocamldep *.mli *.ml > .depend
```

On construira (et reconstruira) les dépendances par « `make depend` ». Notons que l'on ne peut pas appeler la cible `.depend`, car alors cette cible ne dépendrait de rien et elle serait toujours à jour dès qu'elle existe.

Chapitre 2

Code machine

2.1 Les processeurs

On entend par code machine le langage du processeur de l'ordinateur. Le modèle d'un processeur est toujours sensiblement le même, il correspond grosso-modo au modèle de Van Neuman, modèle fondateur de l'ordinateur concret.

Selon ce modèle, l'ordinateur est composé en première approximation d'un processeur d'un banc de registres et d'une mémoire, la mémoire est un grand tableau d'entiers, dit aussi *mots* mémoire. Les registres sont une petite quantité de mémoire rapidement accessible. Le processeur lit une instruction *à partir de la mémoire*, l'exécute, lit une autre instruction l'exécute etc.

Les instructions du processeur sont élémentaires, ce peuvent être des lectures ou des écritures en mémoire, des opérations arithmétiques simples entières ou flottantes, des sauts (qui font que l'instruction exécutée ensuite ne suit pas l'instruction de saut dans la mémoire). Elle sont simples parce que réalisées par des circuits électroniques. L'innovation de Van Neuman est que le programme réside dans la mémoire et que le processeur l'interprète en quelque sorte. Une machine connue précédente (l'ENIAC) n'était pas un ordinateur au sens moderne, mais plutôt une grosse calculatrice : il n'existait pas à proprement parler de programme, pour calculer un résultat spécifique, il fallait changer les câblages entre les diverses unités de calcul. Par contraste la machine de Van Neuman est un calculateur dont la tâche est d'interpréter des programmes, on parle aussi de machine universelle.

En particulier, et c'est cela qui nous intéresse dans notre cours les programmes sont des données résidant en mémoire, ils peuvent être lus ou produits par d'autre programmes. En ce sens, la machine de Van Neuman ouvre la porte de la compilation.

On comprendra donc que les processeurs se ressemblent tous du point de vue de l'utilisateur, puisqu'ils se conforment tous au modèle initial. Les différences entre processeurs proviennent surtout du jeu d'instructions. On distingue :

- Les (vieux) processeurs CISC (*Complex Instruction Set*).
  - Leurs instructions sont complexes, de tailles variables, Beaucoup réalisent des transferts avec la mémoire ; ils possèdent en général peu de registres. Une opération CISC typique est l'addition d'un registre et d'une case de la mémoire, le résultat étant rangé dans la mémoire, ou une instruction spécialisée dans le transfert des zones de mémoire.
  - Ce type de conception correspond d'abord à l'idée d'une programmation directe de la machine, le programmeur apprécie alors les instructions synthétiques qui lui permettent de faire réaliser des opérations compliquées plus rapidement que par une suite d'instructions. Notons que cette attitude perdure, par exemple les instructions MMX réalisent directement en machine des opérations flottantes sur de petits vecteurs, afin d'accélérer les applications de traitement du signal (*i.e.* image et son).
  - Toutefois, les compilateurs ont du mal à bien utiliser ces instructions. Plus grave, leur présence complique notablement le décodage des instructions par le processeur. En particulier, le format des instructions est peu uniforme : les instructions peuvent occuper un nombre variable de mots.

    — Il s'agit plutôt de processeurs un peu anciens, conçus avant 1985. Typiquement : Intel 8086 et Motorola 68000.
— Les (nouveaux) processeurs RISC (*Reduced Instruction Set*)
    — Le jeu d'instruction est réduit et très régulier. Les registres sont nombreux (typiquement 32). En radicalisant, seules deux instructions lisent et écrivent la mémoire (lecture dans un registre, écriture d'un registre). Toutes les autres opèrent entre registres, toujours selon le même schéma.
    — La simplicité du jeu d'instruction autorise des logiques de décodage simples et surtout plus facilement réalisables en parallèle, selon le principe du tuyau (*pipe*). Par exemple, pendant qu'une addition s'exécute dans l'unité de calcul du processeur, l'unité de de décodage des instructions peut être en train de s'occuper de l'instruction suivante, tandis que l'unité de chargement des instructions peu être en train de lire l'instruction suivant l'instruction suivante. En pratique pour une tâche donnée, on peut s'attendre à ce qu'une machine RISC se montre plus rapide qu'une machine CISC, en raison de ce parallélisme interne. Notons aussi que, lors de l'apparition des RISC, la simplicité des processeurs entraînait une conception rapide et donc la mise à disposition du public des technologies de fabrication des circuits les plus récentes (et les plus efficaces).
    — Un compilateur se débrouille bien avec un jeu d'instruction peu étendu et très régulier. Il sait exploiter des registres nombreux. D'autre part, l'exécution en parallèle des instructions impose des contraintes supplémentaires (par ex, le résultat d'une opération n'est pas disponible pour l'instruction suivante) difficiles (mais aussi inhabituelles) pour un humain.
    — Conçus après 1985. Typiquement : Alpha, Sparc et Mips. G4.

Il faut tout de même reconnaître que le fossé entre RISC et CISC n'est pas aussi grand que l'on le pensait dans les années 80. Ici tout est affaire de degré. Les premiers processeurs RISC ne possédaient pas de multiplications câblée (sous prétexte que la multiplication est rare en pratique !), ce n'est plus le cas. À l'inverse, le Pentium (héritier du 8086) a un jeu d'instructions très varié, mais ses diverses incarnations exécutent bien les instructions en parallèle, ce qui était présenté comme propre aux purs RISC à l'origine.

2.1.1 Un peu de culture : le bytecode

On ne saurait passer sous silence le *bytecode* : le compilateur ne produit pas de code pour un processeur réel, mais pour un processeur conventionnel, une machine virtuelle. Les instructions, sont bien représentées par une suite d'entiers, mais c'est un programme qui les lira et les interprétera. Ce dernier programme est bien évidemment écrit dans un langage d'assez bas niveau, typiquement en C. L'avantage de cette technique est la portabilité, pour obtenir un système fonctionnant sur une nouvelle architecture, il n'y a pas besoin de modifier le compilateur, il suffit *a priori* de porter le programme qui implémente la machine virtuelle. On peut aller jusqu'à considérer que la portabilité s'applique aussi aux programmes compilés : ≪ *Compile once, run everywhere* ≫ comme on dit pour Java. C'est un peu exagéré en pratique, car un environnement d'exécution ne se compose pas, dans le cas de Java, seulement d'un processeur, mais aussi de nombreuses fonctions de librairie chargées dynamiquement qui doivent alors se trouver à la fois sur le lieu de compilation et sur celui de l'exécution. On comprend cependant bien le principe, qui autorise les *applets* de Java.

Évidemment, l'exécution de bytecode est plus lente que l'exécution directe de code du processeur, dit aussi code *natif*, car il y a, entre autres, un surcoût dû à la mécanique d'interprétation des instructions.

Des exemples de cette technique sont le système de Java (compilateur `javac`, machine virtuelle `java`) et le système Objective Caml (compilateur `ocamlc`, machine virtuelle `ocamlrun`). Notons que certains compilateurs Java produisent du code natif, tandis que Caml propose un compilateur natif (`ocamlopt`). Notons également qu'il est possible, lors, disons de la première exécution d'une fonction, de transformer le bytecode en instructions de la machine hôte, on parle alors de compilation *à la volée* (*Just In Time* ou *JIT*). Cela revient un peu à déléguer une partie de la compilation au moment de l'exécution et ne se justifie vraiment qu'en cas de chargement de code à l'exécution entre machines hétérogènes.

Dans le cas du bytecode, le concepteur du langage a le choix de la machine cible. Il va donc l'adapter au langage. C'est par exemple le cas de la machine Java qui fournit des instructions d'appel de méthode et de la machine Caml qui fournit des instructions d'appel de fermeture et des opérations arithmétiques sur les 31 bits de poids fort des entiers.

Pourtant une machine virtuelle peut à priori fournir une plate-forme d'exécution indépendante à la fois du langage (c'est bien ce que fait une machine réelle après tout) et de la machine réelle. C'est un peu le sens du projet .NET de Microsoft, mais il y a loin de la coupe au lèvres, le modèle de la machine .NET étant spécifiquement objet, et bien plus complexe qu'une machine réelle. Des développement sont d'ailleurs en cours dans le sens de l'extension de la machine machine .NET pour s'adapter aux langages fonctionnels.

Une référence intéressante sur la conception d'une machine virtuelle (celle de Caml-Light) est le rapport ZINC [1].

2.2 Description d'un processeur

Dans ce cours nous considérerons un processeur RISC particulier : le MIPS, parce qu'il est simple et exemplaire des processeurs modernes, mais aussi parce que nous disposons d'un simulateur de ce processeur.

Le simulateur SPIM est disponible en `http://www.cs.wisc.edu/~larus/spim.html`. Voici des liens sur le manuel en HTML [2] et en Postscript [3]

2.2.1 La mémoire

Tous les processeurs modernes comportent une unité mémoire (MMU) qui permet de manipuler des adresses virtuelles, *i.e.* de faire un renommage, transparent pour l'utilisateur, entre les adresses virtuelles du programme et les adresses réelles en mémoire.

Cela permet à chaque programme de choisir ses adresses indépendamment des autres programmes (qui peuvent être exécutés en même temps sur la même machine avec les mêmes adresses virtuelles mais des adresses réelles différentes).

Du point de vue de l'utilisateur, la mémoire est un (grand) tableau dont les indices sont les adresses. Généralement, la plus petite unité adressable dans la mémoire est l'octet (8 bits) ou *byte*. Mais la taille naturelle des entiers manipulés par le processeur, c'est à dire la taille des entiers contenus dans les registres, mais aussi la taille des adresses, est plus grande, typiquement 32 ou 64 bits (soit 4 ou 8 octets). On notera donc que sur un processeur 32 bits, les adresses des mots mémoires successifs sont croissantes de 4 en 4. Les accès à la mémoire non-*alignés*, c'est à dire ceux qui ne correspondent pas à des adresses multiples de la taille en octets de la valeur accédée, sont soit interdits soit pénalisés. Par exemple, pour le MIPS, l'instruction générique de lecture d'un mot en mémoire `lw` exige des adresses multiples de 4.

La mémoire (virtuelle) d'un programme est partagée en zones. Il s'agit là, plus que d'une convention, d'un principe du système d'exploitation (ici Unix), organisateur de l'exécution des programmes.

1. `http://pauillac.inria.fr/~xleroy/publi/ZINC.ps.gz`
2. `http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/compil/spim-manual/index.html`
3. `http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/compil/spim-manual/spim.ps`

Des adresses hautes vers les adresses basses :

Adresses hautes	Stack
	$\downarrow$
	$\uparrow$
	Données allouées dynamiquement
	Données statiques modifiables
	Texte (programme) non écrivable
Adresses basses	Réservé au système

On distingue donc (du haut vers le bas) :

— La pile (*Stack*). Il s'agit d'une zone mémoire utilisées par les fonctions du programme entre autres pour leurs variables locales.

— Un trou qui va de la fin de la pile au début de zone suivante. Ce trou est énorme et ne correspond à aucune case mémoire valide. Si l'on tente d'y accéder, le système d'exploitation déclenchera une erreur. Toutefois, l'accès illégal peut être provoqué par un accès un peu en deçà de la limite basse de la zone allouée initialement pour la pile. Le système d'exploitation pourra alors réagir en augmentant la zone mémoire dédiée à la pile et ne pas faire échouer le programme. Ainsi l'espace de la pile peut croître dynamiquement en fonction des besoins du programme.

— Les données allouées dynamiquement par le programme. Cette zone est étendue explicitement (vers le haut cette fois) par le programme, cf. l'allocation explicite `new` en Pascal et Java, `malloc` en C, allocation implicite de Caml.

— Les données allouées statiquement par le programme. C'est le compilateur qui alloue cette zone car contrairement à la précédente, sa taille est connue lors de la compilation. Typiquement on y trouvera les variables globales du programme.

— Le texte, c'est à dire le code du programme. On ne peut pas écrire dans cette zone. Essentiellement, cela accélère la lecture en mémoire des instructions à travers un cache, qui est une mémoire d'accès rapide contenant une copie d'une partie de la « vraie » mémoire. En effet, le contenu d'une case du cache ne peut alors jamais différer du contenu de la case mémoire cachée.

Le simulateur SPIM, va émuler cette vision de la mémoire (dans une zone par lui allouée). Dans une machine sans mémoire virtuelle on a généralement une organisation similaire, mais sans la protection contre l'écriture et la lecture dans les zones interdites.

2.2.2 Les registres

Le MIPS comporte 32 registres généraux interchangeables, sauf

— le registre `0` qui vaut toujours zéro, même après une écriture.

— Le registre `31`, utilisé implicitement par certaines instructions pour sauver l'adresse de retour avant un saut.

Les autres registres portent les numéros restants de `1` à `30`. Comme cela n'est ni beau ni pratique, on leur donne des noms conventionnels. Ces noms correspondent à des utilisations préférentielles, qui seront détaillées par la suite.

Nom	Numéro	Usage
zero	0	Zéro (toujours)
at	1	Réservé par l'assembleur
v0 .. v1	2 .. 3	Retour de valeurs
a0 .. a3	4 .. 7	Passage d'arguments
t0 .. t7	8 .. 15	Temporaires non sauvegardés
s0.. s7	16 .. 23	Temporaires sauvegardés
t8.. t9	24 .. 25	Temporaires non sauvegardés
k0.. k1	26 .. 27	Réservés par le système
gp	28	Global Pointer
sp	29	Stack Pointer
fp	30	Frame Pointeur
ra	31	Return Address

Enfin et c'est assez important, il existe des registres spécifiques au processeur. Le principal est le compteur ordinal (*program counter*), noté `pc`. Le processeur incrémente le registre `pc` après la lecture d'une instruction et les instructions de saut écrivent dedans.

Le processeur Pentium ne possède que que huit registres d'usage général, dont les noms conventionnels sont du genre `eax`, `ebx`, `ecx`...

On peut voir les registres comme un tout petit peu de mémoire, très rapidement accessible. La bonne exploitation des registres compte pour beaucoup dans la rapidité d'un programme, car l'accès à une case de mémoire est bien plus coûteuse que l'accès à un registre. L'évolution en cours des processeurs et des mémoires ne fait que renforcer ce décalage, et la multiplications de caches toujours plus grands ne le résout que partiellement.

2.2.3 Le jeu d'instructions

Il convient d'abord d'examiner les *mode d'adressage* c'est à dire l'expression des arguments des instructions. On utilise parfois le vocabulaire suivant :

Immédiat	un entier
Direct	le contenu du registre
Indirect	le contenu de l'adresse contenu dans un registre
Indirect indexé	le contenu de l'adresse contenue dans le registre augmenté d'un déplacement

Mais dans la description d'un processeur, il vaut mieux définir des symboles précis.

n	une constante entière
ℓ	une étiquette (adresse)

r	nom de registre
a	absolu (n ou ℓ)
o	opérande (r ou a)

La plupart des instructions suivent le modèle

— **add** r_1, r_2, o qui place dans r_1 la valeur $r_2 + o$.

Les instructions qui interagissent avec la mémoire sont uniquement les instructions `load` et `store`.

— **lw** r_1, $n(r_2)$ place dans r_1 le mot contenu à l'adresse $r_2 + n$.

— **sw** r_1, $n(r_2)$ place r_1 dans le mot contenu à l'adresse $r_2 + n$.

Les instructions de contrôle conditionnel ou inconditionnel :

— **bne** r, a, ℓ saute à l'adresse ℓ si r et a sont différents,

— **jal** o qui sauve $pc + 1$ dans `ra` et saute à l'étiquette o.

Voici la liste des principales instructions :

Syntaxe	Effet
move r_1, r_2	$r_1 \leftarrow r_2$
add r_1, r_2, o	$r_1 \leftarrow o + r_2$
sub r_1, r_2, o	$r_1 \leftarrow r_2 - o$
mul r_1, r_2, o	$r_1 \leftarrow r_2 \times o$
div r_1, r_2, o	$r_1 \leftarrow r_2 \div o$
and r_1, r_2, o	$r_1 \leftarrow r_2$ `land` o
or r_1, r_2, o	$r_1 \leftarrow r_2$ `lor` o
xor r_1, r_2, o	$r_1 \leftarrow r_2$ `lxor` o
sll r_1, r_2, o	$r_1 \leftarrow r_2$ `lsl` o
srl r_1, r_2, o	$r_1 \leftarrow r_2$ `lsr` o
li r_1, n	$r_1 \leftarrow n$
`la` r_1, a	$r_1 \leftarrow a$

Syntaxe	Effet
lw $r_1, o(r_2)$	$r_1 \leftarrow tas.(r_2 + o)$
sw $r_1, o(r_2)$	$r_1 \rightarrow tas.(r_2 + o)$
slt r_1, r_2, o	$r_1 \leftarrow r_2 < o$
sle r_1, r_2, o	$r_1 \leftarrow r_2 \leq o$
seq r_1, r_2, o	$r_1 \leftarrow r_2 = o$
sne r_1, r_2, o	$r_1 \leftarrow r_2 \neq o$
j o	$pc \leftarrow o$
jal o	$ra \leftarrow pc + 1 \wedge pc \leftarrow o$
beq r, o, a	$pc \leftarrow a$ si $r = o$
bne r, o, a	$pc \leftarrow a$ si $r \neq o$
syscall	appel système
nop	ne fait rien

L'aspect RISC est donc très notable. On peut remarquer par exemple le fonctionnement de l'instruction d'appel de fonction **jal** (*jump and link*). L'adresse de retour de la fonction, c'est à dire l'adresse de l'instruction qui suit le **jal**, est rangée dans le registre `ra`. Un CISC l'empilerait plutôt.

A contrario, les instructions synthétiques sont absentes, même les plus simples. Par exemple le Pentium possède une instruction d'empilage d'un registre, `pushl` r. Du point de vue RISC cette instruction diminue le pointeur de pile de la taille d'un mot et range le registre r à l'adresse pointée par le pointeur de pile. Soit en MIPS :

```
sub   $sp, $sp, 4
sw    r,0($sp)
```

Réaliser ces deux opérations en une seule instruction demande certainement de consacrer des transistors du processeurs et des bits de la définition du format des instruction à ce cas particulier. En conséquence de quoi, la logique de décodage des instructions va se compliquer. Un gain est cependant possible si le compilateur sait exploiter les instructions synthétiques à bon escient et si le processeur exécute les instructions fréquemment utilisées plus rapidement que leur décomposition en instructions plus simples.

2.2.4 Les appels systèmes

Ils permettent l'interaction avec le système d'exploitation, et en dépendent. Le numéro de l'appel système est lu dans `v0` (attention, ce n'est pas la convention standard). Selon l'appel, un argument supplémentaire peut être passé dans `a0`.

Le simulateur SPIM implémente les appels suivants :

Nom	N^o	Effet
`print_int`	1	imprime l'entier contenu dans `a0`
`print_string`	4	imprime la chaîne en `a0` jusqu'à `'\000'`
`read_int`	5	lit un entier et le place dans `v0`
`sbrk`	9	alloue `a0` octets dans le tas, retourne l'adresse du début dans `v0`.
`exit`	10	arrêt du programme en cours d'exécution

2.3 Langage assembleur et langage machine

Le langage assembleur (ou d'assemblage, beurk) est un langage symbolique qui donne des noms aux instructions (plus lisibles que des suites de bits). Il permet aussi l'utilisation d'étiquettes symboliques et de pseudo-instructions et de modes d'adressage surchargés.

Le langage machine est une suite d'instructions codées sur des mots (de 32 bits pour le MIPS). L'assembleur transforme ces instructions en instructions de la machine. Les étiquettes sont donc résolues (quand c'est possible !) et les pseudo-instructions remplacées par une ou plusieurs instructions machine.

L'assemblage est la traduction du langage d'assembleur en langage machine. Le résultat est un fichier objet qui contient, en plus du code, des informations de relocation qui permettent de lier *(linker)* le code de plusieurs programmes ensemble. Le programme final est donc un fichier dont la structure est à l'image de la description donnée pour la mémoire précédemment. Il restera à charger ce programme en mémoire et à le lancer, c'est le système d'exploitation qui s'en charge, lorsque l'utilisateur demande l'exécution du programme. L'adresse de lancement est contenue dans l'exécutable, ou a une valeur conventionnelle.

Dans le cadre de notre cours, le simulateur SPIM prend en entrée un fichier d'assembleur et réalise lui même l'assemblage, puis toutes les opération décrites jusqu'au lancement. L'édition de liens n'est pas réellement nécessaire, puisque qu'il n'y a ni fichiers multiples, ni librairies.

2.3.1 Pseudo-Instructions

La traduction du langage machine en langage assembleur est facile. Elle permet de présenter les instructions machine (mots de 32 bits) sous une forme plus lisible. Le simulateur SPIM présente les instructions machines sous cette forme.

On se rend alors compte dans le cas du MIPS que l'on ne retrouve pas toujours les instructions du fichier initial. En effet, le langage compris par l'assembleur est un peu étendu par rapport à celui du processeur. Certaines des instructions propres à l'assembleur sont de simples commodités : **move** instruction de transfert d'un registre dans un autre est en fait une addition de zéro.

D'autres pseudo-instructions se traduisent en quelques instructions, c'est le cas de l'instruction **li** de chargement d'un entier (32 bits) dans un registre, qui se traduit en un chargement des 16 bits de poids fort suivi d'un ou logique avec les 16 bits de poids faible. Il est d'ailleurs logique qu'une machine dont les instructions tiennent toutes sur 32 bits ne possède pas d'instruction de chargement d'un entier de taille 32 bits.

Un cas important est celui des instructions de comparaison entre registres et saut conditionnel. Le processeur fournit en fait seulement deux instructions, **beq** r_1, r_2, ℓ et **bne** r_1, r_2, ℓ, à savoir effectuer le saut vers ℓ si les contenus des deux registres r_1 et r_2 sont respectivement égaux ou différents. On peut obtenir toutes les instructions de comparaison et saut : **blt** (inférieur strict), **bge** (supérieur ou égal) etc. en combinant les opérations de comparaison, **slt**, sge, etc. et le test d'égalité au registre `zero`.

Assembleur	Langage machine	Commentaire
`blt` r, o, a	`slt $1,` r, o `bne $1, $0,` a	Justifie le registre `at ($1)` réservé par l'assembleur.
`li $t0, 400020`	`lui $1, 6` `ori $8, $1, 6804`	charge les 16 bits de poids fort puis les 16 bits de poids faible
`add $t0, $t1, 1`	`addi $8, $9, 1`	addition avec une constante
`move $t0, $t1`	`addu $8, $0, $9`	addition “unsigned” avec zéro

Pour voir, essayez :

```
% spim -notrap -file hello.spi
```

où `hello.spi` est un fichier assembleur quelconque, et regardez dans la zone dite `Text Segment`.

La zone `Text Segment` se présente sous la forme tabulée :

```
Adresse        Instruction En clair        Adresse             Instruction
machine        machine                     symbolique          assembleur

[0x00400000]   0x0109082a  slt $1, $8, $9                  ; 2: blt $t0, $t1, trois
[0x00400004]   0x14200003  bne $1, $0, 12 [trois-0x00400004]
[0x00400008]   0x3c01003d  lui $1, 61                      ; 4: li $t0, 4000020
[0x0040000c]   0x34280914  ori $8, $1, 2324
[0x00400010]   0x21280001  addi $8, $9, 1                  ; 6: add $t0, $t1, 1
[0x00400014]   0x00094021  addu $8, $0, $9                 ; 8: move $t0, $t1
```

On peut donc voir qu'à l'instruction assembleur **blt** `$t0, $t1, trois` correspond deux instructions machine, **slt** `$1, $8, $9` et **bne** `$1, $0, 12 [trois-0x00400004]`.

2.4 Exemples de programmes en assembleur

Cette section contient divers exemples de programmation en assembleur. Commençons par un premier exemple complet.

```
        .data
hello:  .asciiz "hello\n"  # hello pointe vers "hello\n\0"
        .text
        .globl __start
__start:
        li   $v0, 4        # la primitive  print_string
        la   $a0, hello    # a0  l'adresse  de  hello
        syscall
```

On remarque les détails suivants.

— Les directives `.data` et `.text` indiquent à l'assembleur où ranger ce qui va suivre, respectivement dans le segment statique de données et le segment de code.

— Les étiquettes sont suivies d'un deux points `:`.

— Les noms de registres sont précédés d'un dollar `$`, pour les distinguer des autres symboles (ainsi on peut appeler une étiquette `v0`).

— On dispose de directives particulières pour spécifier les données, ainsi `.asciiz` permet de décrire une chaîne de façon usuelle (ici le caractère `'\000'` est ajouté à la fin de la chaîne, selon la convention du langage C).

Le programme est assemblé, lié, chargé et lancé (ouf !) par :

```
spim -notrap -file hello.spi
```

Par convention le programme commence à l'étiquette `__start`. Si on retire l'option `-notrap`, l'éditeur de liens ajoute un prélude qui se branche à l'étiquette `main` (remplacer alors `__start` par `main`).

2.4.1 Conditionnelle

On utilise des sauts conditionnels et inconditionnels :

Pascal la fonction minimum

```
if t1 < t2 then t3 := t1 else  t3 := t2
```

Assembleur Mips

```
        blt  $t1, $t2, Then   # si t1 < t2 saut à Then
        move $t3, $t2         # t3 := t2
        j    End              # saut à End
Then:   move $t3, $t1         # t3 := t1
End:                          # suite du programme
```

2.4.2 Boucles

Pascal : calcule dans t2 = 0 la somme des entiers de 1 à t1

```
while t1 > 0 do begin t2 := t2 + t1; t1 := t1 -1 end
```

Programme équivalent

```
While:
    if t1 <= 0 then goto End
    else begin
        t2 := t2 + t1;
        t1 := t1 - 1;
        goto While
    end;
End:
```

Code Mips

```
While:
    ble  $t1, $0, End


    add $t2, $t2, $t1
    sub $t1, $t1, 1
    j   While

End:
```

On notera l'utilisation du registre $0 qui contient toujours zéro.

Une transcription alternative de la même boucle en assembleur est :

```
        j Test
Loop:
        add $t2, $t2, $t1
        sub $t1, $t1, 1
Test:
        bgt $t1, $0, Loop
```

L'avantage est qu'une itération de boucle ne donne lieu qu'à un unique saut, contre deux sauts pour le code précédent. Il est probable que le second programme est plus rapide, mais cela demande à être vérifié, comme tout ce que l'on peut supposer sur la rapidité des programmes en assembleur.

2.4.3 Expressions arithmétiques

Le processeur connaît les seules opérations élémentaires. Dès lors, lorsque l'on veut calculer une expression arithmétique, on décompose le calcul en étapes, en gardant les résultats intermédiaires dans des registres.

Pascal La distance

```
v0 := a0 * a0 + a1 * a1 ;
```

Assembleur Mips

```
mul $t0, $a0, $a0 # un premier carré
mul $t1, $a1, $a1 # le second
add $v0, $t0, $t1 # la somme
```

2.4.4 Les données

Données statiques

Les données statiques sont celles qui sont allouées par le compilateur. Dans un langage comme Pascal cela comprend au moins les variables globales.

```
const
  N = 1000 ;
var
  tableau : array [1..N] of integer ;
  c : char ;
  i : integer ;
```

Ici on déclare un tableau de 1000 entiers, un caractère et un entier. En Pascal, la déclaration d'une variable entraîne l'allocation de l'espace mémoire nécessaire et l'établissement d'un lien entre le nom de la variable et l'adresse de l'espace mémoire réservé.

Un code assembleur équivalent sera :

```
        .data
        .align 2        # aligner sur un mot (2^2 octets)
globaux:                # début de la zone des globaux
tableau:                # adresse symbolique de tableau
        .space 4000     # taille  en  octets
c:
        .space 1        # 1 octet
        .align 2
i:
        .space 4        # 4 octets
```

On remarque les contraintes d'alignement introduites pour que les mots se trouvent bien à des adresses de mots. La directive `.space` n de l'assembleur alloue n octets dans le segment de données. Accessoirement la valeur initiale de ces octets est zéro.

On utilisera ensuite les noms symboliques pour accéder aux variables. Par exemple, à une affectation `i := 10`, correspond le code assembleur suivant :

```
.text
la $a0, i
li $v0, 10
sw $v0, 0($a0)
```

Cette utilisation des noms symboliques est pratique, mais la pseudo-instruction `la` s'expanse en deux instructions et on peut faire mieux. Supposons que l'adresse du début de la zone des globaux se trouve dans un registre, par exemple le registre `gp`. On pourra alors écrire plus directement :

```
.text
li $v0, 10
sw $v0, 4004($gp)
```

Le chargement de l'adresse `globaux` dans `gp` ne pose pas de problème, on l'effectuera dans un code de mise en route à l'aide de l'instruction `la` déjà vue. Mais il faut connaître le décalage entre le début de la zone des globaux et l'adresse de `i` et il faut aussi que le déplacement tienne sur 16 bits. Or, un compilateur connaît la taille des données, et peut au moins contrôler la taille des décalages, voir implémenter une politique plus raffinée (un pointeur de données globales par groupe de fonctions, par exemple). Notons que l'assembleur peut aussi nous aider un peu, car on peut définir une constante symbolique égale à ce décalage.

```
ioff = 4004
.text
li $v0, 10
sw $v0, ioff($gp)
```

Certains assembleurs et éditeurs de liens pourraient même accepter une définition du style `ioff = i-globaux`, ce n'est pas le cas de SPIM.

Dans certains langages comme C on peut à la fois définir et initialiser une variable globale :

```
int i = 10 ;
```

L'assembleur fournit les directives correspondantes. Ici, on réserve un mot dans le segment de données et on donne sa valeur :

```
        .data
i:
        .word 10
```

Allocation dynamique

En cours de calcul on peut demander plus de mémoire au système d'exploitation qui sait étendre la zone de données du programme. Du point de vue du langage de programmation, on pensera à `new` de Pascal et Java, où à l'allocation mémoire implicite de Caml.

En SPIM, l'appel système numéro 9 prend la taille dans `v0` et retourne le pointeur vers le début de bloc dans `a0`.

```
# allouer a0 octets de mémoire.
brk:                # procédure d'allocation dynamique
        li $v0, 9   # appel système 9
        syscall     # alloue une taille a0 et
        j  $ra      # retourne le pointeur dans v0
```

En pratique, l'appel au système d'exploitation étant coûteux. On demande la mémoire au système par grosses quantités puis on satisfait les demandes dans les blocs ainsi préalloués. Un registre peut être utilisé pour contenir la première adresse libre.

```
        memsize = 1024*1024
__start:
        li  $a0, memsize
        jal brk
        move $t8, $v0     # t8 réservé
        ...
# allocation d'un tableau de a0 mots
new_array:
        sw  $a0, ($t8)    # écrit la taille dans l'entête
        add $v0, $t8, 4   # v0 <- adresse de la case 0
        add $a0, $a0, 1   # on alloue a0+1 mots
        mul $a0, $a0, 4   # en octets
        add $t8, $t8, $a0 # vraiment
        j   $ra
```

Ici, le code de lancement alloue une grosse zone de mémoire, tandis que la fonction `new_array` renvoie l'adresse d'une zone de mémoire allouée dans cette zone. L'argument `a0` de `new_array` est la taille demandée (en mots), on remarque qu'en fait on alloue en fait `a0`+1 mots et que le premier mot alloué contient la taille du tableau. On ignore les problèmes de débordement et de libération de l'espace mémoire —qu'il conviendrait de traiter, par exemple en modifiant `brk` et en utilisant un glaneur de cellules (*garbage collector*) ou une déallocation explicite (`dispose` en Pascal).

On peut également, si on souhaite se simplifier la vie, allouer toute la zone de mémoire « dynamique » statiquement. Il devient alors impossible d'agrandir cette zone au cours de l'exécution. On aura alors le code de lancement :

```
        memsize = 1024*1024
        cmemsize = 4 * memsize
        .data
dynamique:
        .space cmemsize
__start:
        la $t8, dynamique # t8 réservé
        ...
```

2.4.5 Procédures simples

Dans cette section je montre comment définir et utiliser une procédure simple, qui n'appelle pas d'autre procédure et prend des arguments peu nombreux.

Pour appeler une procédure, on utilise une l'instruction idoine **jal** qui range l'adresse de code la suivant dans le registre `ra`. À la fin d'une procédure, on retournera à l'appelant en sautant à l'adresse contenue dans `ra`. Rappelons aussi que les arguments de la procédure sont conventionnellement rangés dans certains registres (ici de `a0` à `a3`). En forçant un peu la note on remarque que le registre `ra` est un argument supplémentaire.

Soit, par exemple, on définit une procédure `writeln` qui imprime un entier puis un retour à la ligne.

```
        .data           # de la donnée
nl:
        .asciiz "\n"    # la chaîne "\n"

        .text           # du code
writeln:                # l'argument est dans a0
        li  $v0, 1      # le numéro de print_int
        syscall         # appel système
        li  $v0, 4      # la primitive  print_string
        la  $a0, nl     # la chaîne "\n"
        syscall
        j   $ra         # retour par saut à l'adresse ra
```

Voici ensuite un programme simple qui utilise la procédure `writeln` pour afficher les entiers 1 et 2 :

```
        .text           # du code
        .globl __start
__start:
        li   $a0, 1     # a0 <- 1
        jal  writeln    # ra <- pc+1; saut à writeln
        li   $a0, 2     # on recommence avec 2
        jal  writeln
        j    Exit       # saut à la fin du programme
Exit:                   # fin du programme
```

On remarque que, vu du programme principal, la procédure agit comme une instruction, (on l'exécute et on passe à la suivante). Il faut toutefois bien prendre garde à ce que la procédure utilise discrètement certains registres (ici le registre `v0`). Ainsi, si on souhaite afficher les entiers de 1 à 10 par une boucle, on ne pourra pas utiliser `v0` comme compteur de boucle.

Les fonctions sont tout simplement des procédures qui rendent un résultat, ce résultat est rendu dans un registre conventionnel, ici `v0`. On écrira une fonction `twice` qui double son argument ainsi :

```
        .text
twice:                     # l'argument est dans a0
        add $v0, $a0, $a0  # v0 <- a0 + a0
        j $ra
```

2.4.6 Procédures compliquées

On considère maintenant le cas le plus compliqué pour les procédures : le cas des procédures récursives, c'est à dire des procédures qui s'appellent elles-mêmes. L'exemple typique est celui de la fonction factorielle :

```
function fact (n : integer) : integer ;
begin
  if n <= 0 then
    fact := 1
  else
    fact := n * fact (n-1)
end ;
```

Si nous traduisons ce code en suivant la convention de l'argument passé dans `a0` et du résultat rendu dans `v0` nous obtenons ce code :

```
        .text
# a0 est n
fact:
        ble $a0, $0, L1  # si a0 <= 0 aller en L1
        sub $a0, $a0, 1  # argument de l'appel
        jal fact         # v0 <- fact (n-1)
        mul $v0, $v0, $a0  # v0 <- n * v0
        j   $ra
L1:
        li  $v0, 1
        j   $ra
```

Ce code est bien entendu incorrect, l'erreur la plus visible concerne `a0` dans le cas `n > 0`. Après le **jal** `fact`, le registre `a0` ne contient plus `n`. De fait, il semble qu'il doivent contenir zéro. Mais il y a une seconde erreur, un peu plus cachée : le contenu du registre `ra` est détruit par l'instruction **jal** `fact` et l'appel initial à `fact` ne retournera jamais. Graphiquement, on a la situation suivante :

$$f_n \left\{ \begin{array}{l} a_0 \leftarrow a_0 - 1 \\ f_{n-1} \left\{ \begin{array}{l} a_0 \leftarrow a_0 - 1 \\ \cdots \\ v_0 \leftarrow v_0 \times a_0 \end{array} \right. \\ v_0 \leftarrow v_0 \times a_0 \end{array} \right.$$

Où, au pire, on aura dans les n incarnations différentes de l'argument `n` et de l'adresse de retour.

Comme la fonction est récursive, il est vain de tenter de sauver les contenus de `a0` et `ra` dans d'autres registres, l'appel récursif de `fact` détruira aussi les contenus de ces sauvegardes, car il exécutera le même code de sauvegarde. Il convient donc que chaque *appel* de fonction possède en propre un bout de mémoire, pour sauver le contenu des registres qui seront (ou risquent d'être) modifiés par l'appel récursif et dont les valeurs seront encore nécessaires (lues) au retour de l'appel. Cet espace est alloué sur la pile.

La pile

Par convention, la pile grossit vers les adresses décroissantes et le registre `sp` pointe vers le dernier mot utilisé.

Pour sauver un registre r sur la pile, on écrit :

```
sub $sp, $sp, 4  # alloue un mot sur la pile
sw  r, 0($sp)    # écrit r sur le sommet de la pile
```

Pour restaurer un mot de la pile dans un registre r, on écrit :

```
lw  r, 0($sp)    # lit le sommet de la pile dans r
add $sp, $sp, 4  # désalloue un mot sur la pile
```

En général, on alloue et désalloue l'espace en pile par blocs pour plusieurs sauvegardes à la fois. Ainsi, dans le cas de la factorielle, où il y a deux registres à sauvegarder, on commencera par réserver 2 mots en pile, et on oubliera pas de les rendre. Le code est modifié par une sauvegarde préalable des registres `a0` et `ra` dans les registres `t0` et `t1`, afin de mettre en vedette la sauvegarde des registres. En effet, les registres argument `a0` et adresse de retour `ra` servent a communiquer entre le code qui appelle une fonction (l'appelant ou *caller*) et le code de la fonction (l'appelé ou *callee*) et toute discussion de leur sauvegarde manquera un peu de pureté. Bref, on obtient :

```
        .text
# a0 est n
fact:
        ble $a0, $0, L1
        move $t0, $a0  # sauvegarde a0 -> t0
        move $t1, $ra  # sauvegarde ra -> t1
        sub $a0, $t0, 1
        sub $sp, $sp, 8  # réserver deux mots
        sw $t0, 0($sp)  # sauvegarder t0
        sw $t1, 4($sp)  # sauvegarder t1
        jal fact        # car fact peut modifier t0 et t1
        lw $t1, 4($sp)  # restaurer t1
        lw $t0, 0($sp)  # restaurer t0
        add $sp, $sp, 8  # rendre l'espace de pile
        mul $v0, $v0, $t0  # utilisation de t0
        j   $t1         # utilisation de t1
L1:
        li  $v0, 1
        j   $ra
```

Il faut noter qu'ici c'est l'appelant qui sauve les registres dont il sait avoir besoin et dont il pense qu'ils risquent d'être modifiés par un appel de procédure, c'est la convention dite *caller save*.

Il existe bien entendu la convention inverse (dite *callee save*), où l'appelé sauvegarde le contenu des registres dont il sait qu'il les modifie et dont il pense que l'appelant peut avoir encore besoin. Ici on obtiendra sensiblement le même code ! La différence essentielle est que les registres `t0` et `t1` sont cette fois sauvegardés avant d'être affectés.

```
        .text
# a0 est n
fact:
        ble $a0, $0, L1
        sub $sp, $sp, 8  # réserver deux mots
        sw $t0, 0($sp)  # sauvegarder t0
        sw $t1, 4($sp)  # sauvegarder t1
        move $t0, $a0
        move $t1, $ra
        sub $a0, $t0, 1
        jal fact        # au retour de fact t0 et t1 n'ont pas changé
        mul $v0, $v0, $t0  # utilisation de t0
        move $ra, $t1  # utilisation de t1
        lw $t1, 4($sp)  # restaurer t1
        lw $t0, 0($sp)  # restaurer t0
        add $sp, $sp, 8  # rendre l'espace de pile
        j   $ra
L1:
        li  $v0, 1
        j   $ra
```

Évidemment, s'il s'agit de coder la fonction factorielle en assembleur, on se passera des sauvegardes en registres et on écrira plus directement :

```
fact:   blez $a0, fact_0   # si a0 <= 0 saut à fact_0
        sub  $sp, $sp, 8   # réserve deux mots en pile
        sw   $ra, 0($sp)   # sauve l'adresse de retour
        sw   $a0, 4($sp)   # et la valeur de a0
        sub  $a0, $a0, 1   # décrémente a0
        jal  fact          # v0 <- appel récursif (a0-1)
        lw   $a0, 4($sp)   # récupère a0
        mul  $v0, $v0, $a0 # v0 <- a0 * v0
        lw   $ra, 0($sp)   # récupère l'adresse de retour
        add  $sp, $sp, 8   # libère la pile
        j    $ra           # retour à l'appelant

fact_0:
        li   $v0, 1        # v0 <- 1
        j    $ra           # retour à l'appelant
```

Utilisation simple de la pile

Les quelques exemples de programmation assembleur que nous avons vus sont typiques de la programmation assembleur à la main. Dans ce contexte et pour des programmes courts, il est assez facile de bien se servir des registres et donc de produire des programmes relativement efficaces.

Il est toutefois important de connaître le principe de techniques plus simples d'utilisation de la pile.

- Dans le cas de langages tels que C et Pascal (ou même Java), dès que l'on a compris que les arguments et les variables locales des fonctions sont « en pile », on comprend beaucoup de choses.
- On peut vouloir construire un compilateur rapidement sans se préoccuper exagérément de l'efficacité.
- Les machines virtuelles ont peu de registres et fonctionnent selon le modèle simple, car il y a peu à gagner en introduisant de nombreux registres qui sont en fait des cases mémoires du programme machine virtuelle.

Dans un modèle simple, il n'y a que quelques registres spécialisés, et entre autres un pointeur de pile `sp`.

- Les opération prennent leur arguments sur la pile et renvoient leur résultat sur la pile. Ainsi une addition dépile deux entiers et empile leur somme.
- Les fonction font de même.
- Les variables locales des fonctions correspondent à des emplacements de pile.

On se donne parfois un registre supplémentaire, l'*accumulateur* que l'on peut voir comme le sommet de la pile du modèle précédent. Ainsi une addition range la somme de l'accumulateur et d'une valeur dépilée dans l'accumulateur.

Sans développer exagérément, la fonction `twice`, qui double son argument, écrite selon ces idées donnera ceci :

```
#l'accumulateur est v0, on dispose de quelques registres
twice:                          # l'argument n est 0(sp), le retour 4(sp)
        lw   $v0, 0($sp)        # accu <- n
        sub  $sp, $sp, 4        # empiler
        sw   $v0, 0($sp)
        lw   $v0, 4($sp)        # accu <- n
        lw   $t0, 0($sp)        # dépiler
        add  $sp, $sp, 4
        add  $v0, $v0, $t0      # addition
        add  $sp, $sp, 4        # dépiler l'argument
        lw   $ra, 0($sp)        # dépiler adresse de retour
        add  $sp, $sp, 4
        j    $ra                # retour
```

Code qui peut paraître abscons, mais qui devient peut être plus compréhensible comme expansion simple du bytecode (Caml) suivant :

```
twice:
        acc 0
        push
        acc 1
        addint
        return 1
```

Conventions d'appel

Il est maintenant temps de revenir sur les noms symboliques des registres (voir section 2.2.2). En général :

- les arguments sont passés dans les registres `a0` à `a3`. Les arguments en excès sont passés sur la pile.
- La ou les valeurs de retour sont dans `v0` et `v1`. Comme nous ne concevront que des fonction retournant un unique résultat, nous pouvons récupérer `v1` pour un autre usage.
- Les registres `s0` à `s7` sont des *callee save*. Cela implique de les rendre dans l'état où on les a pris.
- Les registres `t0` à `t9` sont des *caller save*. Cela entraîne de ne pas supposer que leur contenu sera retrouvé intact après un appel de procédure.
- Le registre `zero` contient toujours zéro, quoiqu'il advienne. Cette particularité est liée à la conception même du processeur MIPS. Disons qu'elle permet, par exemple, de se passer d'instruction de négation (**sub** `$v0,$zero,$v0`) et de l'instruction de transfert de registre à registre (**add** `$v1,$zero,$v0`). Le format des instructions que le processeur décode en dernière analyse en est simplifié.
- Nous connaissons bien le registre `ra`, qui reçoit l'adresse de retour lors des appels de procédure.
- Le registre `sp` est le pointeur de pile, il contient la limite inférieure de la pile. Il ne faut rien supposer sur le contenu des adresses mémoires au delà de cette limite, car il peut être modifié par le système d'exploitation, lors du traitement d'une interruption par exemple.
- Le registre `gp` est le *global pointer*. Il pointe conventionnellement vers une zone de données globales. Cela permet l'accès à une variable globale en une seule instruction, à condition l'adresse de cette donnée soit à une distance de `gp` exprimable sur 16 bits.
- Le registre `fp` est le *frame pointer*. Il contient grosso modo la limite supérieure de la zone de pile allouée par une fonction. Cette redondance est utile à un *debugger*, qui peut ainsi mettre en rapport facilement positions en pile et noms symboliques des variables. Elle est utile dans le cas d'une technique de compilation qui peut faire croître la pile à l'intérieur des fonctions, et nécessaire lorsque la pile peut croître de façon inconnue à la compilation

(allocation de tableaux en pile). En effet, `fp`, qui, contrairement au pointeur de pile `sp`, ne change pas au cours de l'exécution du code d'une fonction, est alors une référence stable vers les position en pile. Nous ne nous servirons pas de `fp` de cette façon et pourrons le considérer comme un *callee save* supplémentaire d'ailleurs prévu et dénommé `s8`.

— Le registre `at` est réservé pour expansion des pseudo-instructions. On peut s'en servir si on évite d'utiliser les pseudo instructions. . .

— Les registres `k0` et `k1` sont réservés au système d'exploitation, seuls ceux qui écrivent ce système peuvent s'en servir (et savent le faire !).

Rappelons, que sauf pour pour `zero`, `ra`, `k0` et `k1` (voire `at`) on peut ne pas suivre ces conventions. Mais alors on est tout seul, on ne peut pas interagir avec le monde extérieur.

Un peu de culture

Il est bien connu dans les milieux de la « *vraie programmation* » que la pile « *c'est mal* ». En fait, c'est la récursion qui est visée, et effectivement il peut arriver qu'un programme par ailleurs réputé correct en théorie échoue par épuisement de la mémoire, phagocytée par la pile des nombreux appels en cours.

Mais, si la récursion est interdite (et elle l'était dans les vieilles versions de Fortran), alors on peut tout simplement allouer l'espace nécessaire à une fonction dans le segment des données statiques, c'est à dire, lors de la compilation. Dès lors, il n'y a plus aucun risque d'épuisement de la mémoire à l'exécution du fait des appels de fonctions. Mieux, l'ensemble des rapports entre les fonctions peut être assez facilement connu du compilateur et les conventions d'utilisation des registres adaptées en conséquence.

Ce point de vue est en voie d'extinction, en raison de l'expressivité de la récursion et de l'augmentation de la taille des mémoires. Mais la pile peut toujours déborder, bien évidemment.

Chapitre 3

Le langage Pseudo-Pascal

Avant de décrire les phases de notre compilateur une à une, en suivant l'ordre de leur application, il me paraît plus malin de commencer par décrire d'abord notre langage de programmation, en détaillant sa sémantique plutôt que sa syntaxe.

3.1 Expressivité des langages de programmation

Les langages de programmation sont les langages que l'être humain utilise pour dire à un ordinateur ce qu'il doit faire. On peut évoquer les catégories suivantes :

Langages généraux Ils doivent être complets, *i.e.* permettre d'exprimer tous les algorithmes calculables. Au minimum, il faut une mémoire réputée infinie (c'est à dire grande) et la possibilité d'exprimer la récursion (construction primitive ou boucle `while`). Ex : Fortran, Pascal, C, Caml, etc.

Langages spécialisés (DSL pour *Domain Specific Languages*) Ce sont par exemple les langages pour le graphisme, pour commander des robots ou encore la calculette. Ils peuvent ne pas être complets.

Bien qu'ils permettent d'exprimer tous les calculs possibles, (c'est approximativement la thèse de Church), les langages généraux ne sont pas tous équivalents stricto-sensu. Ils se distinguent par leur *expressivité*, c'est à dire par leur capacité d'exprimer des algorithmes succinctement (et directement).

Par exemple, on peut toujours implémenter un algorithme récursif à l'aide d'une pile explicite (un tableau plus un indice), mais le langage qui offre la récursivité permet une implémentation plus concise et élégante.

Plus précisément, lorsque l'on se pose la question de l'expressivité d'un langage on peut examiner les points suivants :

Les fonctions

— Les fonctions peuvent être définies localement à d'autres fonctions, comme en Pascal, ou pas, comme en C.

— Les fonctions peuvent être des valeurs du langages comme en Caml (ou en C !), ou pas, comme en Pascal où une fonction ne peut pas rendre une autre fonction comme résultat.

Les structures de données Au delà des divers entiers et des tableaux la plupart des langages fournissent des produits (*records* de C et Pascal, paires de ML) et des sommes (*enums* et surtout *unions* de C, types dits concrets de ML). Lisp organise toutes ses données autour de la liste que l'on peut voir comme la somme de la liste vide (*nil*) et de la cellule de liste (*cons*). Certaines structures de données comme les chaînes (des tableaux de caractères de taille variable) ne semblent pas à première vue étendre beaucoup l'expressivité du langage, mais leur intérêt pratique apparaît très vite dès que l'on programme.

Les modules, les objets sont des traits qui autorisent la programmation incrémentale.

Le typage restreint l'expressivité au profit de la sécurité. On considérera d'abord le système de type du langage, par exemple en ML on dispose d'un polymorphisme dit générique qui n'existe pas en Java — en ML on pourra écrire une fonction identité qui accepte tous les arguments possibles, c'est impossible en Java.

On pourra aussi s'intéresser à l'impact du système de type sur la programmation : les types sont-ils construits par le compilateur comme en ML, essentiellement donnés par le programmeur comme en C, Pascal et Java, inexistant dans les programmes comme en Lisp et Basic, mais bien présents à l'exécution. Attention, contrairement à une opinion assez répandue, il existe très peu de langages absolument non-typés, à part peut être l'assembleur, on distinguera plutôt entre langages typés statiquement (ML, C, Pascal) : le compilateur vérifie que le programme est bien typé, l'exécution n'échoue jamais (ou rarement) ; et langages typés dynamiquement (Lisp, Basic) : le compilateur ne vérifie rien, mais l'exécution vérifie le bien fondé d'une opération avant de la réaliser. On notera que Java est peu les deux à la fois et on se gardera de trop conclure.

Note : Pour comparer l'expressivité formellement, il faut en général le faire point à point en exhibant un codage d'un langage dans un autre qui respecte certaines règles (compositionalité, localité, *etc.*)

3.2 Comment définir un langage

3.2.1 Syntaxe

La syntaxe décrit les mots et les phrases du langage. Elle ne donne aucun sens aux phrases.

On peut distinguer syntaxe concrète et syntaxe abstraite. La syntaxe concrète est le discours lui-même, en informatique c'est un fichier, mais on pourrait conceptuellement déclamer des programmes. C'est disons une suite de lettres qui forment des mots qui forment des phrases. La syntaxe abstraite est la structure du discours, en informatique c'est un arbre. Dans les compilateurs, le programme à compiler est bien un arbre, dans le discours sur le langage c'est un dessin. La grammaire est la définition de tous les arbres possibles c'est à dire de tous les discours syntaxiquement bien formés d'un langage.

En pratique il est bien commode de commencer par reconnaître les mots avant de s'attaquer aux phrases. On passe donc de la syntaxe concrète à la syntaxe abstraite en deux temps, d'abord par l'analyse *lexicale* qui traduit une suite de caractères en suite de mots puis par l'analyse *grammaticale* qui transforme une suite de mots en arbre de syntaxe abstraite.

L'exemple du langage des expressions arithmétiques éclairera un peu ces notions.

Commençons par définir les entiers, comme une suite de chiffres, les variables comme des suites de caractères alphabétiques, les blancs comme des suites d'espaces et quelques caractères particuliers comme des mots (≪ `(` ≫, ≪ `+` ≫, etc.) On exprimera les deux syntaxes à partir des mots ainsi définis.

Syntaxe concrète représentée par une grammaire, (dans le style dit BNF)

expression	`::=`	**ENTIER**
	`|`	**VARIABLE**
	`|`	*expression binop expression*
	—	`(` *expression* `)`
binop	`::=`	`+	-	*	/`

Dans ce style de description, on distingue les **terminaux**, qui sont les mots, et les *non-terminaux* qui sont les noms définis par la grammaire.

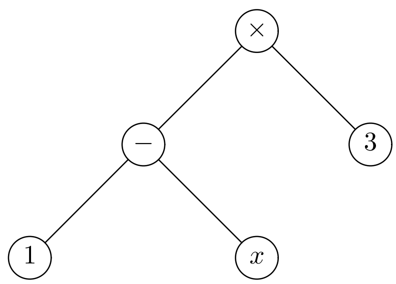
Syntaxe abstraite (en Caml) Représentée par le type `expression`.

```
type expression =
  | Const of int
  | Variable of string
  | Bin of opérateur * expression * expression
and opérateur = Plus | Moins | Mult | Div;;
```

Exemple Ainsi les expressions ≪ `(1 - x) * 3` ≫ et ≪ `(1-x)*(3)` ≫ ont la même syntaxe abstraite :

```
Bin (Mult, Bin (Moins, Const 1, Variable "x"), Const 3);;
```

Les malins remarqueront que l'arbre ci-dessus est décrit sous forme de syntaxe (concrète !) Caml. On devrait donc plutôt dessiner un arbre :



On voit alors que le principal effet de l'analyse syntaxique est de remplacer une structure arborescente décrite à l'aide de parenthèses en cette structure elle-même.

La syntaxe concrète peut aussi permettre d'exprimer la même construction de syntaxe abstraite sous différentes formes. Ainsi, le caractère ≪ A ≫ peut être représenté par ≪ `'A'` ≫ et ≪ `'\065'` ≫, ou plus significativement la construction de liaison de Caml(-Light) peut s'écrire ≪ `let` *d* `in` *e* ≫ ou ≪ *e* `where` *d* ≫. Il convient en général de ne pas abuser de cette possibilité, car elle va contre le principe d'économie de moyens : à quoi bon donner deux façons d'exprimer la même chose ? En outre elle peut rendre les messages d'erreur du compilateur inintelligibles.

3.2.2 Sémantique

Il s'agit de donner un sens aux phrases. C'est beaucoup plus facile à faire dans le cas d'un langage de programmation que dans le cas de la langue naturelle. Il s'agit ici d'expliquer ce qu'un programme fait.

On distinguera en gros trois façons de procéder.

Sémantique informelle Un document de référence décrit la sémantique. Ce document est écrit dans un langage technique qui fait appel à la culture informatique du lecteur. Voici par exemple la sémantique de la boucle `while` en C.

> *L'instruction* `while` *est de la forme :*
>
> `while (`*expression*`)` *instruction*
>
> *la sous-instruction est exécutée de manière répétée tant que la valeur de l'expression reste non nulle. On teste l'expression avant d'exécuter l'instruction.*

L'avantage est bien entendu que la langue naturelle se prête bien aux vastes concepts et aux descriptions synthétiques et qu'elle est connue du lecteur. On notera ici le choix du mot *reste*, à mettre en rapport avec l'idée qu'une boucle peut être exécutée plusieurs fois et que la valeur d'une expression peut changer au cours du temps. L'inconvénient est le manque de rigueur, en particulier la fameuse culture informatique est en perpétuelle évolution et il n'est pas toujours évident de faire la part des concepts généraux (comme ≪ un appel de fonction ≫) et des détails d'implémentation (comme ≪ une adresse mémoire ≫ ou ≪ la zone de code ≫). En outre, il est impossible de faire des preuves satisfaisantes sans une description plus formelle. On pourrait par exemple vouloir prouver qu'une optimisation compliquée ne change pas les résultats d'un programme, ou qu'un programme bien typé n'échoue pas lors de l'exécution.

Sémantique dénotationnelle C'est exactement le contraire de la précédente, on cherche à associer à un programme un objet construit selon les règles de l'art mathématique. Les valeurs d'un langage sont généralement modélisées comme des treillis, les fonctions comme des fonctions continues sur les treillis, la récursion comme un opérateur de point fixe (solution d'une équation) etc. L'avantage est la certitude d'existence mathématique des objets calculés. L'inconvénient est que ce n'est ni toujours facile, ni toujours très parlant.

Sémantique opérationnelle On cherche cette fois à décrire l'effet des programmes. On définit un ensemble de valeurs (ou résultats) puis une relation d'évaluation qui relie des programmes avec des résultats.

La principale différence avec la sémantique dénotationnelle est que le domaine sémantique est d'emblée plus simple, on se préoccupe plus de décrire les résultats possibles que de donner un sens mathématique à chaque bout de syntaxe. La définition des programmes comme une relation vient ensuite plus comme une description suffisamment abstraite du calcul que comme la volonté d'associer les programmes à une valeur du domaine. Le formalisme de cette description peut être emprunté à la logique formelle.

3.3 Sémantique opérationnelle de la calculette

3.3.1 Un interpréteur

Une première possibilité est de donner la sémantique opérationnelle comme un programme. Cela revient à écrire un interpréteur. Les valeurs sont les entiers.

```
type valeur = int
type environnement = (string * valeur) list
```

On peut définir l'évaluation par un programme Ocaml qui prend un environnement initial associant des valeurs à certaines variables, une expression à évaluer et retourne un entier :

```
let cherche x env = List.assoc x env
let rec évalue env = function
| Const n -> n
| Variable x -> cherche x env
| Bin (op, e1, e2) ->
    let v1 = évalue env e1 and v2 = évalue env e2 in
    begin match op with
    | Plus -> v1 + v2 | Moins -> v1 - v2
    | Mult -> v1 * v2 | Div -> v1 / v2
    end
```

(Notons que la sémantique dénotationnelle donnerait le sens d'un programme comme une fonction des environnements dans les entiers.)

3.3.2 Une présentation plus neutre

Définir la sémantique à l'aide d'un programme interpréteur n'est pas très satisfaisant. La présentation manque à la fois d'abstraction (des détails peu important sont mis en avant) et de neutralité (la description se fait dans un langage de programmation particulier).

On a recours à un formalisme spécifique dit, **S**émantique **O**pérationnelle **S**tructurelle qui, de fait, décrit un interpréteur indépendamment de son implémentation.

L'idée est de définir une relation $\rho \vdash e \Rightarrow v$ qui se lit ≪ *Dans l'environnement ρ, l'expression e s'évalue en la valeur v* ≫, par des règles d'inférence ; c'est-à-dire comme le plus petit ensemble vérifiant une certains nombre de règles d'inférence.

Une règle d'inférence est une implication $P_1 \wedge \ldots P_k \Longrightarrow C$ présentée sous la forme

$$\frac{P_1 \wedge \ldots \wedge P_k}{C}$$

que l'on peut lire pour réaliser (évaluer) C il faut réaliser à la fois P_1 et ... P_k.

Dans les jugements de la forme $\rho \vdash e \Rightarrow v$:

— ρ lie des variables x à des valeurs v, i.e. c'est un par exemple un ensemble de paires notées $x \mapsto v$, mais on peu aussi le définir de façon plus abstraite comme une fonction des noms de variables dans les valeurs.

— Si on prend l'exemple facile de la calculette, on aura $e \in$ *expressions* et $v \in \mathbf{Z}$ (entiers relatifs). Dans un excès de formalisem, on note $\bar{n}$ l'entier relatif associé à sa représentation n. Les quelques règles suivantes définissent alors la sémantique de la calculette.

$$\rho \vdash \texttt{Const}\ n \Rightarrow \bar{n} \qquad\qquad \frac{x \in dom\,(\rho)}{\rho \vdash \texttt{Variable}\ x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Bin}\,(\texttt{Plus}\,, e_1, e_2) \Rightarrow v_1 + v_2} \qquad\qquad \frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Bin}\,(\texttt{Times}\,, e_1, e_2) \Rightarrow v_1 * v_2}$$

3.4 Diverses constructions et leur sémantique

Sans prétendre a donner formellement toute la sémantique de Pseudo-Pascal. Cette section montre comment exprimer ses constructions (et d'autres) dans le formalisme SOS.

3.4.1 Les liaisons

Commençons par une construction simple : la liaison locale (le `let` de Caml) de syntaxe concrète :

let VARIABLE = *expression* **in** *expression*

On ajoute un noeud de syntaxe abstraite :

```
| Let of string * expression * expression
```

Informellement, l'expression $\texttt{Let}\,(x, e_1, e_2)$ lie la variable x à l'expression e_1 dans l'évaluation de l'expression e_2.

Formellement :

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho, x \mapsto v_1 \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Let}\,(x, e_1, e_2) \Rightarrow v_2}$$

où $\rho, x \mapsto v$ ajoute la liaison de x à v dans l'environnement ρ en cachant une ancienne liaison éventuelle de x. C'est à dire que $\rho, x \mapsto v$ associe v à x, que ρ possède déjà une liaison pour x ou pas.

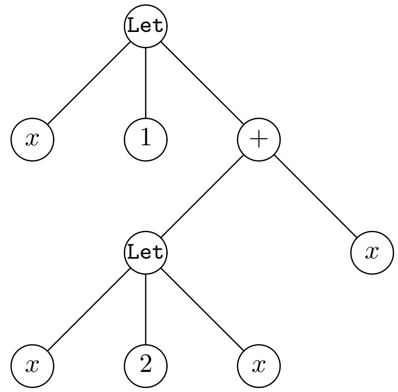
On voit bien comment on peut comprendre une telle règle en modifiant l'interpréteur des expressions arithmétiques. Les environnements ρ étant codés par des paires l'ajout d'une liaison se code très simplement.

```
let ajoute x v env = (x,v)::env
```

Il reste à étendre la fonction d'évaluation.

```
let rec évalue env =
  ...
| Let (x, e1, e2) ->
    let v1 = évalue env e1 in
    évalue (ajoute x v1 env) e2 ;;
```

Ainsi étant donnée l'expression : `let x = 1 in (let x = 2 in x) + x`, on a la syntaxe abstraite.



La sémantique est idéalement donnée par un *arbre de dérivation* qui est la preuve que la valeur de l'expression est 3.

$$\dfrac{\emptyset \vdash \texttt{Const}\ 1 \Rightarrow 1 \qquad \dfrac{\dfrac{\dfrac{x \mapsto 1 \vdash \texttt{Const}\ 2 \Rightarrow 2 \qquad x \mapsto 1, x \mapsto 2 \vdash x \Rightarrow 2}{x \mapsto 1 \vdash \texttt{Let}\,(x, \texttt{Const}\ 2, x) \Rightarrow 2}}{\vdots} \qquad x \mapsto 1 \vdash x \Rightarrow 1}{x \mapsto 1 \vdash \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Let}\,(x, 2, x), x) \Rightarrow 3}}{\emptyset \vdash \texttt{Let}\,(x, \texttt{Const}\ 1, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Let}\,(x, 2, x), x)) \Rightarrow 3}$$

Sous forme compacte, cet arbre exprime aussi un tour d'évaluation de l'interpréteur.

3.4.2 Langages impératifs

Jusqu'ici non avons considéré l'évaluation des expressions. Du point de vue de la sémantique les expressions ont une valeur. Dans notre langage Pseudo-Pascal, comme dans tous les langages impératifs, on considère aussi les *instructions*. Un ensemble minimal d'instructions comprend l'affectation et la séquence :

```
type instruction =
| Affecte of string * expression
| Sequence of instruction * instruction
```

Avec pour syntaxe concrète :

instruction	`::=`	**VARIABLE :=** *expression*
	`|`	*instruction* **;** *instruction*

Du point de vue sémantique l'exécution d'une instruction ne produit pas de valeur en elle même, mais modifie un « *état* ». En faisant abstraction des entrées/sorties, on peut limiter l'état à la *mémoire*. On modélise la mémoire comme une fonction σ des adresses mémoire ℓ dans les valeurs v. L'environnement ρ est maintenant une fonction des noms de variables x dans les adresses ℓ. L'exécution d'une instruction i est rendu par un jugement $\rho/\sigma \vdash i \Rightarrow /\sigma'$ qui se lit *dans l'état-mémoire σ et l'environnement ρ, l'exécution de l'instruction i produit un nouvel état mémoire σ'*. Des règles sémantiques possibles de l'affectation et de la séquence sont alors :

$$\frac{\rho/\sigma \vdash i_1 \Rightarrow /\sigma_1 \qquad \rho/\sigma_1 \vdash i_2 \Rightarrow /\sigma_2}{\rho/\sigma \vdash \texttt{Seq}\,(i_1, i_2) \Rightarrow /\sigma_2} \qquad \frac{x \in dom\,(\rho) \qquad \rho(x) \in dom\,(\sigma) \qquad \rho/\sigma \vdash e \Rightarrow v}{\rho/\sigma \vdash \texttt{Affecte}\,(x, e) \Rightarrow /\sigma, \rho(x) \mapsto v}$$

Il faut aussi modifier les règles d'évaluation des expressions pour tenir compte de la mémoire σ. Il n'y a en fait pas grand chose à modifier, sauf peut-être en ce qui concerne l'accès aux variables.

$$\rho/\sigma \vdash \texttt{Const}\ n \Rightarrow \bar{n} \qquad \frac{x \in dom\,(\rho) \qquad \rho(x) \in dom\,(\sigma)}{\rho/\sigma \vdash \texttt{Variable}\ x \Rightarrow \sigma(\rho(x))}$$

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow v_1 \qquad \rho/\sigma \vdash e_2 \Rightarrow v_2}{\rho/\sigma \vdash \texttt{Bin}\,(\texttt{Plus}\,, e_1, e_2) \Rightarrow v_1 + v_2}$$

Le traitement impératif des liaisons est bien différent de la liaison `let`. Les variables peuvent apparaître à gauche du signe d'affectation `:=` et correspondre à des adresses ℓ, ou dans les expressions et correspondre à des valeurs v. On parle alors parfois de *left-value* et de *right-value*.

Dans l'évaluateur, on peut se passer d'un encodage explicite de la mémoire en utilisant les valeurs mutables de Caml, c'est à dire que les environnements associent maintenant des noms de variables à des références de Caml.

```
type environnement = (string * valeur ref) list
```

L'interpréteur est maintenant constitué de deux fonctions, l'une pour évaluer les expressions l'autre pour exécuter les instructions.

```
let rec évalue env = function
   ...
  | Variable x -> !(cherche x env)
   ...

and execute env = function
| Sequence (i1, i2) -> execute env i1 ; execute env i2
| Affecte (x, e)    ->
    let v = évalue env e in
    let cell = cherche x env in
    cell := v
```

Il faut bien remarquer les deux points.

— Nous ne disposons d'aucune règle pour introduire de nouvelles adresses mémoires, c'est à dire pour allouer. Informellement, ce sont les déclarations de variables qui allouent de la mémoire. Par exemple, si on a une déclaration de variable `x` : **integer** alors les instructions qui sont dans la portée de cette déclaration, s'exécuteront dans un environnement ρ qui lie x à l'adresse contenant une valeur initiale. En Pseudo-Pascal, cette valeur initiale est une valeur invalide notée $\perp$.

  Formellement, mais sans exagérer, imaginons un Pseudo-Pascal très simple qui autorise une seule variable dans les déclarations **var**. Alors la syntaxe abstraite d'un programme **var** `x` : **integer** *instruction*, est un enregistrement :

```
type programme = {variable : string ; instruction : instruction}
```

  L'exécution d'un programme {`variable=`x `;` `instruction=`i} est rendue par le jugement :

$$x \mapsto \ell/\ell \mapsto \perp \vdash i \Rightarrow /\sigma$$

  Dans l'interpréteur, on modifie le type des valeurs et on ajoute une fonction chargée d'exécuter les programmes.

```
type valeur = Undefined | Int of integer

let rec évalue env = function
| Const n -> Int n
  ...

and execute env = function
  ...

let execute_programme {variable=x ; instruction=i} =
  let env = [x, ref Undefined] in execute env i
```

— Le choix de diviser la syntaxe abstraite entre expressions et instructions est bien un choix. Par exemple, Caml possède bien des traits impératifs, mais pas d'instructions. En ce cas, la séquence et l'affectation doivent rendre un résultat et la règle d'évaluation des expressions doit rendre un état mémoire modifié. Le choix des valeurs rendues est assez simple, la valeur d'une séquence `Seq` (e_1, e_2) est la valeur de e_2 et la valeur d'une affectation `Affecte` (x, e) est la valeur de e. Par exemple, en considérant que la séquence est maintenant une expression, on aura :

```
let rec évalue env = function
   ...
| Sequence (e1, e1) ->
    let _ = évalue env e1 in (* explicitement ignorer v1 *)
    evalue env e2
```

3.4.3 Les booléens et la conditionnelle

Les booléens (*true* et *false*) nous conduisent à distinguer entre valeurs entières et booléennes. Le domaine des valeurs est alors la réunion de l'ensemble des entiers et de celui des booléens. Si on veut être très formel, il faut alors explicitement utiliser des fonctions canoniques pour passer des entiers aux valeurs et des valeurs aux entiers. On ne le fera pas dans les règles de SOS. Toutefois, il est intéressant de constater que l'implémentation naturelle en Caml du type des valeurs comme un type somme impose d'expliciter ces fonctions canoniques.

```
type valeur = Int of int | Bool of bool

let int_to_valeur i = Int i
and valeur_to_int = function
| Int i -> i

let rec évalue env = function
| Const i ->  int_to_valeur i

| Bin (op, e1, e2) ->
    let v1 = valeur_to_int (évalue env e1)
    and v2 = valeur_to_int (évalue env e2) in
    int_to_valeur
      (match op with
      | Plus -> v1 + v2 | Moins -> v1 - v2
      | Mult -> v1 * v2 | Div -> v1 / v2)
```

En tant que tels les booléens ont peu d'intérêt, il faut se donner une expression conditionnelle pour les utiliser vraiment. La syntaxe en est bien connue :

$$\begin{array}{lcl} \textit{expression} & \texttt{::=} & \ldots \\ & \texttt{|} & \textbf{if } \textit{expression} \textbf{ then } \textit{expression} \textbf{ else } \textit{expression} \end{array}$$

```
type expression = ...
| If of expression * expression * expression
```

La sémantique aussi est bien connue.

$$\frac{\rho \vdash e_1 \Rightarrow true \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash \texttt{If}\,(e_1, e_2, e_3) \Rightarrow v} \qquad \frac{\rho \vdash e_1 \Rightarrow false \qquad \rho \vdash e_3 \Rightarrow v}{\rho \vdash \texttt{If}\,(e_1, e_2, e_3) \Rightarrow v}$$

Soit encore :

```
let rec évalue env = function
        ...
| If (e1, e2, e3) ->
    let v1 = évalue env e1 in
    match v1 with
    | Bool true  -> évalue env e2
    | Bool false -> évalue env e3
```

On notera que cette construction est particulière, dans le sens que l'evaluation d'une conditionnelle n'entraîne pas systématiquement l'évaluation de ses trois arguments. De fait l'une des expressions e_2 ou e_3 n'est pas évaluée. On parle parfois de construction *paresseuse*. On notera aussi que pour que les booléens aient un véritable intérêt il faut aussi se donner un certain nombre d'opérateurs supplémentaires par exemple l'inférieur ou égal :

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Bin}\,(\texttt{Le}\,, e_1, e_2) \Rightarrow v_1 \leq v_2}$$

Cela ne pose aucune difficulté si on voit $v_1 \leq v_2$ comme une notation traditionnelle pour l'application d'une certaine fonction du produit cartésien des entiers vers les booléens, de même que $v_1 + v_2$ est la notation traditionnelle d'une certaine fonction du produit cartésien des entiers vers les entiers.

Un point remarquable est le statut de la disjonction et de la conjonction (les « ou » et « et » logique). *A priori* on a envie de se simplifier la vie en les voyant comme des fonctions bien connues du produit cartésien des booléens vers les booléens. On imagine sans peine les règles associées. Mais, dans la plupart des langages de programmation ils ont une sémantique plus subtile dite *paresseuse*. Voici par exemple la sémantique du « et ».

$$\frac{\rho \vdash e_1 \Rightarrow false}{\rho \vdash \texttt{Bin}\,(\texttt{And}\,, e_1, e_2) \Rightarrow false} \qquad \frac{\rho \vdash e_1 \Rightarrow true \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Bin}\,(\texttt{And}\,, e_1, e_2) \Rightarrow v_2}$$

La différence entre cette sémantique paresseuse et la sémantique précédente (dite *stricte*) apparaît quand e_1 vaut faux et que l'evaluation de e_2 déclenche une erreur (voir la section suivante à ce sujet). Avec la sémantique stricte l'evaluation de la conjonction échoue, avec la sémantique paresseuse, l'evaluation rend faux. La sémantique paresseuse est en général préférée précisément à cause de cette propriété : on a plus de programmes corrects pour un prix modique. On notera que la sémantique paresseuse revient à comprendre « e_1 `&&` e_2 » comme « `if` e_1 `then` e_2 `else false` ».

Notons que la conditionnelle peut aussi être définie comme une instruction, il n'y a aucune difficulté particulière. On peut alors facilement omettre le `else` de la conditionnelle.

3.4.4 Formalisation des erreurs

L'évaluation peut mal se passer, même dans la calculette. Par exemple, lors d'une division par 0, de l'accès à une variable non liée, ou de l'evaluation de de la condition d'un `if`, si le résultat n'est pas un booléen mais un entier. Dans la formalisation de Pseudo-Pascal, l'accès à une variable non initialisée est aussi une erreur.

Si on souhaite formaliser les erreurs, on peut remplacer la relation $\rho \vdash e \Rightarrow v$ par une relation $\rho \vdash e \Rightarrow r$ où r est une réponse. Les réponses sont l'union des valeurs v ou des erreurs z.

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad v_2 \neq 0}{\rho \vdash \texttt{Bin}\,(\texttt{Div}\,, e_1, e_2) \Rightarrow v_1/v_2} \qquad \frac{\rho \vdash e_2 \Rightarrow 0}{\rho \vdash \texttt{Bin}\,(\texttt{Div}\,, e_1, e_2) \Rightarrow \texttt{Division}}$$

(Il faudrait aussi ajouter d'autres règles pour propager les erreurs, mais c'est assez lourd.)

Le type des erreurs peut être un type somme afin de distinguer les diverses causes d'échec.

```
type erreur =
  Division_par_zéro | Variable_libre of string | Type
```

Les règles conduisent naturellement à définir les résultats comme un type somme des valeurs et des erreurs

```
type résultat = Valeur of valeur | Erreur of erreur
```

En pratique, on identifie résultats et valeurs et on utilise les exceptions de Caml.

```
exception Erreur of erreur

let erreur x = raise (Erreur x)

let cherche x l =
  try List.assoc x l
  with Not_found -> erreur (Variable_libre x)

let valeur_to_int = function
| Int i -> i
| _     -> erreur Type


let rec évalue env = function
    ...
  | Bin (Div, e1, e2) ->
      let v2 = valeur_to_int (évalue env e2) in
      if v2 = 0 then erreur Division_par_zéro
      let v1 = valeur_to_int (évalue env e1) in
      else Int (v1 / v2) ;;
```

Réciproquement, lorsque l'on s'intéresse surtout à la formalisation des calculs sans erreurs, on peut ne donner que les règles définissant les calculs et considérer que toute erreur se traduit par un arbre *bloqué*, c'est à dire par que la tentative de preuve du jugement d'évaluation échoue parce qu'une règle spécifique ne peut s'appliquer en raison de la fausseté clairement identifiable de l'une de ses prémisses. Typiquement, pour la division par zéro ce sera la prémisse $v_2 \neq 0$.

Dans le cadre de la compilation, il convient d'abord de faire la part des erreurs que le compilateur peut prévenir. Ainsi, le compilateur peut detecter les erreurs de type contenus dans le programme qui lui est proposé et refuser de le compiler, ou dans certains cas (C par exemple) au moins produire un avertissement. Mais toutes les erreurs potentielles ne peuvent pas être détectées par le compilateur. C'est par exemple le cas des divisions par zéro.

Pour comprendre ce qui peut se passer concrètement lors de l'execution d'une erreur vis à vis de la sémantique, supprimons tout contrôle des types préalable et considérons notre interpréteur. Les erreurs peuvent correspondre à des vérifications explicites lors de l'exécution ou pas. Prenons par exemple le cas d'une addition $true + 1$, notre sémantique interdit cette addition et notre interpréteur échoue en dénonçant une erreur de typage (détectée à l'exécution). Notre interpreteur réagit de cette façon sa représentation des valeurs distingue entiers et booléens. Un autre interpréteur pourrait très bien représenter les booléens à l'aide des entiers (0 pour *false* et 1 pour *true*). Un tel interpréteur évalue $true + 1$ comme 2, résultat qui n'a aucun sens. On notera que le code machine produit par un compilateur se comportera généralement comme ce second interpréteur.

Certaines erreurs risquent fort d'être fatales. Ansi, considérons l'expressiom 1[0] (accès à la première case de 1) et imaginons un compilateur produisant du code machine et démuni de contrôle de type. Lors de l'exécution, le code va tenter de lire le contenu de l'addresse mémoire 1, l'erreur fatale immédiate est assurée, car l'accès à la zone base de la mémoire est généralement interdite.

En conclusion de ces quelques reflexions, retenons que, si la sémantique peut parfois faire l'économie de la formalisation des erreurs, un compilateur ne peut pas les ignorer totalement. Il doit également tenter de les signaler le plus précisément possible, afin de renseigner le programmeur. La combinaison d'un typage statique (le compilateur rejette les programmes mal typés) ou dynamique (le système d'exécution contrôle les types) et de quelques vérifications à l'exécution (principalement les accès dans les tableaux) est à mon avis un minimum.

3.4.5 Terminaison

L'évaluation d'un programme peut ne pas terminer. En sémantique dénotationnelle, le domaine sémantique contient alors une valeur particulière (généralement notée $\perp$) qui modélise entre autre cet état de fait. En fait $\perp$ est le point minimum des treillis, il représente l'absence d'information. Notons que le formalisme des treillis permet d'exprimer des valeurs partiellement connues. Par exemple la paire $(\perp, \perp)$ n'est pas forcément $\perp$, on sait au moins que c'est une paire, alors qu'avec $\perp$, on ne sait rien du tout.

En SOS à grands pas, c'est à dire en SOS telle que nous l'avons vue jusqu'ici, on ne peut pas modéliser la non-terminaison. En effet, ce formalisme ne sait parler que des programmes qui terminent. On identifie alors tous les programmes qui ne terminent pas, en quelque sorte par défaut. Pourtant, et c'est particulièrement vrai des programmes impératifs, des programmes peuvent ne pas terminer et ne pas faire tous la même chose, on peut donc souhaiter les distinguer dans la sémantique.

Une façon de procéder est de définir une SOS dite « *à petits pas* ». Le calcul est modélisé par une relation de réduction interne, *i.e.* les programmes se réduisent sur eux mêmes (chaque étape élémentaire du calcul est modélisée par un petit pas de réduction) jusqu'à obtention d'une valeur éventuelle. (Voir le cours *Sémantique des langages de programmation*).

3.4.6 Ordre d'évaluation

Le formalisme SOS semble ne pas spécifier l'ordre d'évaluation des arguments disons d'une addition :

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \mathtt{Bin}\,(\mathtt{Plus}\,, e_1, e_2) \Rightarrow v_1 + v_2}$$

Cette écriture nous dit que pour calculer v, il faut calculer v_1 et v_2, elle ne dit pas qu'il faut calculer v_1 avant v_2 ou le contraire.

En revanche, la règle de la liaison dit bien dans quel ordre calculer, en raison de la dépendance explicite sur v_1.

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho, x \mapsto v_1 \vdash e_2 \Rightarrow v_2}{\rho \vdash \mathtt{Let}\,(x, e_1, e_2) \Rightarrow v_2}$$

De même, dans la formalisation impérative, l'ordre d'exécution des instructions est toujours fixé, en raison des dépendances dues à la mémoire σ. Il en sera de même si la mémoire apparaît dans les règles d'évaluation des expressions. Dans ce cas, l'ordre choisi est le plus souvent de la gauche vers la droite.

Dans les cas les plus simples comme celui de la calculette, l'ordre d'évaluation n'est de toute façon pas observable, le résultat est le même quelque soit l'ordre d'évaluation. Ce n'est plus le cas lorsque le langage est suffisamment expressif pour que l'on puisse écrire des programmes qui ne terminent pas et que des erreurs peuvent se produire. Par exemple en Caml :

```
let rec loop () = loop () in
1/0 + loop ()
```

Si les arguments de l'addition sont évalués de la gauche vers la droite, alors le programme ci-dessus échoue à cause de la division par zéro, dans le cas droite-gauche, le programme ne termine pas.

Remarquons que, en cas de formalisation des erreurs, la sémantique SOS tend à spécifier l'ordre d'évaluation de façon un peu implicite. Ainsi la règle de la division par zéro peut s'exprimer de deux façons :

$$\frac{\rho \vdash e_2 \Rightarrow 0}{\rho \vdash \texttt{Bin}(\texttt{Div}, e_1, e_2) \Rightarrow \texttt{Division}} \qquad \frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow 0}{\rho \vdash \texttt{Bin}(\texttt{Div}, e_1, e_2) \Rightarrow \texttt{Division}}$$

L'évaluation est de droite à gauche dans le premier choix et de gauche à droite dans le second. Notons tout de même que la SOS est censée définir une relation entre expressions et résultats et non pas une fonction. Dès lors, on pourrait théoriquement associer plusieurs résultats à un programme donné, par exemple en se donnant les deux règles de la division. Mais ça ne se fait pas trop. . .

En dernière analyse, l'implémentation d'un langage devra bien évaluer les expressions dans un certain ordre. On peut peut-être considérer que ça n'a pas trop d'importance dans le cas des erreurs et de la non-terminaison, mais, en pratique, l'effet de l'ordre d'évaluation des expressions est notable lorsque l'on procède à des effets de bord dans les expressions. Se pose alors la question de savoir si cet ordre est fixé par la sémantique (même informelle) ou laissé à l'appréciation de l'implémenteur. La tendance actuelle est de spécifier l'ordre d'évaluation dans la sémantique (cf. Java), l'avantage est que toutes les implémentations conformes à la sémantique traitent tous les programmes de la même façon. L'avantage de ne pas spécifier l'ordre est que l'implémenteur peut profiter de la liberté qui lui est donnée pour introduire des optimisations (cf. C et Caml). Dans ce cadre l'ordre d'évaluation n'est bien spécifié que pour un certain nombre de constructions (typiquement la séquence, la liaison `let`).

Mon avis sur la question

De toute façon, les programmes qui terminent, sont exempts d'erreurs et bien écrits (de mon point de vue) ne sont pas concernés. Utiliser l'ordre d'évaluation des arguments par exemple de l'addition n'est certainement pas de la bonne programmation, car le programme fonctionne alors un peu par miracle. Chaque fois que cet ordre est important il vaut mieux l'exprimer clairement en décomposant à l'aide de la séquence ou du `let`. Pour lire un entier décimal sous forme de deux caractères pris dans l'entrée standard, on évitera d'écrire :

En C :

```
x = 10 * (getchar() - '0') +
     getchar() - '0' ;
```

En Caml :

```
let input_digit () =
  Char.code (input_char stdin) -
  Char.code '0' in
10 * input_digit () + input_digit ()
```

On écrira plutôt :

En C :

```
c1 = getchar() - '0' ;
c2 = getchar() - '0' ;
x = 10 * c1 + c2 ;
```

En Caml :

```
let c1 = input_digit () in
let c2 = input_digit () in
10 * c1 + c2
```

Ainsi il apparaît clairement que le chiffre des dizaines vient avant celui des unités, et je recommande ce style y compris en Java où la première écriture est correcte.

Enfin, une conséquence un peu surprenante de l'indétermination de l'ordre d'évaluation en Caml est qu'il est possible de laisser l'ordre d'évaluation non spécifié dans l'interpréteur. Par exemple, on avait écrit :

```
let rec évalue env = function
    ...
| Bin (op, e1, e2) ->
    let v1 = évalue env e1 and v2 = évalue env e2 in
    begin match op with
    | Plus -> v1 + v2 | Moins -> v1 - v2
    | Mult -> v1 * v2 | Div -> v1 / v2
    end
```

Comme, dans l'expression `let` d_1 `and` d_2 `in` e, rien n'est sûr sur l'ordre d'évaluation respectif de d_1 et de d_2, le programme interpréteur ne dit rien sur l'ordre de calcul respectif de v_1 et de v_2.

3.4.7 Tableaux

Nous étudions ici les tableaux impératifs, c'est à dire munis d'une instruction d'affectation. Nous supposerons pour simplifier que les tableaux sont alloués dynamiquement et explicitement (leur durée de vie est infinie).

On ajoute trois constructions à la syntaxe :

— L'expression `Alloc` (e_1) alloue un nouveau tableau de taille la valeur de e_1 les cases du nouveau tableau sont indéfinies

— L'expression `Lire` (e_1, e_2) lit la case (correspondant à la valeur de) e_2 du tableau (la valeur de) e_1.

— L'instruction `Ecrire` (e_1, e_2, e_3) écrit la case e_2 du tableau e_1 avec la valeur de e_3.

Soit encore :

```
type expression = ...
| Alloc of expression
| Lire of expression * expression

and instruction = ...
| Ecrire expression * expresssion * expression
```

Avec pour syntaxe concrète :

expression	::=	
	|	**alloc (** *expression* **)**
	|	*expression* [*expression*]
instruction	::=	...
	|	*expression* [*expression*] **:=** *expression*

On peut modéliser les tableaux en tant que valeurs tout simplement par des adresses. On doit alors se donner un minimum d'arithmétique sur les adresses (l'adresse de la case i du tableau t est l'adresse $t + i$). C'est bien ainsi que les tableaux sont implémentés dans les langages comme C ou Pascal. Pour la sémantique, voyons donc plutôt un tableau t comme une fonction d'un intervalle entier dans les adresses, que l'on note comme les environnements. On se donne aussi une valeur invalide notée $\perp$. À cause du parti-pris un peu arbitraire de ne pas modifier la mémoire dans l'évaluation des expressions, l'allocation ne peut apparaître disons que lors d'une affectation :

Création

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow k \qquad k \geq 0 \qquad t = (0 \mapsto \ell_0, \ldots, k-1 \mapsto \ell_{k-1}) \qquad \ell_0 \notin dom\,(\sigma) \ldots\ \ell_{k-1} \notin dom\,(\sigma)}{\rho/\sigma \vdash \texttt{Affecte}\,(x, \texttt{Alloc}\,(e_1)) \Rightarrow /\sigma, l_0 \mapsto \perp, \ldots l_{k-1} \mapsto \perp, \rho(x) \mapsto t}$$

Lecture

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow t \qquad \rho/\sigma \vdash e_2 \Rightarrow k \qquad k \in dom\,(t)}{\rho/\sigma \vdash \texttt{Lire}\,(e_1, e_2) \Rightarrow \sigma(t(k))}$$

Écriture

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow t \qquad \rho/\sigma \vdash e_2 \Rightarrow k \qquad k \in dom\,(t) \qquad \rho/\sigma \vdash e_3 \Rightarrow v}{\rho/\sigma \vdash \texttt{Ecrire}\,(e_1, e_2, e_3) \Rightarrow /\sigma, t(k) \mapsto v}$$

Notons que les règles exhibent de nombreuses conditions explicites, outre la contrainte de fraîcheur des adresses ($l_0 \notin dom\,(\sigma)$, etc.) qui est idéalement toujours satisfaite, on remarque des conditions de bon typage (les indices sont des entiers) et des conditions liées aux bornes des tableaux ($k \geq 0$ et $k \in dom\,(t)$) que le typage ne détecte pas en général.

Pour l'interpréteur, on profite des tableaux mutables du langage hôte.

```
type valeur = Int of int | Array of valeur array | Undefined

type erreur = ... | Type | Index
```

En raison de la représentation des valeurs, l'interpréteur signalera nécessairement les erreurs de type.

```
let array_of_valeur = function
| Array t -> t
| _       -> erreur Type
```

En outre, parce que les accès en dehors des bornes d'un tableau sont détectés par Caml, l'interpréteur signale nécessairement ce type d'erreur. Le code suivant procède toutefois à une vérification explicite, afin de maîtriser le signalement des erreurs (sinon on aurait des exceptions `Invalid_Argument of string`).

```
let rec évalue env = function
    ...
  | Alloc (e1) ->
      let k = int_of_valeur (évalue env e1) in
      if k >= 0 then
        Array (Array.create k Undefined)
      else
        erreur Index
  | Lire (e1, e2) ->
      let t = array_of_valeur (évalue env e1)
      and k = int_of_valeur (évalue env e2) in
      if 0 <= k && k < Array.length t then
        t.(k)
      else
        erreur Index

and execute env = function
    ...
| Ecrire (e1, e2, e3) ->
    let t = array_of_valeur (évalue env e1)
    and k = int_of_valeur (évalue env e2)
    and v = évalue env e3 in
    if 0 <= k && k < Array.length t then
        t.(k) <- v
      else
        erreur Index
```

On notera que comme Caml vérifie les accès aux tableau (et lève l'exception `Invalid_Argument` **of** `string` en cas d'accès hors-bornes), on aurait pu se passer de test explicite sur les indices.

3.5 Les fonctions

3.5.1 Les fonctions globales

Dans un langage comme C (ou Pseudo-Pascal) les fonctions ne peuvent être définies qu'au niveau ≪ *supérieur* ≫ (*top-level*), c'est à dire globalement. Un programme complet est tout simplement une suite de de définitions de fonctions. Ensuite en C il existe une fonction de nom conventionnel (`main`) et l'execution du programme est un appel de cette fonction. Tandis qu'en Pseudo-Pascal le programme comprend aussi en plus de la suite de définitions une instruction à exécuter.

Soyons plus précis en considérant une calculette fonctionnelle. Le programme est tout simplement une liste de définitions de fonctions à un argument, plus une expression à évaluer. Une fonction de nom f est donc une association $f \mapsto (x, e)$, où x est le nom du paramètre de la fonction et e est son corps.

```
type fonction = Fun of string * expression

type fenvironnement = (string * fonction) list

type programme = (string * fonction) list * expression
```

La syntaxe abstraite des expressions est étendue par une construction d'application :

expression	::=	...
	|	**VARIABLE** (*expression*)

```
type expression = ...
| App of string * expression
```

Ainsi sans rentrer exagérément dans les détails, le calcul de 10! avec une telle calculette pourrait s'écrire :

```
fact(x) = if x=0 then 1 else x * fact (x-1) ;;

fact(10)
```

Pour tenir compte des définitions de fonctions dans la sémantique on peut considérer l'évaluation par rapport à deux environnements ρ_f pour les fonctions et ρ pour les valeurs ordinaires. La sémantique de l'application est alors :

$$\frac{\rho_f(f) = \mathtt{Fun}\,(x, e_f) \qquad \rho_f;\ \ \rho \vdash e \Rightarrow v_a \qquad \rho_f;\ \ (x \mapsto v_a) \vdash e_f \Rightarrow v}{\rho_f;\ \ \rho \vdash \mathtt{App}\,(f, e) \Rightarrow v}$$

La définition informelle serait que l'évaluation de l'application d'une fonction est l'evaluation de son corps dans un environnement limité à la liaison de son paramètre à la valeur de l'argument.

L'écriture de l'interpréteur serait alors :

```
let rec évalue fenv env = function
        ...
| App (f, e) ->
    let va = évalue fenv env e in
    let Fun (x,ef) = cherche fenv f in
    évalue fenv [(x,va)]) ef
```

L'environnement ρ_f ne change jamais au cours de l'évaluation, il est en quelque sorte global. Il est conceptuellement simple d'imaginer qu'il contient, en plus des fonctions, toutes les liaisons globales d'un programme. Pour ce qui est de la compilation, on notera qu'un programme ne comporte qu'un nombre fini de fonctions et de variables globales bien connues, et que donc le compilateur peut allouer statiquement l'espace à eux nécessaire. Par contraste, les autres liaisons (celles des arguments et des variables locales) sont réalisées en pile, l'espace nécessaire est réservé au moment de l'appel des fonctions et rendu au moment de leur retour. Cela est correct, car on peut plus accéder aux valeurs des arguments et des variables locales d'une fonction une fois que cette fonction a retourné.

Il est important de remarquer que les fonctions telles que décrites ci-dessus sont assez limitées. À part des noms de fonctions (ou des variables globales), la seule variable qui peut apparaître dans le corps de la fonction est le paramètre. Bien sûr, en pratique, il peut y avoir plusieurs paramètres et la fonction peut également déclarer des variables locales. On obtient ainsi par exemple les fonctions du langage C.

En Caml les fonctions sont plus puissantes dans le sens que d'autres variables peuvent apparaître dans le corps de fonction :

```
let f x =
  let g y = x + y in
  g 1
```

Dans le code ci-dessus, la variable y apparaît dans le corps de la fonction g, bien que ce ne soit pas un paramètre de g. Dans le même ordre d'idée, en Java, les champs d'un objet peuvent apparaître dans le corps des méthodes.

3.5.2 Appel par valeur

L'appel des fonctions de la section précédente est par valeur. C'est à dire que l'appel de fonction crée une nouvelle liaison entre le nom de l'argument (*paramètre formel*) et la valeur de l'argument (*paramètre effectif*). Il est intéressant de modéliser le même comportement dans le cas impératif (section 3.4.2), en présence d'une construction d'affectation. Une variable est maintenant réalisée par une double liaison, des noms aux adresses, puis des adresses aux valeurs (ρ et σ). La règle de l'appel par valeur réclame l'allocation d'une nouvelle case mémoire d'adresse ℓ_a pour y ranger la valeur de l'argument.

$$\frac{\rho_f(f) = (x, e_f) \qquad \rho_f;\ \rho/\sigma \vdash e \Rightarrow v_a \qquad \ell_a \notin dom(\sigma) \qquad \rho_f;\ (x \mapsto \ell_a)/\sigma, \ell_a \mapsto v_a \vdash e_f \Rightarrow v}{\rho_f;\ \rho/\sigma \vdash \texttt{App}\,(f, e) \Rightarrow v}$$

Cette écriture explicite la création d'une nouvelle adresse ℓ_a. On peut aussi voir que la nouvelle adresse ℓ_a n'est accessible que lors de l'évaluation du corps e_f. Cela suggère fortement la réalisation de la liaison entre paramètre formel et paramètre effectif à l'aide d'une pile.

Mais pourquoi alors ne pas créer simplement une nouvelle liaison de variable à adresse, sans créer de nouvelle liaison d'adresse à valeur ? Cela n'a une signification simple que le paramètre effectif possède clairement une adresse, c'est à dire que lorsque le paramètre effectif est une variable. On pourrait alors écrire cette règle :

$$\frac{\rho_f(f) = (x, e_f) \qquad \rho(y) = l_a \qquad \rho_f;\ (x \mapsto \ell_a)/\sigma \vdash e_f \Rightarrow v}{\rho_f;\ \rho/\sigma \vdash \texttt{App}\,(f, y) \Rightarrow v}$$

En Pascal, on spécifie qu'un paramètre est passé par variable en utilisant le mot-clé **var** lors de la définition des fonctions. Par défaut le passage est par valeur. Soient par exemple les deux programmes complets suivant.

Passage par valeur.

```
program un ;
var x : integer ;

procedure p (y : integer) ;
begin
   y := 2
end ;

begin
  x := 1 ;
  p(x) ;
  writeln(x)
end.
```

Passage par variable.

```
program deux ;
var x : integer ;

procedure p (var y : integer) ;
begin
   y := 2
end ;

begin
  x := 1 ;
  p(x) ;
  writeln(x)
end.
```

Le premier programme affiche 1 tandis que le second affiche 2. L'appel par variable fait que `x` et `y` sont deux synonymes de la même case mémoire, on parle alors d'*alias*.

Les paramètres tableaux

Dans le cadre de Pseudo-Pascal, la règle de l'appel de fonction est la règle d'appel par valeur des langages impératifs donnée au début de cette section, que nous rappelons :

$$\frac{\rho_f(f) = (x, e_f) \qquad \rho_f;\ \rho/\sigma \vdash e \Rightarrow v_a \qquad \ell_a \notin dom(\sigma) \qquad \rho_f;\ (x \mapsto \ell_a)/\sigma, l_a \mapsto v_a \vdash e_f \Rightarrow v}{\rho_f;\ \rho/\sigma \vdash \texttt{App}\,(f, e) \Rightarrow v}$$

Si v_a est un tableau $t = (0 \mapsto l_0, \ldots, k-1 \mapsto l_{k-1})$ alors une nouvelle liaison de x à l_a puis à t est créée, comme dans le cas général. Le point important est que les adresses $l_0, \ldots, l_{k-1}$ ne changent pas, ainsi une modification du tableau à l'intérieur de la fonction sera visible après le retour de celle-ci. En revanche, une affectation de la variable x reste toujours invisible de l'extérieur. Le tableau est bien passé par valeur, mais c'est sémantiquement une référence (nom abstrait des adresses) vers une structure mutable (une zone de mémoire où on peut écrire). On peut faire le lien avec Java par exemple, où les objets sont des références, mais aussi avec C, où les tableaux sont expressément définis comme (presque) équivalents à l'adresse en mémoire de leur première case.

Pascal procède différemment. Si une fonction prend un paramètre tableau, alors le paramètre formel sera lié à une *copie* du tableau effectif. C'est à dire que sémantiquement les tableaux ne sont plus des références vers des zones mémoire, mais bien des zones mémoire elles-mêmes. En outre, si le mot-clé **var** précède la définition du paramètre formel, celui ci sera en fait une référence vers la zone mémoire du tableau effectif et les modifications des cases du tableau porteront sur les cases du tableau effectif.

3.5.3 Culture : Fonctions de première classe

Si l'on souhaite que les fonctions soient valeurs du langage au même titre que les entiers, on va se retrouver confronté à un problème sémantique. L'origine de ce problème réside dans les variables dites *libres* des fonctions c'est à dire dans les variables qui apparaissent dans le corps des fonctions et qui ne sont ni des arguments, ni des variables locales. Considérons cet exemple de Caml :

```
let x = 1                (* env1 : x est lié à une valeur v  *)
;;

let rec mem = function [ ] -> false | h::t -> x = h || mem t
;;

let x = 2                (* env2 : x est lié à la valeur w   *)
;;

mem l
;;
```

La variable `x` est libre dans la définition de `mem` et possède deux définitions. Doit-on lorsque l'on évalue le corps de `mem` choisir la liaison de `x` à 1 ou celle de `x` à 2 ? Si l'on fait le choix de la liaison valide au moment de la définition de la fonction `mem`, alors on parle de liaison *lexicale* (C, Pascal, Caml). Si on fait le choix de de la liaison valide au moment de l'appel de la fonction `mem`, alors on parle de liaison *dynamique* (vieux Lisp, Basic).

La liaison lexicale est préférée parce qu'elle permet de toujours de connaître la déclaration d'une variable libre sans appeler la fonction, c'est à dire qu'un compilateur sait toujours ou aller chercher sa valeur en mémoire, quel est son type etc. Mais aussi, un programmeur sait toujours de quelle ≪ *variable* ≫ il s'agit et la robustesse des programmes s'en trouve augmentée.

Notons que si notre langage de programmation permet de rendre des fonctions comme des résultats de fonctions, la liaison lexicale est obligatoire, voici par exemple la définition de la composition de fonction en Caml :

```
let compose f g =
  let h x = f (g x) in
  h
```

On ne voit pas très bien ce que pourrait signifier la liaison dynamique dans ce cas. En revanche, le sens de la liaison lexicale est clair : au moment de la définition de la fonction `h`, les valeurs de `f` et `g` (libres dans la définition de `h`) sont bien connues. Un appel ultérieur à `h` doit savoir les retrouver.

La solution la plus simple et la plus générale pour implémenter les fonctions de première classe est la *fermeture*. Une fermeture est une paire d'un environnement et du corps de cette fonction, l'environnement sera pris comme étant celui qui est valide au moment de la définition de la fonction.

Construisons donc une calculette authentiquement fonctionnelle, à partir de ce principe. Le plus simple est de considérer des fonctions anonymes et une construction d'application plus générale que précédemment :

$$
\begin{array}{lcl}
\textit{expression} & ::= & \ldots \\
 & | & \textbf{fun VARIABLE ->}\ \textit{expression} \\
 & | & \textit{expression}\ (\textit{expression}\ )
\end{array}
$$

```
type expression = ...
| Fun of string * expression
| App of expression * expression
```

Les valeurs doivent maintenant comprendre les fermetures. En SOS, on peut les noter $\langle x, \rho, e\rangle$ (x est le paramètre), et pour les valeurs de l'interpréteur on peut écrire :

```
type valeur = Int of int | Fermeture of string * environnement * expression
```

Les règles de création de fermeture et d'application sont alors :

$$
\rho \vdash \mathtt{Fun}\,(x, e) \Rightarrow \langle x, \rho, e\rangle \qquad \frac{\rho \vdash f \Rightarrow \langle x, \rho_f, e_f\rangle \qquad \rho \vdash e \Rightarrow v_a \qquad \rho_f, x \mapsto v_a \vdash e_f \Rightarrow v}{\rho \vdash \mathtt{App}\,(f, e) \Rightarrow v}
$$

Et le code de l'interpréteur est :

```
let rec évalue env = function
      ...
| Fun (x, e) -> Fermeture (x, env, e)
| App (f, e) ->
    let vf = évalue env f
    and va = évalue env e in
    match fv with
    | Fermeture (x, envf, ef) ->
        évalue (ajoute x va envf) ef
```

On notera le grand intérêt de l'idée de la fermeture. L'expressivité du langage est considérablement augmentée par rapport aux fonctions globales et la sémantique n'est pas vraiment plus compliquée.

On peut se demander alors pourquoi tous les langages n'offrent pas les fonctions de première classe. Je crois que la réponse tient principalement à des questions d'implémentation. Dans le cadre de la compilation, le seul problème de cette approche des fonctions par les fermetures est que les liaisons des variables ne peuvent plus, en toute généralité, être réalisées statiquement ou en pile. En effet, dans l'exemple de la fonction `compose` les liaisons de `f` et `g` survivent à l'appel de `compose` puisqu'elles existent encore lors de l'appel ultérieur de `h`. Le code du corps de la fonction `h` ne pourra donc pas aller chercher les valeurs de `f` et `g` dans la pile là où elles se trouvaient normalement au moment de la création de `h` (ce sont des arguments de `compose`) Il faut donc allouer les environnements dynamiquement et cette allocation est implicite, ce qui à son tour impose de disposer d'un glaneur de cellules (*garbage collector*) chargé de récupérer automatiquement la mémoire qui n'est plus utilisée.

On peut toutefois faire les deux remarques suivantes :

— L'environnement des fermetures n'a pas besoin de contenir des liaisons pour tout le domaine valide au moment de la création de la fermeture, contrairement à ce que semble affirmer notre sémantique. On peut se contenter des seules variables effectivement libres dans la fonction créée. L'avantage est que les environnements des fermetures seront plus petits et que la mémoire est plus efficacement gérée.

— Si il est interdit de rendre les fonctions comme résultat, alors l'environnement peut être réalisé en pile uniquement. Pascal procède de cette façon : les fonctions peuvent être définies dans d'autres fonctions (et alors les liaisons des variables libres ne peuvent pas toutes être allouées statiquement par le compilateur), et passées comme arguments. La description exacte des techniques d'implémentation nécessaires dépasse un peu le cadre du cours, mais on comprendra que dans le cadre de cette restriction, *toutes* les valeurs des variables libres se trouvent bien quelque part dans la pile, dans la portion de pile allouée lors de l'appel des fonctions qui les définissent, appels qui *tous* n'ont pas encore retourné.

Une implémentation possible de ce type de fonctions consiste à supprimer les variables libres en les remplaçant par des arguments supplémentaires (attention, ce n'est pas ainsi que Pascal procède traditionnellement). Ainsi les deux programmes suivants (en syntaxe Caml) font la même chose.

La variable `x` est libre dans `mem`.

```
let member (x,l) =
  let rec mem l = match l with
    | [] -> false
    | h::t -> x = h || mem t
  in mem l
```

La variable `x` est un argument de `mem`.

```
let member (x,l) =
  let rec mem (x,l) = match l with
  | []  -> false
  | h::t -> x = h || mem (x,t) in
  mem (x,l)
```

On notera que la transformation du programme de gauche en celui de droite n'est possible que parce que tous les appels de `mem` sont connus.

3.6 Le langage Pseudo-Pascal (PP)

C'est un Pascal simplifié et subtilement modifié en ce qui concerne les tableaux.

3.6.1 Syntaxe

La syntaxe concrète est (presque) celle de Pascal.

La syntaxe abstraite est définie en Caml. On conserve les informations de types (pour permettre une analyse de type ultérieure)

```
type type_expr = Integer | Boolean | Array of type_expr;;
```

Un programme est composé d'un ensemble de déclarations de variables, de fonctions et d'un corps (une instruction).

```
type var_list = (string * type_expr) list
type program = {
  (* variables globales *)
    global_vars : var_list;
  (* procédures et fonctions *)
    definitions : (string * definition) list;
  (* corps du programme *)
    main : instruction; }

and definition = {
  (* arguments (avec leurs types) *)
    arguments : var_list;
  (* type du résultat (None pour une procédure) *)
    result : type_expr option;
  (* variables locales *)
    local_vars : var_list;
  (* corps de la fonction *)
    body : instruction;  }
```

Les expressions

```
and expression =
    (* constantes *)
  | Int of int | Bool of bool
    (* opérations binaires *)
  | Bin of binop * expression * expression
    (* accès à une variable *)
  | Get of string
    (* appel de fonction *)
  | Function_call of string * expression list
    (* accès dans un tableau à une position *)
  | Geti of expression * expression
    (* Création d'un tableau d'une certaine  taille  et d'un certain type *)
  | Alloc of expression * type_expr

and binop =
(* Arithmétique *)
  | Plus | Minus | Times | Div
(* Comparaisons (entre entiers) *)
  | Lt | Le | Gt | Ge | Eq | Ne
```

Les instructions

```
and instruction =
    (* Affectation d'une variable *)
  | Set of string * expression
    (* Suite d'instructions *)
  | Sequence of instruction list
  | If of expression * instruction * instruction
  | While of expression * instruction
    (* Appel de procédure *)
  | Procedure_call of string * expression list
    (* Ecriture d'un entier *)
  | Write_int of expression
    (* Lecture d'un entier dans une variable *)
  | Read_int of string
    (* Affectation dans un tableau *)
  | Seti of expression * expression * expression
```

3.6.2 Sémantique

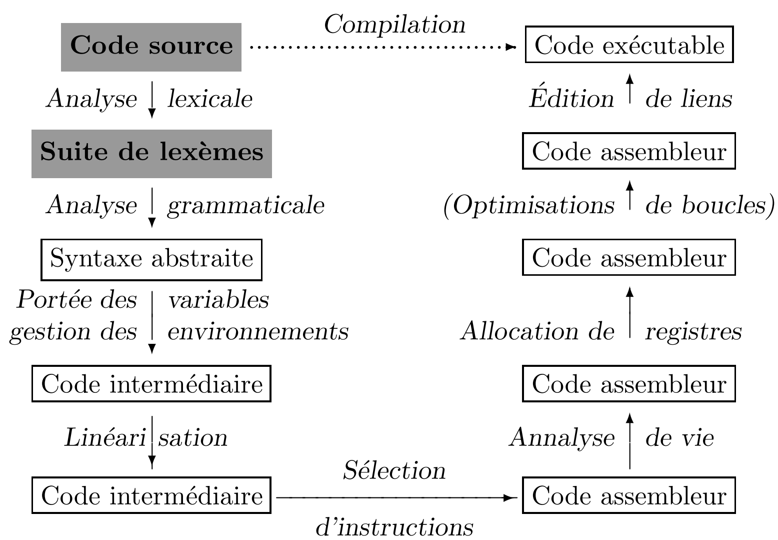
- Le langage est impératif et distingue les expressions et les instructions. Les constructions des instructions sont classiques (affectation, séquence, conditionnelle, boucle `while`), plus deux « primitives » pour écrire et lire un entier (`read_int` et `write_int`).
- Les valeurs sont les entiers, les booléens et les tableaux. Sémantiquement les tableaux sont des références. Les tableaux sont alloués dynamiquement (durée de vie infinie).
- L'ordre d'évaluation est de gauche à droite.
- Les fonctions sont globales, mutuellement récursives et ne peuvent qu'être appelées. Les fonctions se divisent en procédures qui ne rendent pas de résultat, et en fonctions proprement dites. L'appel de procédure est une instruction, tandis que l'appel de fonction est une expression. Notons que les corps des fonctions sont en fait une instruction et que, selon la convention de Pascal, le résultat d'une fonction est passé en affectant la variable homonyme (pas d'instruction `return`). Attention, toutes les règles vues pour les fonctions considéraient que le corps d'une fonction est une expression…

— Les arguments des fonctions sont passés par valeur.
— Les retours de fonction et les affectations suivent la même convention.
— Les variables sont toutes mutables (y compris celles de type tableau) et la portée est lexicale.
— Les variables sont déclarées (leur type est spécifié à ce moment), comme variables globales, variables locales ou arguments. La portée et la durée de vie est infinie pour les premières, limitée à un appel pour les autres.

C'est à vous d'écrire un interpréteur, en grapillant dans ce cours et en utilisant votre culture.

Chapitre 4

Analyse lexicale



L'analyse lexicale se trouve tout au début de la chaîne de compilation, elle collabore avec l'analyse grammaticale pour passer de la syntaxe concrète à la syntaxe abstraite. La mission de l'analyse lexicale est de transformer une suite de caractères en une suite de mots, dit aussi lexèmes (*tokens*). Procéder ainsi en deux temps, en reconnaissant d'abord les mots, puis les phrases, n'est pas justifié par la théorie. En effet, un analyseur grammatical est strictement plus puissant qu'un analyseur lexical et il pourrait reconnaître les mots. La justification est pratique, l'analyseur grammatical est bien plus facile à écrire une fois les mots reconnus.

4.1 Enjeux

La production d'un arbre (de syntaxe abstraite) à partir d'une suite de caractères se retrouve comme première passe dans de nombreuses applications (analyses des commandes, des requêtes, etc.).

Les deux analyses (lexicales et syntaxiques) utilisent de façon essentielle les automates, mais on retrouve aussi les automates dans de nombreux domaines de l'informatique. L'analyse lexicale s'explique dans le cadre restreint des automates finis et des expressions régulières (le terme français « autorisé » est expression rationnelle, mais je préfère adapter la terminologie anglaise). Les expressions régulières sont utilisées dans de nombreux outils Unix (éditeur de textes, commande `grep` etc.), et fournies en bibliothèque dans la plupart des langages de programmation.

Note L'étude détaillée des automates constitue un cours à part entière. Nous nous contentons ici de la présentation formelle minimale, avec comme but :

— d'expliquer le fonctionnement des analyseurs de façon à pouvoir écrire soi-même des analyseurs lexicaux ou grammaticaux,
— de se familiariser aussi avec les expressions régulières et les automates.

Le but du cours n'est pas d'écrire le moteur d'un analyseur, ni de répertorier toutes les techniques d'analyse, mais un peu de théorie ne nuit jamais (voir la section 4.6).

4.2 Les langages formels

On se donne un ensemble Σ appelé alphabet, dont les éléments sont appelés caractères. Un mot (sur Σ) est une séquence de caractères (de Σ). On note ϵ le mot vide, uv la concaténation des mots u et v (la concaténation est associative avec ϵ pour élément neutre). On note Σ^* l'ensemble des mots sur Σ.

Un langage sur Σ est un sous-ensemble L de Σ^*. On se donne quelques opérations sur les langages. Si U et V sont des langages sur Σ, on note UV l'ensemble des mots obtenus par la concaténation d'un mot de U et d'un mot de V ; U^* (resp. U^+), l'ensemble des mots obtenus par la concaténation d'un nombre arbitraire, éventuellement nul (resp. non nul) de mots de U.

4.2.1 Exemples

1. Σ_1 est l'alphabet français et L_1 l'ensemble des mots du dictionnaire français avec toutes leurs variations (pluriels, conjugaisons, etc.).
2. Σ_2 est L_1 et L_2 est l'ensemble des phrases grammaticalement correctes de la langue française. Un ensemble bien difficile à définir formellement. Ou bien, L_2' est le sous-ensemble des palindromes de L_2.
3. Σ_3 est l'ensemble des caractères ASCII, et L_3 est composé de tous les mots-clés de Pseudo-Pascal, de l'ensemble des symboles, de l'ensemble des identificateurs et de l'ensemble des entiers décimaux.
4. Σ_4 est L_3 et L_4 est l'ensemble des programmes Pseudo-Pascal.
5. Σ_5 est $\{a, b\}$ et L_5 est l'ensemble $\{a^n b^n \mid n \in \mathbb{N}\}$ (sous ensemble des expressions bien parenthésées).

4.3 Expressions régulières

La description de langages des mots (voir L_1 et L_3) qui servent à leur tour à définir des langages des phrases (voir L_2 et L_4) est relativement simple. On précise formellement cette simplicité en disant que ces langages des mots se décrivent à l'aide du formalisme relativement limité des *expressions régulières*.

On note a, b, *etc.* des lettres de Σ, M et N des expressions régulières, $[\![M]\!]$ le langage associé à M.

— Une lettre de l'alphabet a désigne le langage $\{a\}$.
— Epsilon : ϵ désigne le langage $\{\epsilon\}$.
— Concaténation : MN désigne le langage $[\![M]\!]\,[\![N]\!]$.
— Alternative : $M \mid N$ désigne le langage $[\![M]\!] \cup [\![N]\!]$.
— Répétition : M^* désigne le langage $[\![M]\!]^*$.

D'autres constructions sont utiles en pratique et exprimables à l'aide des précédentes :

— $[abc]$ pour $(a \mid b \mid c)$ et $[a_1{-}a_2]$ pour $\{c \in \Sigma, a_1 \leq c \wedge c \leq a_2\}$, en supposant que l'alphabet est ordonné.
— $M?$ pour $M \mid \epsilon$, et M^+ pour MM^*.
— $[\hat{}\,abc]$ désigne le complémentaire de $\{a, b, c\}$ dans Σ, vu comme des mots (et de même pour $[\hat{}\,a_1{-}a_2]$). L'interprétation est facile quand Σ est fini ce qui est toujours le cas.
— On a aussi parfois point « . » (ou *underscore* _) pour Σ et $*$ pour Σ^*. La première de ces construction est exprimable comme l'alternative de tous les caractères de l'alphabet, la seconde comme la répétition de la première.

Notons un point de vocabulaire. Lorsqu'un mot appartient à un langage défini par une expression régulière, on dit aussi l'expression (ici dénommée *motif*) *filtre* le mot.

Les *langages réguliers* sont ceux qui peuvent se définir à l'aide des expressions régulières. Dans nos exemples, L_1 est clairement régulier (alternative de tous les mots du dictionnaire, qui est fini) et nous allons voir comment exprimer L_3 avec des expressions régulières. C'est essentiellement l'absence de la récursion qui limite les langages réguliers, ainsi on peut montrer que le langage L_5 (les parenthèses) n'est pas régulier.

Le shell Unix utilise les expressions régulières pour spécifier les noms de fichiers. La commande suivante donne la liste de tous les sources Caml dans le répertoire courant :

```
#  ls *.ml{,[ily]}
```

Notez bien qu'ici l'alternative est exprimée avec la virgule « `,` », le mot vide par rien, et que les accolades sont un simple parenthésage. Dès lors, la commande précédente donne la liste des fichiers dont l'extension est `.ml`, `.mli`, `.mll` (sources du générateur d'analyseurs lexicaux `ocamllex`), et `.mly` (sources du générateur d'analyseurs syntaxiques `ocamlyacc`).

4.3.1 Utilisation pour l'analyse lexicale

Les lexèmes sont définis par l'alternative d'expressions régulières. Par exemple :

1. Les mots-clés : `"let"`, `"in"`. En général, les mots-clés ne peuvent pas être utilisés comme noms de variables. Ce parti pris évite pas mal d'ambiguïtés lors de l'analyse grammaticale ultérieure.
2. Les variables : `['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9']*`. Les noms des variables sont des suites de lettres et de chiffres commençant obligatoirement par une lettre.
3. Les entiers : `['0'-'9']+`
4. Les symboles : `'('`, `')'`, `'+'`, `'*'`, `'-'`, `'/'`,`'='`.
5. Un blanc : `[' ' '\n' '\t']`. Les caractères `'\n'` et `'\t'` sont respectivement le retour à la ligne et la tabulation.

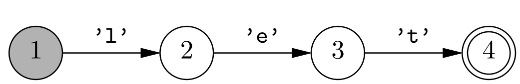
On est passé ici en notation Caml, où un caractère est donné entre quotes (et une suite de caractères entre double quotes), si on veut le caractère quote « `'` », il vaut mieux écrire `'\''`.

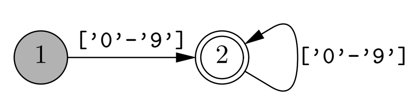
Une fois les lexèmes reconnus, ils sont représentés par un type somme dont nous noterons au passage qu'il n'est pas récursif :

```
type token =
  | LET | IN                          (* mots-clés *)
  | VAR of string                     (* variables *)
  | INT of int                        (* entiers *)
  | LPAR | RPAR | ADD | SUB | MUL | DIV | EQUAL (* symboles *)
```

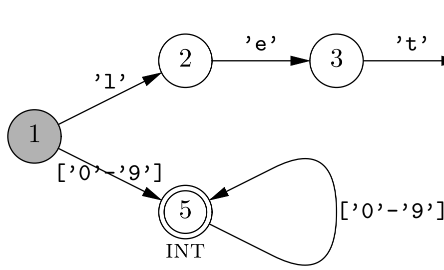
On notera aussi que les blancs sont omis, c'est à dire qu'ils sont peut être des mots du langage mais sont oubliés en route. De fait les blancs servent surtout à séparer les lexèmes.

Il est bien connu que les automates finis (j'en dis un peu plus par la suite) savent reconnaître les langages réguliers, c'est à dire qu'étant donné un langage régulier L on peut construire un automate qui, lorsque l'on lui présente un mot sait répondre par oui ou par non à la question de l'appartenance du mot à L. Très rapidement, l'automate est un graphe dont les sommets sont des états et les arcs des transitions, l'automate est à un instant donné dans un état donné et la consommation d'une lettre du mot le fait changer d'état en suivant une transition. Lorsque le mot est entièrement consommé le mot est reconnu si l'état courant est un état particulier dit final. Par exemple, voici deux automates finis qui reconnaissent respectivement le mot-clé `let` et les entiers (suite non vide de chiffres). Dans ces dessins, les état initiaux sont grisés et les états finaux encerclés.





Dans le second automate, chaque transition en remplace dix, si on suit le formalisme strict des automates. Ensuite, pour reconnaître le mot clé `let` ou un entier, on peut regrouper les deux automates précédents :



On a, ici dans un cas simple, construit l'automate qui reconnaît l'alternative de deux expressions régulières. Selon l'état final atteint (4 ou 5) on connaîtra le lexème reconnu.

Toutefois, ceci ne suffit pas tout à fait pour expliquer l'analyse lexicale, nous savons peut être reconnaître si un mot est dans L, mais nous devons, d'une part, reconnaître une suite de mots de L, et d'autre part, savoir de quels mots il s'agit. Intuitivement, un automate peut facilement reconnaître que le mot présenté est bien une suite de mots de L. En effet, le langage d'une suite de mots de L se définit à l'aide de l'opérateur de répétition *. Mais on ne sait pas alors quels mots ont été reconnus, il vaut mieux reconnaître les mots un par un. Conceptuellement, il suffit d'arrêter l'automate dans un état final sans attendre la fin du mot, puis de recommencer sur la partie non consommée du mot présenté.

Mais il y a encore des cas douteux :

— `let` pourrait être reconnu comme une variable.

— `lettre` pourrait aussi être reconnu comme la séquence `LET; VAR "tre"` ou encore comme la séquence `VAR "let"; VAR "tre"`.

Ces ambiguïtés se lèvent à l'aide de règles spécifiques. Lors de la reconnaissance d'un mot de L, on cherchera :

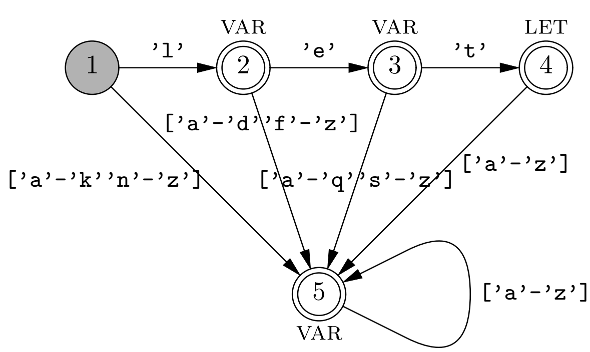
1. Le lexème le plus long possible.
2. Entre deux lexème de longueur maximale, l'ordre de présentation des sortes de mots lève l'ambiguïté, la première gagne.

Ainsi la phrase `let lettre = 3 in 1 + fin` devrait produire la suite de lexèmes :

```
LET; VAR "lettre"; EQUAL; INT 3; IN; INT 1; PLUS; VAR "fin"
```

En raison de la règle du lexème le plus long et à condition que, à taille égale, la reconnaissance des mots-clés prime celle des variables.

La règle de priorité numéro deux (sur l'ordre de présentation) se réalise simplement au moment de la fabrication de l'automate. Voici par exemple un automate qui reconnaît le mot-clé `let` ainsi que les identificateurs composés simplement de lettres minuscules, le mot-clé étant prioritaire.



L'état final 4 pourrait bien correspondre à une variable ou à `let`, on choisit de le faire correspondre à la reconnaissance du mot-clé.

Figure 4.1 – Analyseur lexical de la calculette

```
{
open Token

exception Error
}

rule token = parse
(* Les lexèmes stricto-sensu *)
| '('            {LPAR}
| ')'            {RPAR}
| '+'            {ADD}
| '-'            {SUB}
| '*'            {MUL}
| '/'            {DIV}
| '='            {EQUAL}
| "let"          {LET}
| "in"           {IN}
| ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9']*
                 {VAR (Lexing.lexeme lexbuf)}
| ['0'-'9']+     {INT (int_of_string (Lexing.lexeme lexbuf))}
(* Règles supplémentaires *)
| eof            {EOF}
| [' ' '\n' '\t' ] {token lexbuf}
| ""             {raise Error}
```

Sur cet exemple on peut aussi appréhender la réalisation de la règle du lexème le plus long. Tout en consommant les caractères de l'entrée, on peut se souvenir du dernier état final rencontré. Ensuite, lorsque l'automate est *bloqué*, ici par exemple si il y a un chiffre dans l'entrée, alors on peut revenir au dernier état final vu. Le blocage est facilement détecté en ajoutant un état dit bloqué à l'automate et en complétant les transitions issues de tous les états par des transitions vers l'état bloqué.

4.4 `ocamllex`

Nous ne savons pas précisément comment, à partir de la définition des lexèmes donnés comme expressions régulières, fabriquer l'automate qui les reconnaît. Nous admettons que cet automate existe. Bien mieux, il existe un programme `ocamllex` qui sait le construire pour nous.

L'outil `ocamllex` est lui même un compilateur, qui prend comme source les expressions régulières (dans un fichier *nom*`.mll`) et produit un programme Ocaml (dans un fichier *nom*`.ml`), programme qui réalise l'automate.

4.4.1 Un exemple simple

Commençons par un exemple, celui d'un analyseur lexical pour calculette avec `let`. On écrit le source de la figure 4.1 dans un fichier `lexer.mll`, la commande :

```
# ocamllex lexer.mll
```

produit un nouveau fichier `lexer.ml`. L'exemple suffit déjà pour expliquer pas mal de choses sur la structure des fichiers source de `ocamllex`.

1. Le source commence par du code source Caml entre accolades, `ocamllex` copie ce code quel au début du fichier `lexer.ml`, de même on peut mettre du code Caml à la fin. Ici, le code donné en prélude commence par ouvrir le module `Token`, qui est supposé contenir la définition interne des lexèmes (autrement dit il existe un fichier `token.mli` qui contient la définition de type de la section 4.3.1). Cela permet d'écrire par exemple `LPAR` au lieu de `Token.LPAR`.
2. Ensuite, on trouve la définition de l'automate (ici dénommé `token`) sous forme d'une suite de règles introduites par les mots-clés (de `ocamllex`) **rule** et **parse**. Chaque règle est constituée d'une expression régulière (le motif) et d'une action, du code Caml à exécuter après reconnaissance. Les premières règles de la parenthèse ouvrante au mot-clé (de la calculette) `in` ne posent pas de problème, l'action est de rendre le lexème reconnu.
3. Ça se complique un peu pour les identificateurs, on voit apparaître la variable `lexbuf` et la fonction `Lexing.lexeme` dans l'action. La première est en fait l'argument implicite de l'analyseur, *i.e.* la suite de caractère analysée, la seconde extrait la suite de caractères reconnue de `lexbuf` passé en argument. Le type des `lexbuf` est défini par le module `Lexing` de la bibliothèque standard, c'est une réalisation des suites de caractères qui répond aux besoins des analyseurs lexicaux. Dans ce cas l'entrée s'appelle aussi un *flux*. La règle des entiers s'explique de la même façon. En outre on convertit au passage la chaîne reconnue en entier.

   Notons qu'à partir de la version 3.07 de Caml, on peut écrire.

   ```
   | ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9']* as lxm
                       {VAR lxm}
   | ['0'-'9']+  as lxm {INT (int_of_string lxm)}
   ```

   C'est à dire qu'il est possible de lier la chaîne reconnue à une variable quelconque (ici `lxm`) à l'aide de la construction **as**.
4. Dans la règle suivante, le motif est le mot-clé (de `ocamllex`) **eof**. Cela indique la fin du flux d'entrée, l'automate rend alors un lexème spécifique qui aurait dû être ajouté à la définition des lexèmes de la section 4.3.1.
5. Ensuite, vient la règle de reconnaissance des blancs. On se contente de « manger » le blanc reconnu et de rappeler l'analyseur `token`, en lui passant explicitement l'entrée.
6. Enfin la dernière règle reconnaît le lexème vide. En raison de la règle du lexème le plus long, cette règle s'applique lorsqu'aucune des autres règles ne s'applique. Dès lors, elle identifie les erreurs.

Après la compilation de `lexer.mll`, le fichier `lexer.ml` contient donc la réalisation de l'automate `token` sous la forme d'une fonction de type `Lexing.lexbuf -> Token.token`. La « mission » de cette fonction est de reconnaître et renvoyer le lexème présent au début de son entrée et de consommer les caractères correspondants. La consommation des caractères n'est pas explicitée par le type, elle s'opère par effet de bord sur le flux passé en argument. À titre d'exemple, voici un petit bout de code Caml qui compte les lexèmes présents dans l'entrée standard.

```
let entrée = Lexing.from_channel stdin in (* Fabriquer le flux *)
let count = ref 0 in
while Lexer.token entrée <> Token.EOF do
   count := !count + 1
done ;
Printf.printf "J'ai lu %d lexèmes\n" !count
```

4.4.2 Exemples plus compliqués

Mon propos n'est pas donner une description exhaustive de `ocamllex`. Ceux qui sont intéressés peuvent commencer par consulter le manuel [1]. Je vais plutôt décrire quelques exemples et en profiter pour introduire d'autres traits de `ocamllex`.

1. `http://caml.inria.fr/ocaml/htmlman/manual026.html`

Éliminer les commentaires

Commençons donc par considérer le cas des commentaires. Il est naturel de supprimer les commentaires dès l'analyse lexicale, ainsi les commentaires n'ont aucun impact sur toutes les phases suivantes du compilateur. Il y a trois sortes de commentaires.

1. Les commentaires s'étendent d'un mot particulier jusqu'à la fin de la ligne. C'est par exemple le cas en Java :

   // Je suis un commentaire.

   On élime facilement ce type de commentaire en ajoutant une règle à l'analyseur `token`.

   ```
   | "//" [^'\n']* '\n'?  {token lexbuf}
   ```

   L'élimination s'opère comme pour les blancs en rappelant `token` récursivement. On doit bien remarquer que le motif qui filtre le texte du commentaire est `[^'\n']*` (une suite de caractères différents du retour à la ligne) et non pas `_*` (n'importe quel mot). En effet, avec le second motif, les commentaires s'étendraient du premier `//` au dernier retour à la ligne, en raison de la règle du lexème le plus long. On notera encore que le retour à la ligne est optionnel `'\n'?`, afin de considérer aussi un commentaire en fin d'entrée et sans retour à la ligne.

2. Les commentaires sont compris entre deux mots particuliers. mais ils ne peuvent pas être imbriqués. C'est le cas de la seconde sorte de commentaires de Java.

   /∗ Je suis un commentaire,
   sur deux lignes. ∗/

   Pour éliminer ce type de commentaires on ne peut pas s'inspirer du cas précédent, car il n'y a pas de motif exprimant que l'entrée est différente d'un certain mot, comme il existe un motif exprimant que l'entrée est différente d'un certain caractère. Pour s'en sortir on a recours à un deuxième automate `incomment`, lancé à l'ouverture du commentaire et chargé de reconnaître la fin du commentaire.

   ```
   rule token = parse
     ...
   | "/*"  {incomment lexbuf}

   and incomment = parse
   | "*/"  {token lexbuf}
   | _     {incomment lexbuf}
   | eof   {raise Error}
   ```

   L'automate `incomment` rappelle `token` dès qu'il voit la fin du commentaire, mange un caractère du commentaire et se rappelle, ou signale une erreur si l'entrée touche à sa fin avant la fermeture du commentaire.
   À la reflexion, l'expression régulière suivante fonctionne aussi :

   ```
   rule token = parse
     ...
   | "/*" ([^'*']|('*'+[^'*''/']))* '*'+ '/' {token lexbuf}
   ```

   L'expression régulière `([^'*']|('*'+[^'*''/']))*` décrit touts les mots qui ne contiennent pas `"*/"`, sauf les suites non vides de `'*'`, tandis que `'*'+ '/'` décrit les suites (possiblement vides) de `'*'` suivies de `"*/"`. On peut trouver le premier motif en cherchant à expliciter le complément de `"*/"`. Vous trouverez à la fin de la leçon une autre méthode pour trouver cette expression régulière.

3. Les commentaires sont compris entre deux mots particuliers et ils peuvent être imbriqués. Ce dernier type de commentaires permet de neutraliser du source dans un programme, y compris si le source commenté contient des commentaires. C'est le cas des commentaires de Caml.

(
Un commentaire (* avec un commentaire dedans *)
)

On ne peut plus utiliser un deuxième automate comme précédemment, car le niveau d'imbrication des commentaires est arbitraire. De fait, le langage formel défini informellement ci-dessus n'est pas régulier (c'est plus ou moins le langage des expression bien parenthésées) et donc il ne peut pas être reconnu par un automate fini. On pourrait en utilisant deux automates supplémentaires, reconnaître les commentaires imbriqués au plus une fois mais ce n'est pas très général. Pour s'en sortir on va ajouter de l'état aux automates, en se donnant un compteur `depth` du nombre de commentaires ouverts.

```
{
let depth = ref 0
}
rule token = parse
   ...
| "(*"  {depth := 1 ; incomment lexbuf}

and incomment = parse
| "*)"
   {depth := !depth-1 ;
   if !depth <= 0 then  (* où en sommes nous ? *)
     token lexbuf       (* on ferme le premier "(*" *)
    else                (* on ferme un autre "(*" *)
     incomment lexbuf}
| "(*"
    {depth := !depth+1 ; incomment lexbuf}
| _     {incomment lexbuf}
| eof   {raise Error}
```

Le compteur `depth` est logiquement incrémenté par chaque ouverture et décrémenté par chaque fermeture.
Il est possible de se passer du compteur global réalisé à l'aide de la référence `depth` et de le remplacer par un argument supplémentaire donné à la règle `incomment`.

```
rule token = parse
   ...
| "(*"  {incomment 1 lexbuf}

and incomment depth = parse
| "*)"
   {if depth <= 1 then  (* où en sommes nous ? *)
     token lexbuf       (* on ferme le premier "(*" *)
    else                (* on ferme un autre "(*" *)
     incomment (depth-1) lexbuf}
| "(*"
        {incomment (depth+1) lexbuf}
| _     {incomment lexbuf}
| eof   {raise Error}
```

On notera enfin que les commentaires du source ci-dessus ne seraient pas correctement éliminés, en raison de la présence de la chaîne *"(*"* dans les commentaires. Il faudrait pour dépasser ce petit inconvénient, ignorer le contenu des chaînes dans les commentaires à l'aide d'un troisième automate.

Récupérer les chaînes

Considérons maintenant les chaînes (du langage analysé) définies comme tout ce qui se trouve entre deux caractères double quote « `"` ». On ajoute donc un lexème `STRING` **of** `string` au type des lexèmes et on cherche comment reconnaître les chaînes de l'entrée.

Si le double quote est interdit dans les chaînes, alors il n'y a pas de difficulté on s'en tire un peu comme dans le cas des variables.

```
| '"' [^'"']* '"' as lxm (* Noter: '"' est le caractère « " » *)
   {(* supprimer le premier et le dernier caractère de lxm *)
   STRING (String.sub lxm 1 (String.length lxm-2))}
```

On peut éviter l'appel aux fonctions du module `String`, car la construction **as** permet aussi de nommer des sous-chaînes de la chaîne reconnue par le motif. On écrira donc :

```
| '"' ([^'"']* as content) '"' {STRING content}
```

Mais le programmeur peut légitimement vouloir mettre un double quote dans une chaîne. Le concepteur prévoit alors un mécanisme de citation (*quotation*) : à l'intérieur d'une chaîne `\"` veut dire « `"` » et `\\` veut dire « `\` » (pour donner un moyen de mettre « `\` » à la fin d'une chaîne). Je me félicite des guillemets français qui signifient ce caractère là. Comme il n'y a pas de notion de chaînes imbriquées dans les chaînes, on a le net sentiment que l'on va pouvoir s'en sortir à l'aide d'un automate supplémentaire `instring`, comme pour les commentaires `/*`... `*/`. Il y a une petite différence, ici on doit renvoyer en résultat les caractères de la chaîne reconnue et non plus les ignorer. Pour éviter les recopies de chaîne en pagaille, ou pourrait employer des listes de caractères. Nous allons plutôt employer un tampon (*buffer*) tel que défini par le module `Buffer` de la bibliothèque standard (c'est en gros le même fonctionnement que la classe `StringBuffer` de Java). On notera d'abord, dans le code de la figure 4.2, l'utilisation de la la construction **as**

FIGURE 4.2 – Reconnaissance des chaînes

```
{
let sbuff = Buffer.create 16 (* fabriquer le buffer *)
}

rule token = parse
   ...
| '"'  {STRING (instring lexbuf)}

and instring = parse
(* Fin de la chaîne *)
| '"'
    {let r = Buffer.contents sbuff in (* récupérer le contenu de sbuff *)
     Buffer.clear sbuff ;             (* réinitialiser sbuff *)
     r}
(* Caractères cités *)
| '\\' ('"'|'\\' as c) (* c est le second caractère reconnu *)
    {Buffer.add_char sbuff c ; (* à mettre à la fin de sbuff *)
     instring lexbuf}
| _ as c
    {Buffer.add_char sbuff c ;
     instring lexbuf}
| eof
    {raise Error}
```

pour récupérer un caractère de l'entrée. Ce qu'il faut remarquer c'est que, dans la construction *motif* as *variable*, *variable* est de type `char` quand *motif* est un motif caractères ou une alternative de motifs caractère. Jusqu'ici, *motif* pouvait filter des chaînes de longueur diverses et le type de *variable* était `string`.

On remarquera aussi que le type de l'automate `instring` est `Lexing.lexbuf -> string`. Enfin, on ne se laissera pas intimider par les mécanismes de citation de Caml : la notation `'\\'` désigne bien le caractère « \ ».

Abondance de mots-clés

Dans un langage programmation normal il y a souvent un nombre important de mots-clés. En principe cela ne pose pas de problème, il suffit de se donner une règle de reconnaissance par mot-clé et `ocamllex` construit l'automate. Mais en pratique, si il y beaucoup de mots-clés, l'automate sera gros voire énorme. On peut surmonter cet inconvénient en utilisant la clé anglaise de la programmation : la table de hachage. (La clé anglaise taiwanaise est un outil bon marché et polyvalent, mais moins efficace qu'une clé plate Facom de la bonne taille.) Les tables de hachage sont disponibles en Caml dans le module `Hashtbl` de la bibliothèque standard. Les tables de hachage définissent des associations de n'importe quoi à n'importe quoi, on les utilise ici pour définir une association des chaînes aux lexèmes. Une fois une suite de lettres reconnue, on vérifie si par hasard cette suite de lettres n'est pas un mot-clé. Si oui, on a reconnu le mot-clé, si non, on a reconnu un identificateur (voir la figure 4.3). On remarquera l'utilisation plutôt simple des tables de hachage et le respect de la sémantique des mots-clés prioritaires sur les identificateurs.

Figure 4.3 – Reconnaissance *a posteriori* des mots-clés

```
{
let keywords = Hashtbl.create 17 (* création de la table de hachage *)

(* initialisation de la table *)
let _ =
  Hashtbl.add keywords "let" LET ; (* associer LET à "let" *)
  Hashtbl.add keywords "in" IN ;   (* associer IN à "in" *)
  ...
}

rule token = parse
  ...
| ['a'-'z']+
   {let lxm = Lexing.lexeme lexbuf in
    try
      Hashtbl.find keywords lxm (* chercher lxm dans la table *)
    with Not_found -> (* lxm n'est pas un mot-clé *)
      Var lxm }       (* c'est donc un identificateur *)
```

Les erreurs

Pour le moment nos analyseurs se contentent signaler les erreurs, sans donner aucune information spécifique. On peut enrichir l'information donnée au programmeur en différenciant les erreurs, pour signaler un caractère illégal, une chaîne non-terminée etc. Mais l'information qui aidera sans doute le plus le programmeur est une position dans le fichier analysé. Or, dans une action, la fonction `Lexing.lexeme_start` (resp. `Lexing.lexeme_end`) fournit la position dans l'entrée du début (resp. de la fin) du dernier lexème reconnu. On peut alors transmettre cette position comme argument de l'exception `Error` en l'accompagnant d'un message d'erreur explicatif.

```
{
exception Error of int * string

let error pos = raise (Error (pos,msg))

}

rule token = parse
  ...
| "" {error (Lexing.lexeme_start lexbuf) "Caractère illégal"}

and incomment = parse
  ...
| eof {error (Lexing.lexeme_start lexbuf) "commentaire non terminé"}
```

En fait, il faudrait travailler un petit peu plus pour par exemple transmettre la position du commentaire ouvert et non refermé.

Dans le cas où l'entrée est un fichier, la position comptée en caractères à partir du début du fichier est assez peu pratique, même si un éditeur tel que `emacs` sait automatiquement retrouver une telle position. Il est plus pratique de donner la position sous la forme d'un numéro de ligne et d'un compte de caractères à partir du début de la ligne. Considérons par exemple le fichier `er.ml` suivant :

```
let x = 1
let y = "coucou
let z = 1
```

La tentative de compilation `ocamlc er.ml` donne :

```
File "er.ml", line 2, characters 8-9:
String literal not terminated
```

Le compilateur retrouve assez facilement un numéro de ligne à partir d'une position (en réouvrant le fichier), et le confort d'utilisation gagné vaut ce petit effort.

Une autre possibilité est de tenter de corriger les erreurs (en les signalant tout de même !) et de reprendre l'analyse. On pourra par exemple simplement ignorer les caractères spéciaux. Mais c'est en fait difficile et souvent un peu vain, car l'analyseur aura du mal à deviner ce que le programmeur a en tête. Il ne saura pas, par example, où refermer une chaîne qui court jusqu'à la fin de l'entrée.

4.5 Bibliothèque des expressions régulières

Un outil tel que `ocamllex` facilite l'écriture des analyseurs lexicaux, mais il n'est pas très pratique pour programmer à l'aide des expressions régulières, comme on le fait par exemple beaucoup en Perl. En effet, on doit mettre l'analyseur dans son propre fichier `.mll` ce qui est un peu lourd. Par ailleurs, les générateurs d'analyseurs lexicaux visent plutôt un public de concepteurs de compilateurs et leurs concepteurs ne proposent pas toujours quelques traits courants et pratiques qui séduisent le plus vaste public des programmeurs. Des bibliothèques ≪ d'expressions régulières ≫ répondent à ce besoin de plus grande flexibilité et d'expressivité étendue.

En Caml, on dispose de la bibliothèque `Str`. Elle ne sera pas décrite ici, ceux qui sont intéressés peuvent consulter le manuel. Il y a deux points notables :

— Le parti-pris syntaxique est à l'opposé de `ocamllex` : au lieu de citer les caractères de l'alphabet on cite les constructions des expressions régulières. En `ocamllex` on écrivait `('a'|'b')`, en `Str` on écrira `\(a\|b\)`. Enfin, ce n'est pas tout à fait exact, les caractères `$^.*+?[]` sont spéciaux et doivent parfois être cités avec ≪ `\` ≫ pour se signifier eux mêmes.

Figure 4.4 – Exemple d'utilisation de la bibliothèque `Str`

```
open Str
open Printf
(*
  - ^ initial -> début de ligne
  - [^.]      -> tout sauf le point
  - \.        -> un point
  - \(... \)  -> groupage, groupes numérotés de gauche à droite
  - .*        -> n'importe quelle chaîne
  - $         -> fin de ligne
*)


let auto = regexp "^[^.]+\.\([^@.]+\)@\(.*\)$" (* compilation de l'automate *)

let extrait s =
  if
     string_match auto s 0 &&    (* filtrage *)
        String.lowercase (matched_group 2 s) = (* extraction 2ème groupe *)
       "polytechnique.fr"
  then
    printf "Le nom est %s\n" (matched_group 1 s)   (* extraction 1er groupe *)
  else
    printf "Ce n'est pas une adresse de l'X\n"
```

— Un trait supplémentaire intéressant est que, en cas de réussite du filtrage, le parenthésage permet d'extraire des sous-chaînes de la chaîne filtrée.

Nous nous contenterons donc d'un exemple simple. Nous cherchons à reconnaître des adresses de courrier électroniques de la forme *Prénom*.*Nom*`@polytechnique.fr`, afin d'en extraire le nom. En outre, nous acceptons les variations de casse dans le nom de domaine `polytechnique.fr`. Le code est donné par la figure 4.4.

La bibliothèque `Str` ne fait pas partie de la bibliothèque standard. Par conséquent, l'argument `str.cma` doit être donné explicitement lors de l'édiction de liens :

```
# ocamlc options str.cma files...
```

4.6 Un peu de théorie

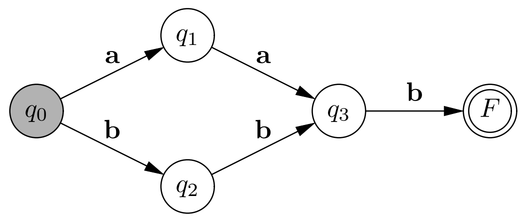
Cette section culturelle explique les principes des générateurs d'analyseurs lexicaux tels que `ocamllex`. Le principe général est celui d'une véritable compilation des expressions régulières aux automates.

4.6.1 Automates finis déterministes (DFA)

Un automate fini déterministe M est un quintuplet $(\Sigma, Q, \delta, q_0, F)$ où
— Σ est un alphabet ;
— Q est un ensemble fini d'états ;
— $\delta : Q \times \Sigma \to Q$ est la fonction (partielle) de transition ;
— q_0 est l'état initial ;
— F est un ensemble d'états finaux.

On peut étendre δ sur $Q \times \Sigma^* \to Q$ par $\begin{cases} \delta(q, \epsilon) = q \\ \delta(q, aw) = \delta(\delta(q, a), w) \end{cases}$ .Le langage $L(M)$ reconnu par l'automate M est l'ensemble $\{\, w \mid \delta(q_0, w) \in F \,\}$ des mots permettant d'atteindre un état final à partir de l'état initial.

Exemple Soit un automate :



L'automate reconnaît le langage $\{\mathbf{aab}, \mathbf{bbb}\}$, La formalisation comme un quintuplet est laissée en exercice.

4.6.2 Automates finis non-déterministes (NFA)

La définition est la même que celle de automates déterministes, compte tenu des deux détails suivants :

1. Les transitions sont définies par une relation et non plus par une fonction, c'est à dire que plusieurs transitions issues d'un état donné peuvent porter la même étiquette.
2. Il existe des transitions « *spontanées* » qui portent une étiquette spéciale, classiquement ϵ.

On peut exprimer ces modifications en définissant les transitions entre états comme une relation δ (fonction dans les booléens) sur $Q \times (\Sigma \cup \{\epsilon\}) \times Q$. Une telle relation peut aussi très bien se noter comme une liste de triplets $q \overset{a}{\mapsto} q'$.

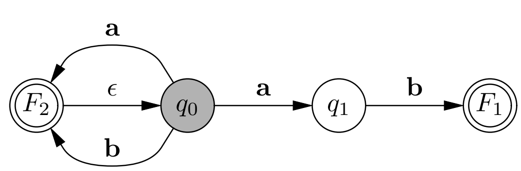
On étend δ sur $Q \times \Sigma^* \times Q$ par

$$\left\{ \begin{array}{ccc} & \delta(q, \epsilon, q) & \\ & & \\ \delta(q, \epsilon, q'') \wedge \delta(q'', w, q') & \Rightarrow & \delta(q, w, q') \\ \delta(q, a, q'') \wedge \delta(q'', w, q') & \Rightarrow & \delta(q, aw, q') \end{array} \right.$$

(Il y a un peu d'abus, la relation définie est le point fixe des implications et il y a quelques quantificateurs implicites.)

Le langage $L(M)$ reconnu par un automate non déterministe est $\{w \mid \exists q_f \in F, \delta(q_0, w, q_f)\}$. Notons, et c'est assez intéressant, que les transitions définissent aussi une fonction de $Q \times \Sigma^*$ vers 2^Q (ensembles d'états) : à un état q et un mot w, on associe l'ensemble Q' des états q' tels que la relation $\delta(q, w, q')$ tient.

Exemple Soit un automate :

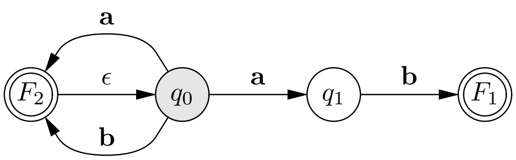


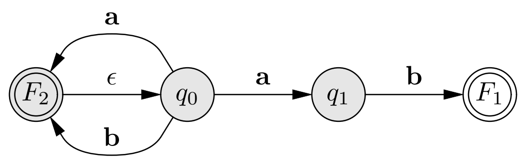
L'automate reconnaît le langage des mots d'au moins une lettre formés avec **a** et **b**. On note que le mot **ab** peut être reconnu de deux façons différentes (à q_0 et **ab**, on associe $\{q_0, F_1, F_2\}$). On peut intuitivement voir la reconnaissance d'un mot par un tel automate comme le calcul d'un ensemble d'états effectués ainsi.

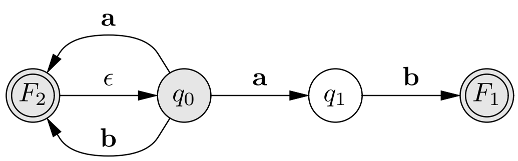
— Initialement, l'ensemble des états est l'état initial plus tous les états accessibles par une suite de transitions spontanées.

— Pour consommer un caractère a, l'automate suit toutes les transitions étiquetées par a et issues de son ensemble d'états courant. Ensuite, il complète le nouvel ensemble d'états courants comme initialement.

Ainsi la consommation du mot **ab** peut se décrire par les trois dessins suivants (cette fois ci, c'est l'ensemble des états courants qui est grisé).







4.6.3 Compilation des expressions régulières

Nous sommes maintenant équipés pour décrire la fabrication d'un automate fini déterministe reconnaissant un langage régulier donné. Il s'agit d'une véritable compilation qui comprend trois phases successives [2].

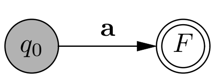
Des expressions régulières aux NFA

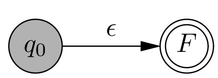
L'intérêt des automates non-détermistes est qu'il est facile d'associer un automate (Q, δ, s, F) reconnaissant un langage L à une expression régulière M définissant le langage L.

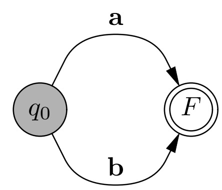
— $\llbracket a \rrbracket = (\{s, f\}, \{s \overset{a}{\mapsto} f\}, s, \{f\})$
— $\llbracket \epsilon \rrbracket = (\{s, f\}, \{s \overset{\epsilon}{\mapsto} f\}, s, \{f\})$
— $\llbracket M \mid M' \rrbracket = (Q \cup Q' \cup \{s''\}, \delta \cup \delta' \cup \{s'' \overset{\epsilon}{\mapsto} s, s'' \overset{\epsilon}{\mapsto} s'\}, s'', F \cup F')$
— $\llbracket M\ M' \rrbracket = (Q \cup Q', \delta \cup \delta' \cup \{f \overset{\epsilon}{\mapsto} s', f \in F\}, s, F')$
— $\llbracket M^* \rrbracket = (Q, \delta \cup \{f \overset{\epsilon}{\mapsto} s, f \in F\}, s, \{s\})$

Ce n'est pas la seule construction possible. Par exemple, on peut exprimer graphiquement une construction légèrement différente (la modification ne porte que sur l'alternative). La nouvelle construction produit des automates à un seul état initial et un seul état final. Ces automates sont représentés par des boîtes portant le nom du motif représenté, leur état initial est à gauche et leur état final à droite.

— Le motif est a, ϵ ou un motif $[ab]$ (en optimisant un peu par rapport à l'expansion en $a \mid b$).

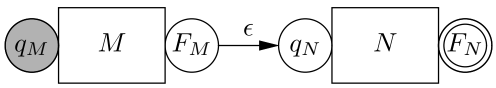


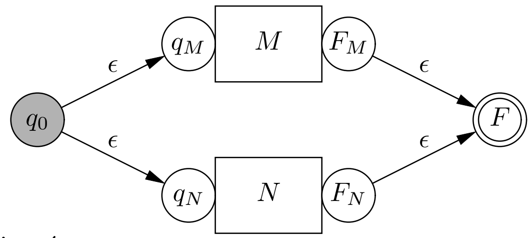




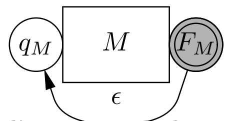
— Le motif est $M\ N$ ou $M \mid N$.

2. `ocamllex` procède en fait en une seule passe, qui combine les deux premières phases de notre description.

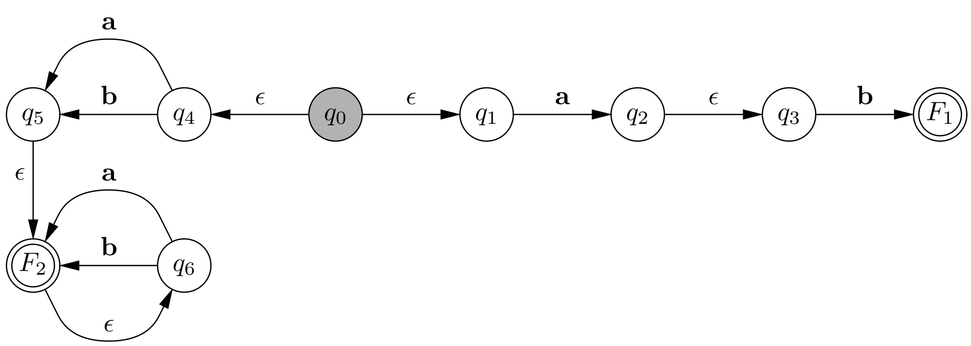




— Le motif est M^*, on notera qu'aucun état n'est ajouté.



La première construction de l'alternative à l'avantage de laisser intacts les états finaux, de sorte qu'ils dénonceront plus tard la branche choisie. Ainsi, en suivant la première construction de l'alternative et en optimasant le motif $[ab]$, l'expression régulière $([ab]+ \mid ab)$ se compile en l'automate suivant :



Des NFA aux DFA

On peut très bien exécuter directement un automate non-déterministe, en considérant un ensemble d'états courants. Mais la manipulation des ensembles coûte toujours un peu cher, et dans un contexte de compilation, il vaut mieux transformer les NFA en des DFA équivalents (*i.e.* qui reconnaissent le même langage). Cela revient à payer le prix de la réalisation des ensembles une seule fois, et est particulièrement rentable lorsque l'analyseur a vocation a être exécuté de nombreuses fois.

L'idée est inspirée de l'exécution des automates non-deterministes, il suffit de considérer tous les ensembles d'états possibles durant toutes les exécutions possibles. Soit, à partir d'un NFA $A_n = (Q, \delta, q_0, F)$ on souhaite trouver un DFA équivalent $A_d = (R, \gamma, Q_0, G)$. On choisira les états de A_d parmi l'ensemble des parties de Q, on notera donc les états de A_d, Q_0, Q_1, etc. On définit deux fonctions sur 2^Q.

— la fermeture $\mathcal{F}$, comme le point fixe de $S = S \cup \{q \mid \exists q' \in S, \delta(q', \epsilon, q)\}$ (autrement dit, on suit toutes les transitions spontanées possibles issues des états de S).

— la consommation d'un caractère a, noté $\mathcal{C}_a$ comme $\mathcal{C}_a(S) = \{q \mid \exists q' \in S, \delta(q', a, q)\}$.

L'algorithme de traduction consiste à tout simplement calculer l'ensemble des ensembles d'états atteignables (les Q_i) à partir de l'état initial de A_n, en se rappelant au passage des transitions entre les Q_i. Plus formellement, on calcule le point fixe

$$R = \mathcal{F}(\{q_0\}) \cup \{Q_i \mid \exists Q_j \in R \wedge \exists a \in \Sigma, Q_i = \mathcal{F}(\mathcal{C}_a(Q_j))\} \cup R$$

Avec en outre, $\gamma(Q_j, a)$ défini comme $Q_i = \mathcal{F}(\mathcal{C}_a(Q_j))$, et G composé des états de R qui contiennent au moins un état final de F.

Ça à l'air un peu compliqué, mais un exemple expliquera mieux ce qui se passe. Considérons toujours le même automate, celui qui résulte de la compilation de $([ab]+ \mid ab)$, en se plaçant dans l'alphabet $\Sigma = \{\mathbf{a}, \mathbf{b}\}$. Dans un premier temps, nous disposons de l'état initial $Q_0 = \mathcal{F}(\{q_0\}) = \{q_0, q_1, q_4\}$ les états et les transitions sont ensuite calculées ainsi :

— $Q_1 = \mathcal{F}(\mathcal{C}_{\mathbf{a}}(Q_0)) = \{q_2, q_3, q_5, q_6, F_2\}$ avec donc $Q_0 \overset{\mathbf{a}}{\mapsto} Q_1$.

— $Q_2 = \mathcal{F}(\mathcal{C}_{\mathbf{b}}(Q_0)) = \{q_5, q_6, F_2\}$, avec donc $Q_0 \overset{\mathbf{b}}{\mapsto} Q_2$.

— $Q_3 = \mathcal{F}(\mathcal{C}_{\mathbf{a}}(Q_1)) = \{q_6, F_2\}$, avec donc $Q_1 \overset{\mathbf{a}}{\mapsto} Q_3$. On remarque aussi que l'on a $Q_3 = \mathcal{F}(\mathcal{C}_{\mathbf{a}}(Q_3))$ et $Q_3 = \mathcal{F}(\mathcal{C}_{\mathbf{b}}(Q_3))$, on peut donc ajouter deux transitions $Q_3 \overset{\mathbf{a}}{\mapsto} Q_3$ et $Q_3 \overset{\mathbf{b}}{\mapsto} Q_3$ et aucun nouvel état.

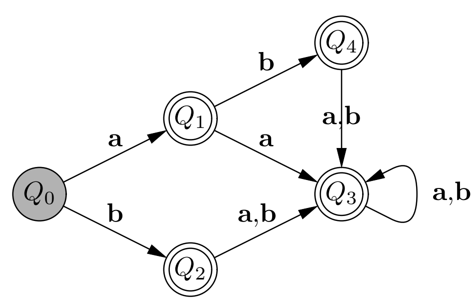
— $Q_4 = \mathcal{F}(\mathcal{C}_{\mathbf{b}}(Q_1)) = \{q_6, F_1, F_2\}$, avec donc $Q_1 \overset{\mathbf{b}}{\mapsto} Q_4$.

— On a $Q_3 = \mathcal{F}(\mathcal{C}_{\mathbf{a}}(Q_2))$ et $Q_3 = \mathcal{F}(\mathcal{C}_{\mathbf{b}}(Q_2))$, soit encore deux nouvelles transitions vers Q_3, $Q_2 \overset{\mathbf{a}}{\mapsto} Q_3$ et $Q_2 \overset{\mathbf{b}}{\mapsto} Q_3$.

— Et il en va de même pour Q_4 : on a $Q_3 = \mathcal{F}(\mathcal{C}_{\mathbf{a}}(Q_4))$ et $Q_3 = \mathcal{F}(\mathcal{C}_{\mathbf{b}}(Q_4))$.

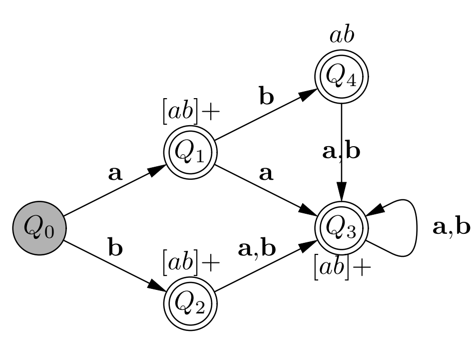
Le calcul est maintenant terminé, car le cas de la consommation de **a** et **b** a été examiné à partir de tous les états possibles de A_d et on obtient donc l'automate de la figure 4.5.

FIGURE 4.5 – Automate de $([ab]+|ab)$



Il est maintenant intéressant d'examiner l'état Q_4 qui contient les deux états finaux du NFA F_1 et F_2. Cet état n'est atteint que si l'entrée est **ab**. Or, F_1 traduit le filtrage de cette entrée par le motif ab, tandis que F_2 traduit le filtrage par $[ab]+$. Ce n'est pas bien grave si on ne s'intéresse qu'à la définition du langage reconnu. En revanche, si l'automate est censé reconnaître un mot-clé (ab) et des variables ($[ab]+$), il faut faire un choix. Le choix est arbitraire et repose sur l'ordre de présentation des motifs. On suppose pour la suite que ab prime sur $[ab]+$ et on décore les états finaux par le motif choisi. On obtient l'automate de la figure 4.6.

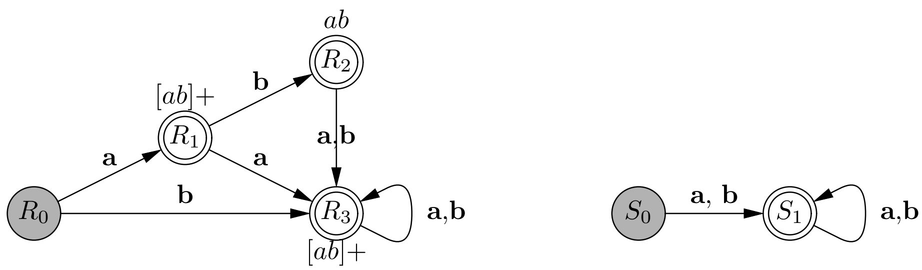
FIGURE 4.6 – Automate de $([ab]+|ab)$, états finaux distingués



Minimisation des DFA

L'automate déterministe donné comme compilation de $([ab]+|ab)$ n'est pas optimal : il existe un automate plus petit qui reconnaît le même langage que lui. De fait, selon que l'on souhaite distinguer les motifs reconnus ou pas et en revenant à l'expression régulière on trouve facilement deux automates équivalents (voir la figure 4.7).

FIGURE 4.7 – Automates optimaux



Évidemment on peut maintenant se demander comment produire un automate optimal à partir d'un automate donné. Je vais juste donner l'idée. Deux états Q_i et Q_j de l'automate donné sont équivalents, noté $Q_i \cong Q_j$, quand les suffixes du langage L reconnus à partir de ces états sont exactement les mêmes. On peut fusionner les états équivalents, le langage reconnu ne changera pas. Tous les états finaux de l'automate de la figure 4.5 sont équivalents. En effet toutes les reconnaissances amorcées à partir de ces états définissent le langage $[ab]*$, on peut alors *à fortiori* fusionner Q_1, Q_2, Q_3 et Q_4 en S_1. Notons que si l'on distingue les états finaux (figure 4.6), alors on ne peut fusionner que Q_2 et Q_3.

Le principe d'un algorithme de minimisation est de remplacer les états par les classes d'équivalence de $\cong$. Un algorithme possible fonctionne par raffinements successifs d'une partition initiale des états en états finaux et non finaux (si les états finaux sont distingués, il y a un élément de la partition initiale par sorte d'état final), jusqu'à obtenir une partition stable sous la relation $\delta(\_, a)$ pour tous les caractères a. Plus précisément la relation recherchée est :

$$\forall R \in \mathcal{P}, \forall Q, Q' \in R \times R, \forall a \in \Sigma, \exists! R', \quad \delta(Q, a) \in R' \wedge \delta(Q', a) \in R'$$

Il y a de nombreuses variations de cette idée, les variations concernent surtout l'arrangement des itérations et les structures de données. L'algorithme le plus efficace utilise la relation inverse δ^{-1}.

4.6.4 Réalisation des automates

On peut réaliser les automates directement par du code. En Caml on aura recours à une fonction par état, chaque fonction filtrant le caractère courant. Ainsi, l'automate optimal de gauche de la figure 4.7, donnerait lieu à ce genre de code :

```
let rec state0 = function
| 'a' -> state1 (next_char ())
| 'b' -> state3 (next_char ())
| _   -> error ()

and state1 = function
| 'a' -> state3 (next_char ())
| 'b' -> state2 (next_char ())
| _   -> "[ab]+"
  ...
```

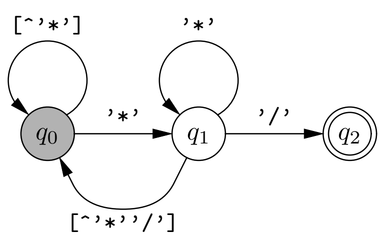
Mais le code risque d'être assez abondant. On a plutôt tendance à définir l'automate par la table de ses transitions, c'est à dire comme une matrice d'entiers, les lignes étant les états et les colonnes les caractères. Ainsi sur l'alphabet $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ on a la matrice de transitions :

$$\begin{array}{c} \begin{array}{ccc} \mathbf{a} & \mathbf{b} & \mathbf{c} \end{array} \\ \begin{pmatrix} 1 & 3 & -1 \\ 3 & 2 & -1 \\ 3 & 3 & -1 \\ 3 & 3 & -1 \end{pmatrix} \end{array} \qquad \begin{pmatrix} - \\ \texttt{"[ab]+"} \\ \texttt{"ab"} \\ \texttt{"[ab]+"} \end{pmatrix}$$

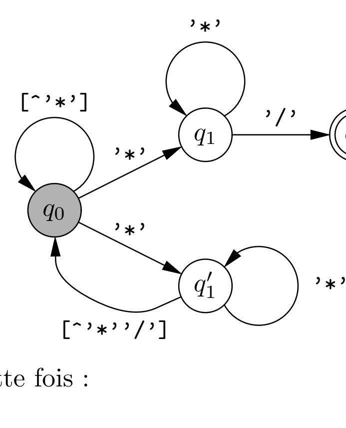
Une entrée -1 indique une erreur et le vecteur de droite désigne les états finaux. On peut alors interpréter la table pour réaliser l'automate spécifié. Le calcul d'une transition prend un coût constant, car il consiste à accéder dans un tableau. *A priori* nous n'avons pas gagné beaucoup de place (car il y a normalement de l'ordre de 256 caractères possibles), mais les matrices de transition sont souvent creuses (chaque ligne possède beaucoup de valeurs identiques). On peut alors représenter la matrice de transition de façon compacte en pratique, tout en gardant un coût constant pour la réalisation d'une transition.

4.6.5 Exemple d'exercice sur les automates

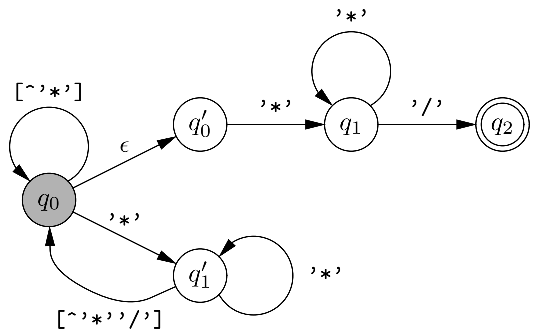
Rappelons que le problème est de trouver une expression régulière qui décrit les commentaires de C : il s'étendent d'un mot `"/*"` au premier mot `"*/"` qui suit. Or, on peut assez facilement trouver l'automate déterministe qui, laché dans un commentaire, sait en trouver la fin :



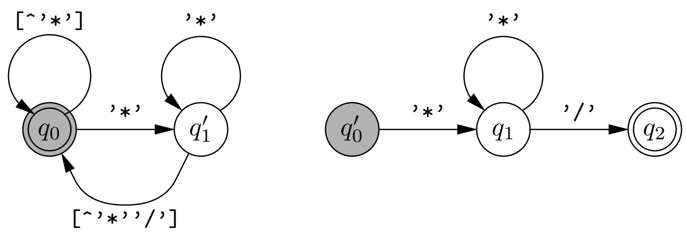
Ensuite on cherche à trouver une expression régulière décrivant le langage de cet automate. On va donc inverser la construction présentée précédemment dans un cas particulier. On duplique d'abord l'état q_1 en répartissant astucieusement ses transitions. On a alors l'automate non-déterministe équivalent suivant :



Soit encore, en dupliquant q_0 cette fois :



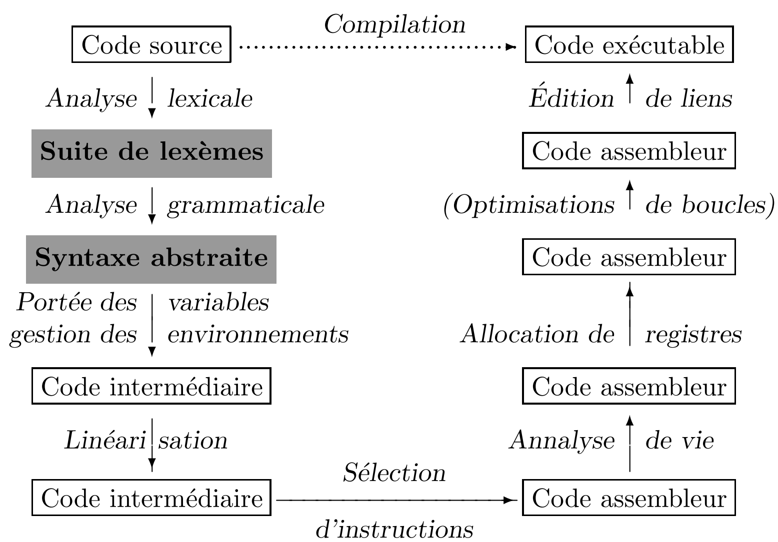
Nous retrouvons maintenant la composition en séquence des deux automates suivants :



Le langage de l'automate de gauche est `([^'*']|('*''*'*[^'*''/'])*` (répétition des deux sortes de chemins possibles de q_0 à lui même) tandis que le langage de l'automate de droite est `'*'+ '/'` (facile). Bon, la dérivation de l'expression régulière à partir de l'automate déterministe manque de généralité, mais elle est correcte. L'idée étant de vérifier à chaque étape que les chemins de l'état initial à l'état final ne changent pas.

Chapitre 5

Analyse grammaticale



Comme déjà dit, l'analyse grammaticale fabrique l'arbre de syntaxe abstraite à partir des lexèmes produits par l'analyse lexicale. L'arbre de syntaxe abstraite est important, car il est le support de la sémantique du langage. Il importe donc, pour comprendre ce que fait exactement un programme, de bien comprendre d'abord comment son source s'explique en terme de syntaxe abstraite. Il n'est pas surprenant que cette compréhension découle directement d'une connaissance un peu fine du processus de l'analyse grammaticale. Le mieux est je crois, pour être précis, de donner un tour théorique au discours.

5.1 Grammaires

Les grammaires algébriques définissent une classe bien particulière de langages formels (voir le chapitre précédent, section 4.2) : les langages algébriques (*context-free*).

Dans ce cadre, les lettres de l'alphabet Σ s'appellent les (symboles) *terminaux* et sont notés par des minuscules a, b, c etc. ou parfois **id**, **int**, **+**, lorsque qu'ils sont connus. On se donne un ensemble de nouveaux symboles V, dits *non-terminaux* que nous noterons par des majuscules, A B, C etc. Les symboles de la grammaire sont à la fois les terminaux (pris dans Σ) et les non-terminaux (pris dans V), quand nous parlons d'eux nous les noterons à l'aide de lettres grecques majuscules Δ, Γ etc. Dans le même ordre d'idée, nous noterons les mots formés de symboles terminaux et non-terminaux. par des lettres grecques minuscules, α, β, γ, etc. Toutefois, La lettre grecque ϵ désigne toujours le mot vide. Un non-terminal particulier est dit symbole de départ.

Une grammaire algébrique (*context-free*) est une liste de *productions* de la forme $A \to \alpha$. On regroupe parfois plusieurs productions de même membre gauche $A \to \alpha_1$, $A \to \alpha_2$, ... , $A \to \alpha_n$ en écrivant $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$.

Le langage $L(G)$ engendré par une grammaire G est l'ensemble des mots produit en partant du symbole de départ S (souvent sous-entendu) et en appliquant la démarche suivante aux mots α :

1. Si α n'est formé que de terminaux alors α est un mot w de $L(G)$.
2. Sinon, α peut se décomper en $\beta A \gamma$, où A est un non-terminal.
3. Alors, on considère une production $A \to \delta$, on remplace A dans α par δ, noté $\alpha \Rightarrow \beta\delta\gamma$ et on recommence en 1.

L'opération décrite s'appelle une dérivation de w et se note $S \overset{*}{\Rightarrow} w$ (w est un mot de terminaux). On utilise la même notation pour les étapes intermédiaires $\alpha \overset{*}{\Rightarrow} \beta$, où α et β sont des mots de symboles quelconques de la grammaire. la figure 5.1 donne une grammaire G des expressions arithmétiques.

FIGURE 5.1 – Une grammaire pour les expressions arithmétiques

$$E \to E \texttt{+} E \qquad E \to E \texttt{-} E \qquad E \to E * E \qquad E \to E \mathbin{/} E$$

$$E \to (E) \qquad E \to \mathbf{int}$$

Et voici trois dérivations de la même expression arithmétique **1 + 2 * 3** en admettant que **1**, **2** et **3** sont des entiers $\mathbf{int}_1$, $\mathbf{int}_2$ et $\mathbf{int}_3$:

$$\underline{E} \Rightarrow E + \underline{E} \Rightarrow \underline{E} + E * E \Rightarrow \mathbf{1} + \underline{E} * E \Rightarrow \mathbf{1} + \mathbf{2} * \underline{E} \Rightarrow \mathbf{1} + \mathbf{2} * \mathbf{3}$$

$$\underline{E} \Rightarrow \underline{E} * E \Rightarrow \underline{E} + E * E \Rightarrow \mathbf{1} + \underline{E} * E \Rightarrow \mathbf{1} + \mathbf{2} * \underline{E} \Rightarrow \mathbf{1} + \mathbf{2} * \mathbf{3}$$

$$\underline{E} \Rightarrow \underline{E} + E \Rightarrow \mathbf{1} + \underline{E} \Rightarrow \mathbf{1} + \underline{E} * E \Rightarrow \mathbf{1} + \mathbf{2} * \underline{E} \Rightarrow \mathbf{1} + \mathbf{2} * \mathbf{3}$$

(Le non-terminal substitué par le membre droit d'une production à chaque étape est souligné et le remplacement apparaît comme ça.)

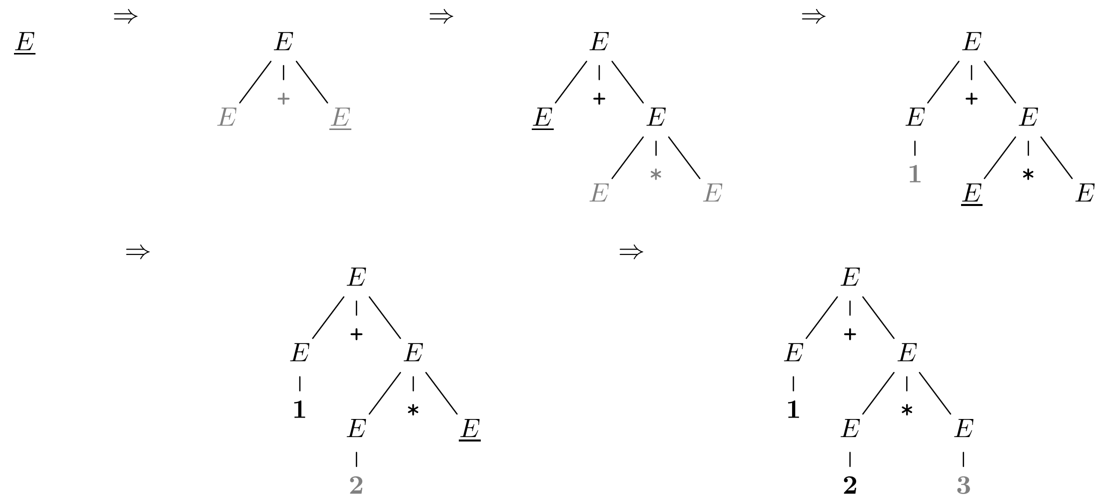
La question de l'analyse syntaxique consiste d'abord à décider si un mot de non-terminaux quelconque appartient à $L(G)$ ou pas, autrement dit de trouver une dérivation du mot. En pratique l'analyse syntaxique revient aussi à donner un « *sens* » au mot. Ainsi si nous considérons le mot **1 + 2 * 3**, nous savons maintenant de façon trois fois certaine qu'il s'agit bien d'un mot de $L(G)$. Le sens à lui donner serait normalement un arbre de syntaxe abstraite mais ici nous pouvons aussi l'exprimer comme l'entier résultat du calcul proposé. L'idée est de parcourir les dérivations en replaçant les productions de la forme $E \to E\ \texttt{op}\ E$ par l'opération correspondante et les autres productions par rien. À ce compte, seules deux productions importent et nous obtenons les calculs suivants (à lire de la droite vers la gauche) :

$$\underline{7} \Rightarrow 1 + \underline{6} \Rightarrow 1 + 2 * 3 \qquad \underline{9} \Rightarrow \underline{3} * 3 \Rightarrow 1 + 2 * 3 \qquad \underline{7} \Rightarrow 1 + \underline{6} \Rightarrow 1 + 2 * 3$$

Un analyseur syntaxique va partir du mot $w =$ **1 + 2 * 3** et chercher à produire une dérivation de E qui engendre ce mot. C'est cette dérivation qui est le support de ce qu'un compilateur situé en aval de l'anlyseur comprendra. On constate ici que :

1. la première et la troisième dérivation produisent le même sens,
2. la première et la deuxième dérivation produisent un sens distinct

La première remarque nous conduit à penser que les dérivations sont trop précises, seul importe ici le choix de la première production utilisée. On peut l'exprimer mieux en considérant des arbres de dérivation. Un tel arbre se construit à partir d'une dérivation quelconque en appliquant les productions non plus sur des mots des symboles de la grammaire mais sur une stucture d'arbre ad-hoc. Ainsi, la première dérivation nous donne les arbres successifs :



(Le mot de chaque étape se retrouve en lisant les feuilles de l'arbre de la gauche vers la droite.)

Deux derivations qui ont le même sens produisent au final le même arbre. Ainsi, toutes les dérivations possibles de $w = \mathbf{1} + \mathbf{2} * \mathbf{3}$ s'expriment en deux arbres :

On observe maintenant que l'arbre de syntaxe abstraite traditionnel se déduit de l'arbre de dérivation en enlevant touts les non-terminaux (et les parenthèses, redondantes dans une structure arborescente). Ici on obtient les deux arbres de syntaxe abstraite.

On peut alors aussi dire (et c'est exactement la même chose) que $1 + 2 * 3$ pourait se comprendre comme $1 + (2 * 3)$ ou $(1 + 2) * 3$.

Lorsque l'on raisonne sur les analyses il n'est pas commode de faire des dessins d'arbre. On désigne donc une classe de dérivations équivalentes (*i.e.* qui ont le même arbre de dérivation au final) par une dérivation particulière. On a ainsi les dérivations *gauches* (resp. *droites*) qui consistent à substituer toujours le non-terminal le plus à gauche (resp. à droite) dans la chaîne de symboles en cours de dérivation. Il existe une unique dérivation gauche (une unique dérivation droite) dans une classe de dérivations équivalentes. Intuitivement, cela veut dire que l'ordre dans lequel on calcule les arguments des opérations n'a pas d'importance. Ainsi nos deuxième et troisième dérivations sont gauches. et puisque nous pouvous exhiber deux dérivations gauches (ou deux arbres de dérivation) du mot $w = \mathbf{1} + \mathbf{2} * \mathbf{3}$, notre grammaire ne donne pas un sens bien clair aux expressions arithmétiques. On dit qu'elle est *ambigüe*.

Afin de donner un sens clair à la syntaxe concrète on souhaite disposer d'une grammaire G', équivalente à G (*i.e.* qui définit le même langage) et non-ambigüe. Soit une grammaire G' pour laquelle il existe une unique dérivation gauche (ou droite) de tout les mots de $L(G') = L(G)$. Or, dans le cas des expressions arithmetiques, on sait comment procéder depuis l'école primaire. Il suffit d'effectuer les multiplications et les divisions avant les additions et les soustractions. En outre, il faut aussi effectuer les soustractions et les divisions de la gauche vers la droite ($1-2-3$ se calcule comme $(1-2)-3$ et non pas comme $1-(2-3)$). Remarquons que, si seule la valeur entière d'une expression nous intéressait, nous aurions pu oublier ce dernier point dans le cas de l'addition et de la multiplication qui sont associatives. Mais ce n'est de toute façon pas sain, car pour donner une sémantique précise aux programmes, il convient de définir précisément la syntaxe abstraite en fonction de la syntaxe concrète. À partir de ces intuitions nous pouvons produire la grammaire G' de la figure 5.2. Cette grammaire est certainement équivalente à G et non-ambigüe, nous l'utilisons depuis l'enfance pour nous parler entre nous de tous les calculs élémentaires possibles, et nous nous comprenons.

FIGURE 5.2 – Une grammaire non-ambigüe pour les expressions arithmétiques

$$E \to E \texttt{+} T \qquad E \to E \texttt{-} T \qquad E \to T$$

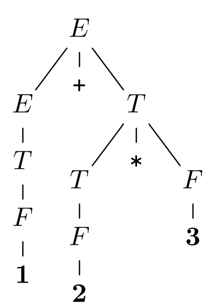
$$T \to T \texttt{*} F \qquad T \to T \texttt{/} F \qquad T \to F$$

$$F \to \texttt{(}E\texttt{)} \qquad F \to \mathbf{int}$$

5.2 Analyse descendante (*top-down parsing*)

Une fois bien défini le langage à analyser, c'est à dire une fois posée une grammaire G. Nous voulons d'abord vérifer qu'un mot w de Σ^* (une suite de lexèmes) appartient bien à $L(G)$.

Un première intuition est la suivante : G' n'est pas ambigüe (et $L(G) = L(G')$), il existe donc un unique arbre de dérivation de w. Dans le cas de notre exemple, le voici :



Nous aimerions alors faire correspondre chaque non-terminal de cet arbre à un appel de fonction, et chaque terminal à la consommation d'un lexème dans un flux. Observons d'abord que cela revient à construire l'arbre de dérivation de la racine vers les feuilles et que le flux impose que l'arbre se construise de gauche à droite (et donc on reconstitue une dérivation gauche, dans le bon ordre). Dans cet exemple, nous aimerions appeler E d'abord, qui appelle E, puis consomme `+`, puis appelle T. Le premier appel recursif de E devrait appeler T, qui appellerait F qui consommerait enfin le lexème **1**. Ensuite après les retours de F puis de T, `+` serait en tête du flux, prêt à être consommé par l'appel initial de E. Si nous cherchons maintemant à écrire la fonction `expr` correspondant à E, nous sommes immédiatement confrontés à un premier problème grave : `expr` doit commencer par appeler `expr` sans rien consommer dans le flux. Dès lors, tout appel à `expr` bouclera d'entrée de jeu. Sur la gramaire G' le problème est révélé par la production $E \to E \texttt{+} T$ où E apparait en tête du membre droit. Une telle production est dite *récursive à gauche*. Tant qu'il ne s'agit que de reconnaître $L(G)$ nous pouvons très bien utiliser la grammaire G'' de la figure 5.3 qui est équivalente à G et n'est pas récursive à gauche.

FIGURE 5.3 – Une autre grammaire non-ambigüe pour les expressions arithmétiques

$$E \to T \texttt{ + } E \qquad E \to T \texttt{ - } E \qquad E \to T$$

$$T \to F \ast T \qquad T \to F \texttt{ / } T \qquad T \to F$$

$$F \to \texttt{(}E\texttt{)} \qquad F \to \textbf{int}$$

Nous pouvons maintenant examiner l'écriture de l'analyseur d'un peu plus près. Nous nous donnons le cadre suivant :

1. Nos lexèmes sont ceux de la section 4.3.1.
2. Nous pouvous appeler et définir des fonctions récursives. Notre analyseur est une fonction qui renvoie `()` si le flux contient un mot de $L(G)$, et qui sinon, lève l'exception `Error`.
3. Nous pouvons regarder quel est le lexème en attente (fonction `look` de type `flux -> token`)
4. Nous pouvous manger le lexème en attente (function `eat` de type `flux -> unit`)
5. Nous pouvons vérifier que le lexème en attente et le manger (fonction `is` de type `flux -> token -> unit`). C'est une commodité qui s'écrit à l'aide des deux fonctions précédentes :

```
let is flux tok =
  if look flux = tok then eat flux
  else raise Error
```

Pour vérifier que l'entrée est bien un E (*i.e.* qu'il existe une dérivation de E vers w) nous devons de toute façon commencer par appeler `term` (qui correspond à T). Ensuite, au retour de `term` si tout s'est bien passé, nous allons regarder en tête de l'entrée. Il y a alors deux cas :

1. Si nous voyons `+` ou `-`, nous le consommons puis appelons `expr`.
2. Sinon, seule la production $T \to F$ peut s'appliquer, T est déjà reconnu. La fonction `expr` retourne immédiatement.

Bref, nous transformons la grammaire G'' en le programme de la figure 5.4.

Si nous voulons aussi vérifier que toute l'entrée est bien un E, alors nous utilisons le lexème **eof**. On complète la grammaire par une production $S \to E$ **eof** et le programme par une fonction :

```
let start flux = expr flux ; is EOF flux
```

Mais un compilateur ne saurait se contenter de vérifier que son entrée est correcte, il cherche à donner un sens à cette entrée en terme de syntaxe abstraite. C'est assez facile (figure 5.5), au lieu de ne rien faire en cas de succès, on construit l'arbre (de syntaxe abstraite). On notera que l'arbre de syntaxe abstraite produit ne correspond pas aux habitudes $1 - 2 - 3$ est interprété comme $1 - (2 - 3)$. On peut tout de même arriver à produire l'arbre de syntaxe abstraite qui obéit aux conventions usuelles sans boulverser la structure de l'analyseur. Ce programme est donné dans la version web du cours.

En ce qui concerne la réalistion de l'analyse syntaxique, il y a une différence notable entre le schéma fonctionnel de la chaîne de compilation et l'organisation des rapports entre les analyseurs lexical et grammatical. La chaîne de compilation fait apparaître deux phases successives : d'abord l'analyse lexicale qui produit une suite de lexèmes, puis l'analyse grammaticale consomme cette suite. Mais en pratique les appels à l'analyseur lexical sont opérés par l'analyseur grammatical en fonction de ses besoins. Ils sont mieux décrits par le schéma de la figure 5.6. L'analyseur lexical consomme les caractères de l'entrée un par un à la demande, c'est la boite flux qui offre cette interface, et de même l'analyseur lexical montre un flux de lexèmes à l'analyseur grammatical. Le schéma simplifie un peu les choses, mais il est conceptuellement facile d'offrir nos fonctions `look` et `eat` à partir de `next_token` et d'un tampon pouvant contenir un lexème. Le principal impact

Figure 5.4 – Un analyseur écrit à la main

```
let rec expr flux =
  term flux ;
  begin match look flux with
  | (ADD|SUB) -> eat flux ; expr flux
  | _         -> ()
  end

and term flux =
  factor flux ;
  begin match look flux with
  | (MUL|DIV) -> eat flux ; term flux
  | _         -> ()
  end

and factor flux = match look flux with
| INT _  -> eat flux
| LPAR   -> eat flux ; expr flux ; is RPAR flux
| _ -> raise Error
```

ce cette technique en deux flux est que la mémoire nécessaire pour stoker les lexèmes utiles à un instant donné est constante. Si on produisant d'abord disons une liste de tous les lexèmes, l'analyse demanderait nécessairement une taille mémoire proportionnelle à la longeur de l'entrée.

5.3 Analyse LL

Sans tenir du miracle, la démarche de la section précédente semble quand même un peu difficile à appliquer mécaniquement. J'illustre maintenant la production systématique d'un analyseur similaire à partir de la grammaire 5.1, augmentée d'une production $S \rightarrow E$ **eof**. Soit en fait une compilation des grammaires vers les analyseurs. La cible de ce genre de compilation est traditionnellement un automate. Il ne surprendra personne que cet automate est muni d'une pile. Toutefois, je préfère choisir comme cible une classe restreinte de programmes Caml, que je pense du même ordre de puisssance que l'automate traditionnel.

1. Le programme est une définition de fonctions récursives qui prennnent un flux en argument. Il y a une fonction `parseA` par non-terminal A.
2. Chaque fonction doit appeler `look` initialement. et filtrer le résultat de cet appel (par un filtrage `match look flux with`).
3. Les actions du filtrage de `parseA` sont obligatoirement des séquences d'appels aux fonctions définies en 1. et à `is`. Elles s'ecrivent mécaniquement à partir du membre droit α d'une production $A \rightarrow \alpha$ en remplaçant les non-terminaux B par des appels à `parseB` et les terminaux **token** par l'appel `is TOKEN flux`.
4. Les filtrages se terminent obligatoirement par la clause `| _ -> raise Error`.

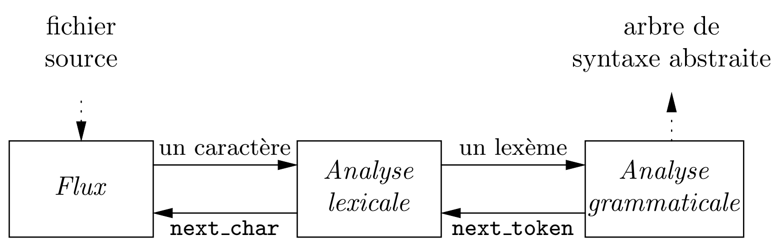
Si cette description vous semble un peu abstraite, vous pouvez dès maintenant consulter l'exemple d'analyseur de la figure 5.8. Ces analyseurs parcourent l'entrée de la gauche vers la droite pour construire implicitement une dérivation gauche (comme précedemment), en ne se décidant qu'à la vue d'un unique lexème. D'où le nom d'automate LL(1), (*Left-to-right parse, Leftmost derivation, (1-token lookahead)*)

La grammaire G est récursive à gauche, nous devons d'abord faire disparaître cette récursion. Il existe une procédure qui fonctionne toujours. Son idée est de remplacer les productions de la

FIGURE 5.5 – Raffinement du programme 5.4 pour construire l'arbre de syntaxe abstraite

```
type ast = Int of int | Binop of binop * ast * ast
and binop = Add | Sub | Mul | Div

let rec expr flux =
  let t1 = term flux in
  match look flux with
  | ADD -> eat flux ; Binop (Add, t1, expr flux)
  | SUB -> eat flux ; Binop (Sub, t1, expr flux)
  | _   -> t1


and term flux =
  let t1 = factor flux in
  match look flux with
  | MUL -> eat flux ; Binop (Mul, t1, term flux)
  | DIV -> eat flux ; Binop (Div, t1, term flux)
  | _   -> t1


and factor flux = match look flux with
| INT i  -> eat flux ; Int i
| LPAR   ->
    eat flux ;
    let t = expr flux in
    begin match look flux with
    | RPAR -> t
    | _    -> raise Error
    end
| _ -> raise Error
```

FIGURE 5.6 – Rapports entre les analyseurs selon des flux.



forme $A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \beta_m$ (où aucun β_j ne commence par A), par deux groupes de productions :

$$A \to \beta_1 A' \mid \ldots \beta_m A' \qquad\qquad A' \to \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \epsilon$$

À condition que tous les α_i soient différents de ϵ (il n'existe pas de production $A \to A$, peu productive de toute façon et donc éliminable), on fabrique une grammaire équivalente et qui n'est plus récursive à gauche. Si nous appliquons cette transformation à une adaptation de G tenant compte des priorités, nous obtenons la grammaire G''' (figure 5.7).

FIGURE 5.7 – Élimination de la récursion à gauche.

$$S \to E\ \textbf{eof}$$

$$E \to E \texttt{+} E \qquad E \to E \texttt{-} E \qquad E \to T$$

$$T \to T \texttt{*} T \qquad T \to T \texttt{/} T \qquad T \to F$$

$$F \to (E) \qquad F \to \textbf{int}$$

$$\Downarrow$$

$$S \to E\ \textbf{eof}$$

$$E \to TE_0 \qquad E_0 \to \texttt{+}\, TE_0 \ \mid \texttt{-}\, TE_0 \ \mid \ \epsilon$$

$$T \to FT_0 \qquad T_0 \to \texttt{*}\, FT_0 \ \mid \texttt{/}\, FT_0 \ \mid \ \epsilon$$

$$F \to (E) \ \mid \ \textbf{int}$$

En fait, l'élimination de la récursion gauche présentée ne suffit pas, car une grammaire peut être cyclique ou récursive à gauche de façon indirecte, c'est à dire qu'il existe des dérivations non triviales de la forme $A \stackrel{*}{\Rightarrow} A\alpha$ ($\alpha = \epsilon$ pour le cycle). Considérez par exemple la grammaire $A \to Bb \mid b$ et $B \to Aa \mid a$. La technique précédente se genéralise et dans tous les cas on peut transformer une grammaire en une grammaire équivalente sans cycles ni récursion à gauche. À titre indicatif, les transformations suivantes règlent le cas de l'exemple :

$A \to Bb \mid b \qquad B \to Aa \mid a$	
$A \to b \qquad A \to Aab \mid ab \qquad B \to Aa \mid a$	substitution de B selon ses productions
$A \to Aab \mid b \mid ab$	production B inutile (départ en A)
$A \to bA' \mid abA' \qquad A' \to abA' \mid \epsilon$	élimination de la récursion gauche de A

Bon, revenons à notre grammaire G''' et à notre pouvoir d'analyse limité, nous devons maintenant au vu d'un seul lexème nous décider parmi toutes les productions possibles associées à un non-terminal donné. Pour ce faire introduisons une fonction FIRST définie des mots α vers les ensembles de non-terminaux et telle que $a \in \text{FIRST}(\alpha)$ si et seulement si il existe une dérivation de la forme $\alpha \stackrel{*}{\Rightarrow} a\beta$. Autrement dit, $\text{FIRST}(\alpha)$ est l'ensemble des non-terminaux qui peuvent se trouver en tête d'une chaîne dérivée à partir de α. Dès lors, si pour une production $A \to \alpha_1 \mid \ldots \alpha_n$, les ensembles $\text{FIRST}(\alpha_1), \ldots, \text{FIRST}(\alpha_n)$ sont deux à deux disjoints, nous saurons nous décider pour un α_i particulier.

Par exemple, dans le cas de la grammaire G'' on a immédiatement $\text{FIRST}((E)) = \{(\}$ et $\text{FIRST}(\textbf{int}) = \{\textbf{int}\}$, nous saurons donc le moment venu d'analyser un F choisir entre utiliser la production $F \to \mathtt{(}E\mathtt{)}$ (on voit **(**) ou la production $F \to \textbf{int}$ (on voit **int**), ou signaler une erreur (dans tous les autres cas). De même $\text{FIRST}(FT_0) = \text{FIRST}(F) = \text{FIRST}(\mathtt{(}E\mathtt{)}) \cup \text{FIRST}(\textbf{int}) = \{\mathtt{(}, \textbf{int}\}$. Et donc, une tentative d'analyser un T se solde par une analyse d'un F, suivie d'un T_0 si on voit **(** ou **int**, et par une erreur sinon. Mais cette belle simplicité se gâte avec T_0, on trouve bien $\text{FIRST}(\mathtt{*}\ FT_0)$ et $\text{FIRST}(\mathtt{/}\ FT_0)$ disjoints mais rien ne nous permet de décider sur le champ entre la production $T_0 \to \epsilon$ (analyse de **int eof** par exemple) et une erreur (analyse de **int int**, commencée en T). Le cas général doit donc sérieusement considérer ϵ. Et de fait nous avons posé $\text{FIRST}(FT_0) = \text{FIRST}(F)$ ce qui est ici exact mais ce qui serait faux si on pouvait avoir $F \stackrel{*}{\Rightarrow} \epsilon$. Par ailleurs, comme nous venons de le voir l'information fournie par FIRST ne suffit pas à trancher le cas des productions de la forme $A \to \epsilon$.

Pour traiter le cas général, on définira FIRST vers $\Sigma \cup \{\epsilon\}$, l'intention étant que $\epsilon \in \text{FIRST}(\alpha)$ traduit l'existence d'une dérivation $\alpha \stackrel{*}{\Rightarrow} \epsilon$.

$$\text{FIRST}(\epsilon) = \{\epsilon\} \qquad\qquad \text{FIRST}(a) = \{a\}$$

$$\text{FIRST}(A) = \text{FIRST}(\alpha_1) \cup \ldots \cup \text{FIRST}(\alpha_n), \text{ si les productions de } A \text{ sont } A \to \alpha_1 \mid \ldots \mid \alpha_n$$

$$\text{FIRST}(\Gamma\alpha) = \text{FIRST}(\Gamma), \text{ si } \epsilon \notin \text{FIRST}(\Gamma)$$

$$\text{FIRST}(\Gamma\alpha) = (\text{FIRST}(\Gamma) \setminus \{\epsilon\}) \cup \text{FIRST}(\alpha), \text{ si } \epsilon \in \text{FIRST}(\Gamma)$$

Ces règles suffisent pour calculer FIRST pour tous les non-terminaux de la grammaire, par point fixe.

Il nous faut ensuite, pour régler le cas des productions $A \to \epsilon$, calculer une nouvelle information. Nous pouvons nous servir de l'ensemble $\text{FOLLOW}(A)$ des non-terminaux qui peuvent, dans les mots intermédiares d'une dérivation, se trouver juste après un non-terminal A. Clairement, une production $A \to \epsilon$ peut s'appliquer quand le lexème courant est dans $\text{FOLLOW}(A)$. Pour définir FOLLOW de façon plus effective, nous pouvons adopter les règles suivantes, support d'un éventuel calcul par point fixe. De chaque décomposition possible de chaque production, une contrainte à satisfaire par FOLLOW est déduite et FOLLOW doit remplir toutes les contraintes possibles :

Production	**Contrainte**
$A \to \alpha B \beta$	$(\text{FIRST}(\beta) \setminus \{\epsilon\}) \subseteq \text{FOLLOW}(B)$
$A \to \alpha B$	$\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$
$A \to \alpha B \beta$, avec $\epsilon \in \text{FIRST}(\beta)$	$\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$

Pour exploiter FIRST et FOLLOW, il est pratique de construire une table d'analyse prédictive. Les lignes de cette table sont indicées par les non-terminaux et ses colonnes par les terminaux. Les cases contiennent des mots α. On remplit la ligne A de la table avec les membre droits de ses productions de la façon suivante :

— Pour toutes les productions $A \to \alpha$, mettre α dans les case de la la colonne a pour tous les a de $\text{FIRST}(\alpha)$.

— En outre, si ϵ est dans $\text{FIRST}(\alpha)$, mettre α dans les cases de la colonne a pour tous les a de $\text{FOLLOW}(A)$.

Si, après examen de toutes les productions, aucune case ne contient plus d'un élément, alors notre analyseur est écrit : les cases vides donneront lieu à des erreurs, les cases ne contenant qu'une entrée, à la poursuite de l'analyse. Pour fixer les idées voici les fonctions FIRST, FOLLOW et la table dans le cas de la grammaire G'''.

	FIRST	FOLLOW
S	(, **int**	
E	(, **int**	), **eof**
E_0	ϵ, +, -	), **eof**
T	(, **int**	), +, -, **eof**
T_0	ϵ, *, /	), +, -, **eof**
F	(, **int**	), +, -, *, /, **eof**

	int	**(**	**)**	**+**	**-**	*****	**/**	**eof**
S	E **eof**	E **eof**						
E	TE_0	TE_0						
E_0			ϵ	+ TE_0	- TE_0			ϵ
T	FT_0	FT_0						
T_0			ϵ	ϵ	ϵ	* FT_0	/ FT_0	ϵ
F	**int**	(E)						

L'analyseur est enfin donné par la figure 5.8. Le code a le cachet du code engendré automatiquement, un certain nombre d'optimisations évidentes sont possibles. Ceci ne doit pas nous cacher que la table d'analyse prédictive peut être utile quand on écrit un analyseur à la main.

Figure 5.8 – Un analyseur LL(1) des expressions arithmétiques

```
let rec start flux = match look flux with
| INT _|LPAR -> expr flux ; is EOF flux
| _          -> raise Error

and expr flux = match look flux with
| INT _|LPAR -> term flux ; expr0 flux
| _          -> raise Error

and expr0 flux =  match look flux with
| ADD      -> is ADD flux ; term flux ; expr0 flux
| SUB      -> is SUB flux ; term flux ; expr0 flux
| RPAR|EOF -> ()
| _        -> raise Error

and term flux = match look flux with
| INT _|LPAR -> facteur flux ; term0 flux
| _          -> raise Error

and term0 flux =  match look flux with
| MUL              -> is MUL flux ; facteur flux ; term0 flux
| DIV              -> is DIV flux ; facteur flux ; term0 flux
| RPAR|EOF|ADD|SUB  -> ()
| _                -> raise Error

and factor flux = match look flux with
| INT i  -> is (INT i)
| LPAR   -> is LPAR ; expr flux ; is RPAR flux
| _      -> raise Error
```

Si la construction de l'analyseur échoue, alors la grammaire G présentée n'est pas LL(1). Cela peut provenir d'une grammaire ambigüe (car toutes les grammaires LL(1) sont non-ambigües) mais pas forcément. Construisons par exemple la table de la grammaire G'' (figure 5.3) qui nous avait servi de point de départ pour écrire un analyseur à la main, en oubliant soustraction et division :

$$S \to E\ \mathbf{eof} \qquad E \to T + E \qquad E \to T \qquad T \to F * T \qquad T \to F \qquad F \to (E) \qquad F \to \mathbf{int}$$

	int	(	)	+	*	**eof**
S	E **eof**	E **eof**				
E	$T, T + E$	$T, T + E$				
T	$F, F * T$	$F, F * T$				
F	**int**	(E)				

Certaines cases contiennnent deux mots, la grammaire G'' n'est donc pas LL(1). Pourtant, G'' est non-ambigüe.

Mais ici (ce n'est évidemment pas vrai en général) les mots qui occupent la même case ont un préfixe (non-vide) commun, par exemple T pour T et $T + E$. On peut factoriser ce préfixe commun en remplaçant les deux productions $E \to T \mid T + E$ par trois productions $E \to TE_0$ et $E_0 \to \epsilon \mid + E$. Ce procédé de *factorisation gauche* est général. Ici la grammaire résultante et la table seront :

$$S \to E\ \mathbf{eof} \qquad E \to TE_0 \qquad E_0 \to \epsilon \qquad E_0 \to + E \qquad T \to FT_0 \quad T_0 \to \epsilon \qquad T_0 \to * T$$

$$F \to \mathbf{int} \qquad\qquad F \to (E)$$

	int	(	)	+	*	**eof**
S	E **eof**	E **eof**				
E	$T + E_0$	$T + E_0$				
E_0			ϵ	$+E$		ϵ
T	$F * T_0$	$F * T_0$				
T_0			ϵ		$*T$	ϵ
F	**int**	(E)				

La grammaire transformée est donc LL(1). Notons que pour écrire un analyseur prédictif à la main, la transformation n'est pas strictement nécessaire. Il suffit de se donner le pouvoir d'examiner les lexèmes à l'intérieur du corps des fonctions. De fait, l'analyseur de la figure 5.4 est moralement LL(1), modulo la détection des erreurs transformée en lecture du plus long préfixe possible correct dans le flux.

Dans le cas plus général des grammaires des langages de programmation, élimination de la récursion gauche et factorisation gauche ne suffisent pas toujours pour produire une grammaire LL(1) et donc un analyseur ; et ceci même lorsque la grammaire de départ est non-ambigüe. Par ailleurs, ces transformations sont peu pratiques lorsque l'on veut un analyseur qui produit un arbre de syntaxe abstraite.

On généralise le principe de l'examen d'un lexème à celui de k lexèmes. Les tables qui en résultent sont potentiellement énormes et cette technique LL(k) n'est pas utilisée en pratique. Dans un analyseur écrit à la main, on ne se privera pas d'examiner plus d'un lexème dans certains cas particuliers, tout en restant prudent.

5.4 Analyse montante (*bottom-up parsing*)

Autant la technique LL peut nous guider lors de l'écriture d'analyseurs, autant elle n'est pas adaptée à la production automatique d'analyseurs (puissance réduite, transformations de la grammaire nécessaires). Heureusement, il existe une autre technique dite LR(1), qui procède toujours en lisant les lexèmes de gauche à droite (d'où le L), cherche cette fois une dérivation droite (d'où le R) et se décide au vu d'un lexème d'avance (d'où le 1).

5.4.1 Automates shift-reduce

Dans la présentation traditionelle de l'analyse montante, une certaine sorte d'automate est chargé de l'analyse. Les automates de ce style consomment un mot dans un flux et utilisent une pile auxiliaire de symboles de la grammaire, ils peuvent procéder à deux actions :

— *shift*, *i.e.* consommer et empiler un lexème, ou

— *reduce*, *i.e.* réduire une production. Cela revient à appliquer une production sur le sommet (partie droite) de la pile.

L'automate démarre avec une pile vide, procède à ses actions comme il l'entend, jusqu'à se retrouver bloqué. Alors, si la pile contient le symbole de départ et lui seul, il y a succès, sinon il y a échec. Grâce au renversement opéré par la pile, ces automates procèdent à l'analyse selon une dérivation droite.

Pour s'en convaincre, considérons comment un tel automate produit les deux dérivations droites possibles du mot de non-terminaux **1 + 2 * 3** dans la grammaire ambigüe G (figure 5.1) des expressions arithmétiques (deux premiers examples de la figure 5.9). Pour retrouver les dérivations, il suffit, de procéder à l'envers de l'analyse, le mot intermédiaire à chaque étape est la concaténation de la pile et du flux et on applique la production utilisée à chaque étape *reduce* au sous-mot correspondant au sommet de la pile (la limite entre pile et flux est indiquée par •). On peut aussi remplacer les non-terminaux de la pile par les arbres de dérivation qui leur correspondent. On voit alors clairement que l'arbre final est construit à partir des feuilles et de la gauche vers la droite, le renversement opéré pour retrouver la dérivation expliquant que cette dernière est droite.

L'étape cruciale qui détermine le choix de l'une ou l'autre des dérivations droites est signalée (un *shift* de `*` contre un *reduce* de $E \to E + E$). Si nous levons l'ambiguïté en transformant la grammaire, alors seule une de ces deux étapes sera possible. Pour comprendre comment l'automate peut se décider à coup sûr entre *shift* et *reduce*, il vaut mieux s'affranchir de cette ambigüité. Donnons nous donc la grammaire non-ambigüe G' des expressions arithmétiques (avec les productions $E \to E + T$ et $T \to T * F$ entre autres), et examinons comment l'automate reconnaît **1 + 2 * 3** selon cette grammaire (dernier exemple de la figure 5.9). Il apparaît alors clairement d'abord que tous les entiers sont d'abord shiftés (empilés), puis réduits. Mais l'entier **1** est réduit en E (en trois étapes), tandis que **2** est réduit en T et **3** seulement en F. C'est certainement le bon choix dans tous les cas, car sinon, il y aurait une erreur plus tard. La dernière de ces décisions ne s'explique certainement pas uniquement par la fin du flux (car la réduction à E s'impose dans le cas d'un seul entier dans l'entrée). En revanche, on la comprend mieux si on examine la pile au moment de choisir de réduire trois symboles selon $T \to T * F$ plutôt qu'un seul selon $T \to F$: le sommet de pile, invite l'automate à en faire le maximum. Remarquons aussi, à l'étape cruciale (distinguée), que la présence de `*` en tête du flot invite à ne pas réduire $E + T$ en attente sur la pile.

5.4.2 Programmation en Caml d'un analyseur montant

L'intervention d'un automate obscurcit un peu le propos. Comme pour l'analyse descendante on se propose donc d'écrire quelques analyseurs à la main avant d'automatiser le procédé. On considère une fois encore une grammaire ambigüe des expressions arithmétiques en la simplifiant beaucoup :

$$S \to E\ \mathbf{eof} \qquad\qquad E \to E + E \qquad\qquad E \to \mathbf{int}$$

On aura besoin d'un type de tous les symboles de la grammaire, le voici :

```
type symbol = E | Terminal of token
```

L'automate shift/reduce sera réalisé par une bête fonction recursive, qui se décicidera au vu du premier lexème et de la pile de symboles (ou plus exactement de quelques éléments de son sommet) : `auto`, de la forme :

```
let rec auto stack flux = match look flux, stack with
(* Accepter l'entrée (qui est finie) *)
| EOF, [E] -> ()
  ...
```

Figure 5.9 – Fonctionnnement de l'automate *shift-reduce*

Pile	Flux	Action	Production	Mot
	$\mathbf{1} + \mathbf{2} * \mathbf{3}$	*shift*		$\bullet\ \mathbf{1} + \mathbf{2} * \mathbf{3}$
$\mathbf{1}$	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$E \to \mathbf{int}$	$\underline{\mathbf{1}} \bullet + \mathbf{2} * \mathbf{3}$
E	$+ \mathbf{2} * \mathbf{3}$	*shift*		$E \bullet + \mathbf{2} * \mathbf{3}$
$E +$	$\mathbf{2} * \mathbf{3}$	*shift*		$E + \bullet\ \mathbf{2} * \mathbf{3}$
$E + \mathbf{2}$	$* \mathbf{3}$	*reduce*	$E \to \mathbf{int}$	$E + \underline{\mathbf{2}} \bullet * \mathbf{3}$
$E + E$	$* \mathbf{3}$	*shift*		$E + E \bullet * \mathbf{3}$
$E + E *$	$\mathbf{3}$	*shift*		$E + E * \bullet\ \mathbf{3}$
$E + E * \mathbf{3}$		*reduce*	$E \to \mathbf{int}$	$E + E * \underline{\mathbf{3}} \bullet$
$E + E * E$		*reduce*	$E \to E * E$	$E + \underline{E * E} \bullet$
$E + E$		*reduce*	$E \to E + E$	$\underline{E + E} \bullet$
E				E

Pile	Flux	Action	Production	Mot
	$\mathbf{1} + \mathbf{2} * \mathbf{3}$	*shift*		$\bullet\ \mathbf{1} + \mathbf{2} * \mathbf{3}$
$\mathbf{1}$	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$E \to \mathbf{int}$	$\underline{\mathbf{1}} \bullet + \mathbf{2} * \mathbf{3}$
E	$+ \mathbf{2} * \mathbf{3}$	*shift*		$E \bullet + \mathbf{2} * \mathbf{3}$
$E +$	$\mathbf{2} * \mathbf{3}$	*shift*		$E + \bullet\ \mathbf{2} * \mathbf{3}$
$E + \mathbf{2}$	$* \mathbf{3}$	*reduce*	$E \to \mathbf{int}$	$E + \underline{\mathbf{2}} \bullet * \mathbf{3}$
$E + E$	$* \mathbf{3}$	*reduce*	$E \to E + E$	$\underline{E + E} \bullet * \mathbf{3}$
E	$* \mathbf{3}$	*shift*		$E \bullet * \mathbf{3}$
$E *$	$\mathbf{3}$	*shift*		$E * \bullet\ \mathbf{3}$
$E * \mathbf{3}$		*reduce*	$E \to \mathbf{int}$	$E * \underline{\mathbf{3}} \bullet$
$E * E$		*reduce*	$E \to E * E$	$\underline{E * E} \bullet$
E				E

Pile	Flux	Action	Production	Mot
	$\mathbf{1} + \mathbf{2} * \mathbf{3}$	*shift*		$\bullet\ \mathbf{1} + 2 * \mathbf{3}$
$\mathbf{1}$	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$F \to \mathbf{int}$	$\underline{\mathbf{1}} \bullet + 2 * \mathbf{3}$
F	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$T \to F$	$\underline{F} \bullet + 2 * \mathbf{3}$
T	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$E \to T$	$\underline{T} \bullet + 2 * \mathbf{3}$
E	$+ \mathbf{2} * \mathbf{3}$	*shift*		$E \bullet + 2 * \mathbf{3}$
$E +$	$\mathbf{2} * \mathbf{3}$	*shift*		$E + \bullet\ 2 * \mathbf{3}$
$E + \mathbf{2}$	$* \mathbf{3}$	*reduce*	$F \to \mathbf{int}$	$E + \underline{2} \bullet * \mathbf{3}$
$E + F$	$* \mathbf{3}$	*reduce*	$T \to F$	$E + \underline{F} \bullet * \mathbf{3}$
$E + T$	$* \mathbf{3}$	*shift*		$E + T \bullet * \mathbf{3}$
$E + T *$	$\mathbf{3}$	*shift*		$E + T * \bullet\ \mathbf{3}$
$E + T * \mathbf{3}$		*reduce*	$F \to \mathbf{int}$	$E + T * \underline{\mathbf{3}} \bullet$
$E + T * F$		*reduce*	$T \to T * F$	$E + \underline{T * F} \bullet$
$E + T$		*reduce*	$E \to E + T$	$\underline{E + T} \bullet$
E				$E \bullet$

```
(* Petite fonction pour éviter d'écrire 10 fois la même chose *)
and shift stack flux =
  let tok = look flux in
  eat flux ;
  auto (Terminal tok::stack) flux
```

La grammaire est simplifiée mais elle presente encore une ambiguité typique de la grammaire de départ. En effet, on peut voir $E + E + E$ comme $(E + E) + E$ ou comme $E + (E + E)$. Lors de l'analyse, le choix va se poser quand on aura déja reconnu $E + E$ et que le premier lexème du flux est `+`. Pour obtenir la première inteprétation il faudra réduire, tandis que pour obtenir la seconde interpétation il faudra shifter. Nous décidons par exemple de faire pencher les arbres à gauche, c'est à dire que $E + E + E$ s'interprète comme $(E + E) + E$. Nous aurons donc la règle :

```
let rec auto stack flux = match look flux, stack with
  ...
(* Reduction de E → E + E *)
| ADD, E::Terminal ADD::E::rem -> auto (E::rem) flux
 ...
```

Par ailleurs, si le sommet de la pile n'est pas de la forme ci-dessus, il faut shifter le terminal « `+` ».

```
| ADD, _ -> shift stack flux
```

Mais, compte tenu de la nature de la grammaire, on peut préciser ce que l'on entend par « *toutes les autres situations* ». Ici, E sera tout seul sur la pile. On écrira donc plutôt.

```
| ADD, [E] -> shift stack flux
```

Interessons nous maintenant aux entiers. Dans toutes les situations il faut les shifter, puis les réduire. On serait donc tenté d'écrire :

```
| INT _, _ -> shift stack flux
| _, Terminal (INT _)::rem -> auto (E::rem) flux
```

Mais ici encore, on souhaite être beaucoup plus précis.

```
| INT _, (Terminal ADD::E::_|[] ->) shift stack flux
| (ADD|EOF), Terminal (INT _)::rem -> auto (E::rem) flux
```

Bref, voici la fonction `auto` complète.

```
let rec auto stack flux = match look flux, stack with
(* Que faire avec un seul E ? *)
| EOF, [E] -> ()
| ADD, [E] -> shift stack flux
(* Réduction de E → E + E *)
| (ADD|EOF), E::Terminal ADD::E::rem -> auto (E::rem) flux
(* Réduction de E → int *)
| (ADD|EOF), Terminal (INT _)::rem -> auto (E::rem) flux
(* Shift de int *)
| INT _, (Terminal ADD::E::_|[]) -> shift stack flux
(* N'importe quoi d'autre est une erreur *)
| _ -> raise Error
```

Nous choisissons maintenant de compliquer un peu notre grammaire, en lui ajoutant une production $E \to (E)$. En refléchissant aux couples premier symbole du flux, sommet de la pile on obtient finalement cet automate :

```
let rec auto stack flux = match look flux, stack with
(* Ouf ! *)
| EOF, [E] -> ()
| (ADD|EOF|RPAR), E::Terminal ADD::E::rem -> auto (E::rem) flux
```

```
| ADD, ([E]|E::Terminal LPAR::_) -> shift stack flux
| INT _, (Terminal ADD::E::_|Terminal LPAR::_|[]) -> shift stack flux
| (ADD|EOF|RPAR), Terminal (INT _)::rem -> auto (E::rem) flux
(* Avec des parenthèses *)
| RPAR, E::Terminal  LPAR::_ -> shift stack flux
| (ADD|RPAR|EOF), Terminal RPAR::E::Terminal  LPAR::rem -> auto (E::rem) flux
| LPAR, (Terminal ADD::E::_|Terminal LPAR::_|[]) -> shift stack flux
(*  N'importe quoi d'autre  est  une erreur *)
| _ -> raise Error
```

Bon, nous sommes arrivés à construire un analyseur shift/reduce à la main. Mais il est clair que ce sera difficile dans le cas général. Par ailleurs on souhaite éviter les analyses répétées de la pile (même si elles sont particulièrement faciles à programmer en Caml).

5.4.3 Analyse LR(1)

Dans les deux sections précédentes nous avons argumenté qu'un analyseur LR se décidait au vu du lexème en tête du flux et d'une fraction de sa pile (vers le sommet) L'idée de base des analyseurs LR(1) est d'assurer le contrôle de l'automate *shift-reduce* à l'aide d'un automate fini. Un état donné représente donc un état d'avancement de l'analyse. Dans le cadre qui nous intéresse cet état est représente par un ensemble de *productions pointées*, de la forme $A \to \alpha \bullet \beta$ où $A \to \alpha\beta$ est une production de la grammaire. Reprenons par exemple le cas de la grammaire :

$$S \to E\ \mathbf{eof} \qquad\qquad E \to E + E \qquad\qquad E \to \mathbf{int} \qquad\qquad E \to (E)$$

Initialement on a rien empilé, et on veut reconnaître E**eof**. L'état initial contient donc $S \to\bullet\ E\mathbf{eof}$. Pour identifier E dans le flux, nous devons identifier l'un des membre droits des productions de E. Soit un état initial numéro 0 :

$$\{\, S \to\bullet\ E\mathbf{eof},\quad E \to\bullet\ E + E,\quad E \to\bullet\ \mathbf{int},\quad E \to\bullet\ (E)\,\}$$

Le procédé appliqué dit de fermeture est décrit précisément par la suite. Il revient à ajouter toutes les productions pointées $A \to\bullet\ \alpha$ dès que $\ldots \to \ldots \bullet A \ldots$ apparaît dans l'état.

Admetons maintenant que E a été reconnu, la pile contiendra donc un E et, compte tenu de notre état initial le flux sera **eof** ou + E. Soit un état numéro 1 (déjà fermé) :

$$\{\, S \to E \bullet \mathbf{eof},\quad E \to E \bullet\!+ E\,\}$$

Pour pouvoir passer le contrôle à cet état après la reconnaissance de E dans le flux initial, on utilise la pile de l'automate *shift-reduce*, c'est à dire que lorsque l'automate passe dans un état quelconque, cet état est systématiquement empilé, en même temps que le symbole de la grammaire qui était empilé aux sections précédentes. Ici on doit donc supposer que l'état 0 est déjà sur la pile. Ainsi, lorsque l'automate effectuera plus tard l'ultime action *reduce* qui empile E il trouvera l'état initial sur la pile et saura effectuer une transition de l'état initial à l'état ci-dessus.

Mais pour le moment E n'est pas reconnu, supposons que le flux débute par un entier, alors cet entier est shifté et l'automate passe dans l'état numéro 2 suivant (qui est également empilé)

$$\{\, E \to \mathbf{int} \bullet \,\}$$

Le point est à la fin, nous pouvons réduire (en fait il faudrait à priori regarder le premier lexème du flux, qui ici doit être + ou **eof**, mais bon). La réduction revient ici à dépiler **int** et à empiler E à la place, l'état $E \to \mathbf{int} \bullet$ est dépilé et on assure la transition vers l'état 1 en fonction de l'état en sommet de pile (ici l'état 0) et du non-terminal reconnu (ici E). L'état numéro 1 est maintenant en sommet de pile et il saura quoi faire selon le premier lexème du flux (annoncer une reconnaisance réussie en cas de **eof**, shifter un + ou signaler une erreur autrement).

Voici maintenant le procédé général de construction de l'automate de contrôle. Soit une grammaire dont le symbole de départ est S_g. Le symbole de départ de la grammaire considérée ensuite est un nouveaux non-terminal S. On ajoute également une production un peu spéciale $\rightarrow S$ dite axiome, ainsi qu'une production $S \rightarrow S_g$ \$, où \$ est un non-terminal réservé pour signaler la fin de l'entrée. L'automate *shift-reduce* est raffiné en un automate LR :

— Il y a en fait quatre actions, *shift*, *reduce*($A \rightarrow \alpha$) (la production réduite est indiquée), *accept* (succès) et *error* (échec).
— La pile est maintenant composée de couples (Δ_i, s_i) d'états et de symboles de la grammaire, sauf le fond de pile s_0 qui est seulement l'état initial de l'automate.
— L'automate est déterminé par deux fonctions :
  — $\mathsf{goto}(s, \Delta)$ vers les états, ce sont les transitions de l'automate de contrôle, indicées par tous les symboles de la grammaire.
  — Et $\mathsf{action}(s, a)$ vers les actions, ce sont des actions de l'automate *shift-reduce*, commandées par l'automate de contrôle au vu du premier lexème du flux.

Si sa pile est $s_0, (\Delta_1), \ldots (\Delta_m, s_m)$, l'automate consulte le lexème a en attente et par cas sur la valeur de $\mathsf{action}(s_m, a)$ effectue les actions suivantes.

shift Le lexème est consommé et $(a, \mathsf{goto}(s_m, a))$ est empilé.

reduce$(A \rightarrow \alpha)$ L'automate dépile l eléments, où l est la longueur de α et $(A, \mathsf{goto}(s_{m-l}, A))$ est empilé.

accept* ou *error L'automate signale le succès ou un échec et s'arrête.

Remarquez que, en cas de *shift*, une transition de l'automate de contrôle est effectuée et que l'état atteint est empilé. En cas de *reduce*, l'état courant de l'automate de contrôle est oublié et on opère de même, cette fois à partir de l'état qui prévalait avant d'engager la reconnaissance de la production réduite.

Un état I de l'automate de contrôle est un ensemble de configurations C. une configuration C est une paire composée,

1. d'une production pointée $A \rightarrow \alpha \bullet \beta$, où $A \rightarrow \alpha\beta$ est une production,
2. et du prochain lexème possible a après $\alpha\beta$.

Une configuration C décrit un état courant de l'automate *shift-reduce* sous-jacent, avec α en sommet de pile et un flux dont le début dérive de βa.

La fermeture d'un ensemble de configurations I est le plus petit ensemble contenant I et satisfaisant

$$((A \rightarrow \alpha \bullet B\beta, a) \in I \ \wedge \ B \rightarrow \gamma \in G \ \wedge \ b \in \mathrm{FIRST}(\beta a)) \Longrightarrow (B \rightarrow\bullet \ \gamma, b) \in I$$

L'intuition est que nous cherchons à identifier (en un même état de l'automate de contrôle) certaines configurations de l'automate *shift-reduce*. Dans une configuration $A \rightarrow \alpha \bullet B\beta$, une partie α est déjà reconnue et une autre $B\beta$ devrait l'être. Donc, il faut certainement s'attendre à reconnaître aussi γ pour toutes les productions $B \rightarrow \gamma$; la deuxième partie de la configuration sert à affiner cette première règle à l'aide des non-terminaux qui peuvent se trouver après B ($\mathrm{FIRST}(\beta a)$ ne peut pas contenir ϵ) et donc après γ. L'ajout du symbole a augmente sensiblement le pouvoir discriminant des états, notamemt dans le cas des productions complètement reconnues (de la forme $A \rightarrow \alpha \bullet$).

Les transitions sont définies ainsi :

$$\mathsf{goto}(I, \Delta) = \text{fermeture}(\{(A \rightarrow \alpha\Delta \bullet \beta, a) \mid (A \rightarrow \alpha \bullet \Delta\beta, a) \in I\})$$

Autrement dit, on passe de I à J en faisant avancer de point $\bullet$ d'un cran. Le symbole Δ change de statut il prend maintenant part à un membre droit de production reconnu. Notons que si Δ est un non terminal a, alors l'automate *shift-reduce* devra effectuer un *shift*. Et que dans tous les cas l'automate ce contrôle effectue une transition de I à J. L'autre cas, (Δ est un non-terminal A) correspond à la transition de l'automate de contrôle à effectuer après la réduction d'une production de membre gauche A.

L'état initial est la fermeture de $\{(\rightarrow\bullet \ S, a) \mid a \in \Sigma\}$. Les états sont les ensembles de configurations fermées non-vides atteignables par une suite de transitions arbitraires à partir de l'état initial. Cette construction ressemble fort, en plus compliqué dans les détails, à la déterminisation d'un automate fini dont les états seraient les configurations.

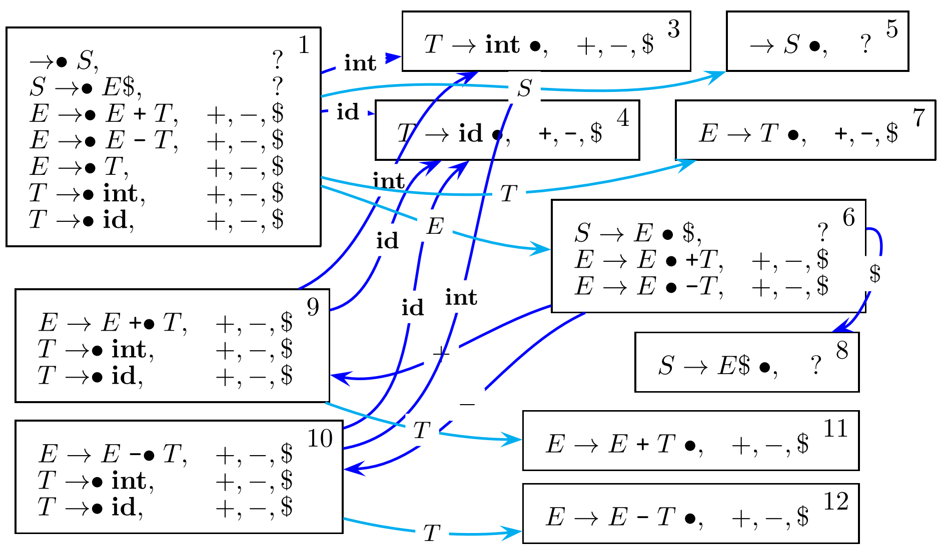
Une fois construit le graphe décrit ci dessus, on en tire action.

1. Si $(A \to \alpha \bullet a\beta, b) \in I$ et $\mathsf{goto}(I, a) = J$, alors $\mathsf{action}(I, a) = \mathit{shift}$ (on peut représenter simultanément $\mathsf{goto}(I, a)$ en écrivant $\mathit{shift}(J)$).,
2. Si $(A \to \alpha \bullet, a) \in I$, alors $\mathsf{action}(I, a) = \mathit{reduce}(A \to \alpha)$. Sauf, si $A \to \alpha$ est l'axiome $\to S$, auquel cas $\mathsf{action}(I, a) = \mathit{accept}$.

Si $\mathsf{action}(I, a)$ n'est pas défini par ces deux règles, alors on a $\mathsf{action}(I, a) = \mathit{error}$. Si il y a des conflits, c'est à dire plusieurs valeurs possibles pour $\mathsf{action}(I, a)$, alors la grammaire G n'est pas LR(1). Cela peut provenir d'une grammaire ambigüe mais aussi (plutôt rarement en pratique) d'une grammaire non-ambigüe suffisamment compliquée.

En pratique, les fonctions action et goto sont réalisées par des tables, des états de l'automate de contrôle par les symboles de la grammaire pour action, et des états par les non-terminaux pour goto. Ces tables seront normalement assez creuses et peuvent compactées.

Figure 5.10 – exemple de graphe des états LR(1)



Sans détailler plus avant, la figure 5.10 donne le graphe pour la grammaire suivante des sommes et des différences arithmétiques (symbole de départ E) :

$$E \to E + T \qquad E \to E - T \qquad E \to T \qquad T \to \mathbf{id} \qquad T \to \mathbf{int}$$

Dans les états, notez la notation des configurations de même productions pointée en une seule ligne montrant un ensemble de prochains lexèmes possibles. Voir aussi l'animation Postscript de la construction du graphe, dans la version web du cours.

Dans cette figure, on constate que l'on peut enlever la seconde partie des configurations, les ensembles de productions pointées suffisent à définir les états. Les générateurs d'analyseurs grammaticaux les plus répandus (`yacc`, `bison`, etc.) ne suivent pas exactement la constructions des tables LR(1), car ces tables sont de taille importante en pratique. Le graphe est construit comme dans le cas LR(1), mais on efface ensuite les ensembles de prochains lexèmes possibles des états et on fusionne les états ainsi identifiés avant de construire les tables de l'analyseur. Les tables sont alors plus petites, tandis que la puissance de compilation n'est pas excessivement diminuée. Cette technique est dénommée LALR(1) (pour *Look-Ahead* LR(1)). Si la production des tables s'opère sans conflit, la grammaire compilée est dite LALR(1).

Enfin, on définit assez naturellement les grammaires LR(k) en considérant les automates de contrôle qui se decident à l'aide de k lexèmes d'avance. Les tables deviennent potentiellement énormes.

Du point de vue du la puissance de toutes les techniques vues, on a les relations suivantes, où l'acronyme de chaque technique désigne l'ensemble des grammaires qu'elle peut traiter.

$$\mathrm{LL}(k) \subset \mathrm{LL}(k+1) \qquad \mathrm{LR}(k) \subset \mathrm{LR}(k+1) \qquad \mathrm{LL}(k) \subset \mathrm{LR}(k) \qquad \mathrm{LALR}(1) \subset \mathrm{LR}(1)$$

Toutes les grammaires citées sont non-ambigües. Compte tenu de la description donnée dans ce cours, l'inclusion $\mathrm{LL}(1) \subset \mathrm{LR}(1)$ se comprend, si on considère qu'un automate LR(1) dispose de plus d'informations qu'un programme LL(1). Les programmes doivent deviner le membre droit à choisir en fonction d'un lexème d'avance, tandis que l'automate ne prend réellement sa décision qu'une fois que le membre droit est en pile, en fonction toujours du lexème d'avance. Cette remarque ne remplace bien entendu pas une démonstration.

Dans la pratique, les langages de programmation ont des grammaires LALR(1) et leurs tables sont de taille raisonnable. Réciproquement, comme les générateurs d'analyseurs disponibles reposent sur la technique LALR(1), les langages de programmation tendent à avoir des grammaires LALR(1). Les grammaires LL ne sont par pour autant absentes des langages de programmation.

— Pascal a une grammaire essentiellement LL(2 ?). On y arrive assez facilement en faisant débuter les instructions par des mots-clés différents et en exigeant des constructions fermées (typiquement `begin`… `end`, `var`… `begin`). Pour les expressions, le problème des opérateurs est soluble en pratique, car bien balisé. Pascal a été conçu dans cet esprit.
— Il existe des outils plus souples que les compilateurs traditionnels de grammaires, tels `camlp4`, reposant sur les grammaires LL, quand même plus simples à comprendre.

5.5 `ocamlyacc` en pratique

L'utilisation d'un outil de compilation des grammaires en analyseurs a principalement deux avantages, d'une part l'écriture des analyseurs (et surtout leur modification) devient facile, d'autre part, les conflits détectés proviennent le plus souvent de grammaires ambigües qui sont donc détectables facilement.

L'outil `ocamlyacc` transforme un fichier source *Nom*`.mly` contenant une description de grammaire, en deux fichiers *Nom*`.ml` et *Nom*`.mli` contenants l'analyseur grammatical.

En effet la description de la grammaire comprend celle de ses non-terminaux, cette dernière est traduite en un type des lexèmes, de nom conventionnel `token`. Le fichier *Nom*`.mli` exporte ce type des lexèmes, au profit de l'analyseur lexical. Pour chaque symbole de départ S, l'analyseur sera une fonction homonyme prenant en argument un analyseur syntaxique de type `(Lexing.lexbuf -> `*`Nom`*`.token)` et un flux de type `Lexing.lexbuf`. La figure 5.11 donne un exemple de spécification de grammaire pour `ocamlyacc`. Il s'agit encore une fois de la grammaire ambigüe des expressions arithmétiques, (avec la négation ou moins « unaire » en plus) ; l'intention est ici d'écrire un analyseur syntaxique qui rend un arbre de syntaxe abstraite (de type `Ast.t`). Cet exemple fait apparaître les trois sections des fichiers `.mly`.

1. La première section, de `%{` à `}%`, dite prélude, contient du code source Caml, à mettre en tête du fichier `.ml` produit. Ici on se contente d'ouvrir le module `Ast`, réputé contenir la définition du type des arbres de syntaxe abstraite.

```
type t = Int of int | Binop of binop * t * t
and binop = Add | Sub | Mul | Div
```

2. Ensuite vient une section de déclarations destinées à `ocamlyacc`, il s'agit ici de la déclaration des lexèmes (notez la syntaxe tordue de la déclaration du lexème `INT` qui prend un argument entier), puis du point d'entrée et de son type. Cette section se termine avec le mot-clé (de `ocamlyacc`) `%%`. On notera que les commentaires de cette section sont ceux de C (`/*` … `*/`). En effet, `ocamlyacc` est essentiellement `yacc`, qui est plutôt orienté vers C.
3. Enfin, vient la section qui définit les productions de la grammaire, on prendra un peu garde à la syntaxe, le non-terminal est suivi de « `:` » ses membres droits sont séparées par « `|` » et le dernier membre droit est suivi de « `;` ». Une production vide, se note par un membre droit vide (il n'y en a pas ici). Les actions sont données entre accolades, à raison d'une par membre droit, elles sont évaluées si le membre droit est réduit et constituent la valeur du

Figure 5.11 – Un exemple `arith.mly` de source `ocamlyacc`

```
%{
open Ast
%}
/* Déclaration des lexèmes */
%token LPAR RPAR
%token ADD SUB MUL DIV
%token <int> INT
%token EOF

/* Point d'entrée */
%start expr
%type <Ast.t> expr
%%

expr:
  expr1 EOF {$1}
;

expr1:
  expr1 ADD expr1 {Binop (Add,$1, $3)}
| expr1 SUB expr1 {Binop (Sub,$1, $3)}
| expr1 MUL expr1 {Binop (Mul,$1, $3)}
| expr1 DIV expr1 {Binop (Div,$1, $3)}
| SUB expr1       {Binop (Sub, Int 0, $2)}
| INT             {Int $1}
| LPAR expr1 RPAR {$2}
;
```

non-terminal correspondant. Ici, les actions consistent comme souvent à construire l'arbre de syntaxe abstraite. Dans une action, on peut faire référence aux valeurs des symboles du membre droit par un entier préfixé de ≪ `$` ≫. Ces valeurs sont les résultats de l'analyse des non-terminaux et l'argument des terminaux pour ceux qui en ont un, comme montré par la production $E \to \textbf{int}$.

La compilation par ≪ `ocamlyacc arith.mly` ≫ se solde par ≪ `20 shift/reduce conflicts` ≫. Et c'est bien normal, car la grammaire est assez ambigüe. Un analyseur est quand même produit, `ocamlyacc` ≪ résolvant ≫ les conflits selon ses règles (*shift* gagne sur *reduce*, et entre deux *reduce*, celui de la production qui apparaît en premier dans le source gagne). On ne peut pas laisser `ocamlyacc` resoudre les conflits pour nous, sauf dans les rares cas où on comprend ce qui se passe et où on estime que c'est satisfaisant. L'automate produit et le détail des conflits sont donnés par un fichier `arith.output` créé par `ocamlyacc` si on lui passe l'option `-v`.

Mais ici, reportons nous d'abord aux deux premières exécutions de l'automate à pile de la figure 5.9 qui, rapellons le, décrivent deux dérivations de la même expression arithmétique dans la même grammaire ou presque. À l'étape critique signalée, les deux automates ont la pile E `+` E et `*` est le lexème en tête du flux. Selon les règles usuelles, il ne faut pas réduire la somme et `*` doit être shifté. Dans le fichier `arith.output` on trouve en particulier le détail des conflits de la réduction des sommes (attention, les entiers des `shift` sont des états de l'automate, tandis que ceux des `reduce` sont des numéros donnés aux productions) :

```
16: shift/reduce conflict (shift 10, reduce 2) on ADD
16: shift/reduce conflict (shift 11, reduce 2) on SUB
```

```
16: shift/reduce conflict (shift 12, reduce 2) on MUL
16: shift/reduce conflict (shift 13, reduce 2) on DIV
state 16
expr1 : expr1 . ADD expr1  (2)
expr1 : expr1 ADD expr1 .  (2)
expr1 : expr1 . SUB expr1  (3)
expr1 : expr1 . MUL expr1  (4)
expr1 : expr1 . DIV expr1  (5)
```

On retrouve au passage les ensembles de productions prointées qui définissent les états. Un autre conflit intéressant apparaît, celui entre le *shift* de `+` et le *reduce* de la somme, issu de la confrontation entre les deux premières productions pointées. Cette ambigüité, dite d'associativité, se révèle sur le mot de la grammaire E `+` E •`+` E à voir comme (E `+` E) `+` E (réduire) ou comme E `+` (E `+` E) (shifter) ; et là, l'interprétation usuelle commande de réduire.

Nous pourrions bien évidemment réécrire un peu la grammaire, mais `ocamlyacc` fournit un mécanisme de priorité bien plus pratique. Nous pouvons associer des niveaux de priorité *aux lexèmes* (plusieurs lexèmes peuvent avoir la même priorité). Ces priorités s'étendent toutes seules *aux productions*, la priorité d'une production étant celle de son dernier lexème. Dans un conflit *shift*/*reduce*, les priorités du lexème à shifter et de la production à réduire sont comparées, le plus fort gagne (silencieusement). Dans un conflit *reduce*/*reduce*, les priorités des deux règles sont comparées. On voit alors que le système des priorités correspond à l'intuition : la multiplication est plus prioritaire que l'addition. Mais ce n'est pas tout, dans le cas du conflit d'associativité, production et lexème ont la même priorité (celle de `+`). Heureusement `ocamlyacc` autorise aussi de munir les niveaux de priorité d'une associativité, à gauche (`%left`), à droite (`%right`) ou interdite (`%nonassoc`). Ces associativités entrent en jeu dans un conflit *shift*/*reduce* quand les priorités du lexème et de la règle sont identiques. Alors, si l'associativité est gauche, on va réduire, si l'associativité est droite on va shifter, et si l'associativité est interdite l'automate signalera que l'entrée est incorrecte. Ici on veut donc associativité gauche (arbres qui penchent à gauche), et deux niveaux de priorité (`*` et `/`, plus prioritaires que `+` et `-`). On écrit donc, après la définition des lexèmes.

```
/* Des moins prioritaires aux plus prioritaires */
%left ADD SUB
%left MUL DIV
```

Modifions le fichier et recompilons, les conflits disparaissent ou plus exactement `ocamlyacc` ne les signale plus. Mais que se passe-t-il maintenant pour le moins unaire ? Regardons donc dans le nouveau fichier `arith.output`, où est réduite la production $E \to$ `-` E (numéro 6) :

```
state 9
expr1 : expr1 . ADD expr1  (2)
expr1 : expr1 . SUB expr1  (3)
expr1 : expr1 . MUL expr1  (4)
expr1 : expr1 . DIV expr1  (5)
expr1 : SUB expr1 .  (6)

MUL  shift 12
DIV  shift 13
.  reduce 6
```

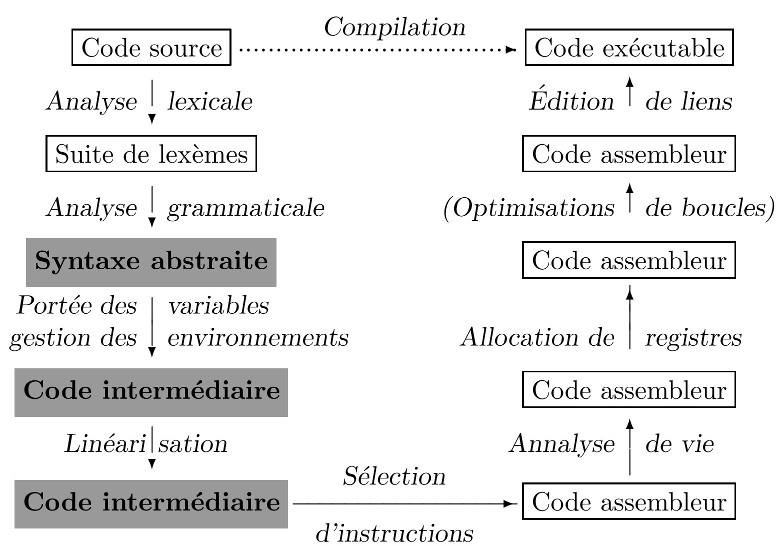
Il y a *shift* pour `*` et `/` et *reduce* pour tous les autres lexèmes (« `.` » indique le comportement par défaut de l'automate). C'est logique compte-tenu des priorités, `-` E •`*` E est actuellement interprété comme `-` (E `*` E). Or, tous ces conflits devraient se résoudre par un *reduce* : `-` E • `op` E est à comprendre comme (`-` E) `op` E. Autrement dit, il faut rendre la production du moins unaire plus prioritaire que les quatre opérateurs. C'est possible en donnant un nom à un niveau de priorité et en forcant la priorité de la production ainsi :

```
%left ADD SUB
%left MUL DIV
%left UMINUS /* left sans importance ici, car pas de lexème de cette priorité */
   ...
| SUB expr1 %prec UMINUS {Binexp (Sub, Int 0, $2)}
```

Une fois ces modifications faites, on peut s'amuser à vérifier que l'automate se comporte correctement dans l'état 9. On voit que les actions sont toutes de réduire : ≪ `.  reduce 6` ≫.

Chapitre 6

Analyse sémantique et code intermédiaire



Un compilateur complet comprend un certain nombre de phases, disons optionnelles, de niveau sémantique (c'est à dire qui s'appliquent au langage source), il s'agit d'abord de la vérification des types (ou de vérifications plus simples du bon usage des noms, souvent effectuées à l'occasion du typage), mais aussi d'optimisations de haut-niveau reposant sur la sémantique du langage et idéalement appliquées à l'arbre de syntaxe abstraite. Je ne détaillerai pas ces phases optionelles, le typage est déjà traité dans le cours langage et programmation de majeure I.

Dans le chemin de la syntaxe concrète au code machine, la phase « sémantique » pourrait *a priori* être toute la traduction de la syntaxe abstraite vers le code machine : un sens est donné au langage à l'aide des moyens d'expression de la machine. Pour des raisons de bonne compréhension et de souplesse on ne produit pas directement du code pour la machine ciblée. On se donne un code dit intermédiaire qui est le code d'une machine idéale et on produit du code pour cette machine.

6.1 Les environnements

Toutes les opérations sémantiques ont besoin de réaliser les règles de résolution des reférences de variables. C'est à dire qu'elles doivent savoir définir une liaison entre une variable et quelque chose et retrouver le quelque chose plus tard au vu du nom de la variable. Il est logique de regrouper l'ensemble des fonctionnalités liées aux environnements dans un module idoine dénommé `Env`, afin de les offrir à toutes les phases sémantiques, mais aussi pour bien structurer notre code.

6.1.1 Réalisation des liaisons

Dans les langages compilés les liaisons des variables sont réalisées selon le principe de la portée lexicale. Prenons un exemple (en Caml) :

```
let x = "coucou" in
x ^ (let x = 1 in string_of_int (x+1)) ^ x)
```

Les occurences des variables dans les expressions (dites *non-liantes*) font référence à la liaison (les occurrences *liantes*) la plus proche en regardant vers le haut (dans l'arbre de syntaxe abstraite).

Du point de vue de la gestion des environnements l'évaluation de l'expression ci-dessus demande de (évaluation de la gauche vers la droite) :

— créer une liaison entre `x` et `"coucou"`, pour évaluer `x ^`,
— créer une liaison entre `x` et `1` pour évaluer `string_of_int (x+1)`,
— retrouver la première liaison pour évaluer `^ x`.

Autrement dit (et c'est un peu plus abstrait) :

— `string_of_int (x+1)` est évalué dans un environnement où `x` vaut `1`.
— `x ^` et `^ x` sont évalués dans un environnement où `x` vaut `"coucou"`.

Notons qu'à l'exécution du code compilé, rien n'oblige à détruire l'espace mémoire réservé à la seconde liaison de `x` en même temps que la destruction de la liaison correspondante, ou à l'allouer en même temps que sa création. La politique ≪ *lexicale* ≫ s'applique aux liaisons uniquement. On peut la réaliser selon principalement deux schémas, d'abord un schéma impératif ou un schéma plus fonctionnel (au sens de langage fonctionnel).

Figure 6.1 – Réalisation des tableaux associatifs.

```
exception Free of string (* en cas de référence à une variable non liée *)

let env = Hashtbl create 17

let get x =
  try Hashtbl.find env x
  with Not_found -> raise (Free x)

let set x v =
  (* récupérer l'ancienne valeur de x *)
  let old_v =
    try Some (get x)          (* x avait une valeur *)
    with Free _ -> None in (* x n'etait pas lié *)
  (* creation d'un liaison de x à v, qui efface la précédente *)
  Hashtbl.replace env x v ;
  (* renvoyer l'ancienne valeur de x *)
  old_v

let restore x old_v = match old_v with
| None   -> Hashtbl.remove env x      (* détruire la liaison de x *)
| Some v -> Hashtbl.replace env x v   (* restaurer la liaison de x *)
```

Un premier schéma impératif utilise les tableaux associatifs On peut les voir comme des tableaux ordinaires dont les indices ne sont pas forcément des entiers consécutifs (ici ce sont des chaînes). La figure 6.1 décrit la réalisation de cette structure de données à l'aide de tables de hachage (module `Hashtbl`). Nous avons d'abord besoin d'une opération pour accéder à une association (fonction `get`), et d'associer un nom de variable à une valeur (fonction `set`). Mais, lorsque `set` range la valeur `v` dans la case `x`, il détruit irrémédiablement ce qui s'y trouvait avant. Je propose donc que `set` renvoie l'ancienne valeur de cette case, si elle existait, ainsi qu'une dernière fonction `restore` pour la remettre dans sa case le temps venu. Pour traiter le cas sans liaison pré-existante (que les tables de hachage révèlent), j'utilise le type `option` du module ≪ *ouvert par défaut* ≫ `Pervasives`.

Voyons donc comment utiliser ce style d'environnements dans un interprète, on aura :

```
let rec eval = function
| Var x           -> get x
| Let (x, ex, e) ->
   let vx = eval ex in
   let old_vx = set x vx in
   let ve = eval e in
   restore x old_vx ;
   ve
| ...
```

C'est un peu compliqué et source d'erreurs idiotes (oublier `restore` par exemple), on souhaiterait dans un esprit plus fonctionnel, passer l'environnement à l'évaluateur et evaluer les expressions en fonction des environnements (c'était l'idée des interprèteurs du chapite 3). On alors besoin d'une fonction `extend` pour créer une liaison et toujours d'une fonction `get`. Le plus simple est alors de ne pas détruire une ancienne liaison par une nouvelle mais de la cacher, l'ancienne liaison existe toujours, mais elle n'est plus accesssible. Une première réalisation à base de listes de couples (module `List`) est vite programmée :

```
(* On pourrait utiliser List.assoc qui fait la même chose *)
let rec get x env = match env with
| []          -> raise (Free x)
| (y,v)::rem ->
  if x=y then v
  else get x rem

let extend x v env = (x,v)::env
```

L'utilisation de tels environnements est bien plus simple (et moins dangereuse) que celle des environnement impératifs :

```
let rec eval env = function
| Var x           -> get env x
| Let (x, ex, e) ->
   let vx = eval env ex in
   eval (extend x vx env) e
| ...
```

Le principal désavantage de cette technique est que les `get` sont assez inefficaces (de l'ordre de la taille des environnements). Heureusement il existe des associations de style fonctionnel plus efficaces (en log de la taille des environnements), réalisées à base d'arbres équilibrés. Elles sont disponibles dans la bibliothèque standard de Caml dans le module `Map`. L'utilisation du module `Map` illustrée par la figure 6.2 est un rien complexe, car il faut appliquer un foncteur, c'est à dire une sorte de fonction des modules dans les modules.

6.1.2 Réalisation des environnements

Dans un programme, tous les noms ne sont pas à comprendre de la même façon et deux noms identiques peuvent faire référence à des entités distinctes selon le contexte de leur utilisation. Les

FIGURE 6.2 – Réalisation fonctionnelle des associations.

```
(* Module des chaînes ordonnées *)
module OrderedString = struct
  type t = string
  let compare s1 s2 = Pervasive.compare s1 s2 (* ordre standard *)
end

(* Application du foncteur, pour créer le module des associations aux chaînes *)
module StringMap = Map.Make OrderedString

let get x env =
  try StringMap.find x env
  with Not_found -> raise (Free x)

let extend x v env = StringMap.add x v env
```

noms se classent par catégories disjointes. Par exemple, en Pascal une fonction et une variable peuvent avoir le même nom (mais pas en Caml, où les fonctions sont des valeurs du langage comme les autres). En général, champs d'enregistrements, variables normales et variable désignant des types appartiennent à des espaces de noms différents.

Tous ces noms correspondent à des liaisons traitées sur le mode lexical ou parfois sur un mode global (fonctions de Pseudo-Pascal) selon des modalités qui changent d'un espace de nom à l'autre. Considérons par exemple le cas de Pseudo-Pascal :

1. Deux catégories de noms, une pour les fonctions, l'autre pour les variables.
2. Les fonctions sont globales, potentiellement mutuellement récursives : on peut y faire référence n'importe où dans le programme.
3. Les variables sont globales ou locales, dans le corps d'une fonction on peut faire référence aux variables locales de cette fonction et aux variables globales.

Nous pouvons maintenant préciser l'interface de notre module des environnements de Pseudo-Pascal (figure 6.3). Notons d'abord que le type `environment` des environnements est paramétré (par `'a` et `'b`). En effet le module `Env` est utile pour toutes les phases sémantiques qui en ont besoin et n'associent pas toujours les mêmes valeurs aux fonctions et aux variables « normales ». Par exemple, un typeur souhaitera associer des types, un interprèteur des valeurs du langage source.

Les deux catégories de noms incitent à proposer deux fonctions d'accès différentes, une pour les variables, une pour les fonctions. La règle de formation des environnement au début du monde est donc de créer toutes les liaisons des fonctions et des variables globales et la règle à appliquer dans les corps de fonctions est donc de mettre à jour les liaisons locales uniquement. Selon le contexte (on souhaite conserver les anciennes liaisons locales ou pas) on utilisera `add_local_vars` ou `change_local_vars`.

Le code qui réalise cette interface ne sera pas commenté. En particulier son type est *abstrait*, son nom (`environment`) apparaît dans l'interface, mais pas sa définition. Disons juste qu'il s'agit d'un enregistrement à trois champs, chaque champ étant une table d'associations de style fonctionnel (definie par ailleurs). Disons aussi que la fonction `find_var`, cherche d'abord parmi les liaisons locales, puis en cas d'échec parmi les liaisons globales, comme le revèle un extrait de l'implémentation `env.ml` :

FIGURE 6.3 – Les environnements de Pseudo-Pascal, interface `env.mli`

```
type ('a, 'b) environment
(* type des environnement qui associe aux variables des valeurs de type 'a et
   aux fonctions des valeurs de type 'b *)

exception Free of string
(* retourne l'identificateur recherché lorsqu'il n'est pas trouvé *)

val create_global : (string * 'a) list -> (string * 'b) list ->
  ('a,'b) environment
(*
   "create_global v d" crée un environement avec les liaisons globales
   v et les définitions d. Sur une telle table, find_var x retournera la
   valeur de la liaison x dans v et find_definition x la valeur de la
   définition x dans d.
*)

val add_local_vars : ('a,'b) environment -> (string * 'a) list ->
  ('a,'b) environment
(* ajoute des liaisons locales à un environement *)

val change_local_vars : ('a,'b) environment -> (string * 'a) list ->
  ('a,'b) environment
(* remplace les liaisons locales *)

val find_var : ('a,'b) environment -> string -> 'a
val find_definition : ('a,'b) environment -> string -> 'b
(*
   "find_var env x" recherche la valeur de x dans les liaisons locales ou
   globales de env.

   "find_definition env x" recherche la valeur de x dans les definitions
   de env.
*)
```

```
type ('a, 'b) environment = {
    definitions : 'b table;
    global_vars : 'a table;
    local_vars : 'a table;
  }

...

let find_var env x =
  try get x env.local_vars
  with Free _ -> get x env.global_vars
```

Quelque soit la sophistication du langage compilé, les environnements sont gérées d'une façon similaire. Par exemple dans le cas de Pascal (avec des fonctions locales), il faudra prévoir d'étendre aussi la partie `definition` des environnements. La plus grande complication prévisible provient de la compilation séparée moderne, nos uniques tables des globaux et des fonctions, disparaissent au profit d'une table des modules, association entre les noms de modules et les tables qui les décrivent. Un environnement ≪ global courant ≫ contient les entitées définies par le module en cours de compilation. Notez que cet environnement ≪ global courant ≫ se construit aussi à partir des tables des modules ≪ ouverts ≫ (construction `open` en Caml).

6.1.3 Les environnements à l'exécution

Plus tard, lors de l'exécution les variables ont disparu en tant que concept. Elles sont essentiellement remplacées par des *cases* dont le contenu est la valeur de la variable. Ces cases sont de deux sortes : des cases de la mémoire ou des registres du processeur.

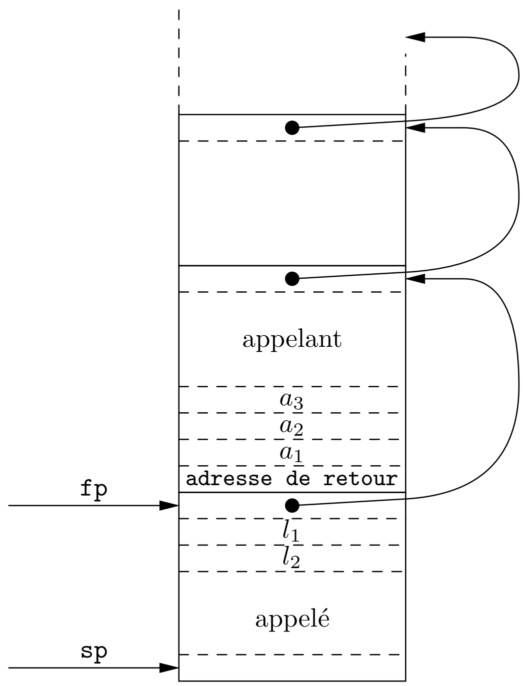
Methode traditionnelle

En oubliant pour le moment les registres et en se restreignant à la compilation de Pseudo-Pascal, l'allocation des cases mémoire devient assez simple :

- Le compilateur connaît l'espace nécessaire aux variables globales : il l'alloue donc statiquement. C'est à dire qu'au lancement du programme le segment de données statique sera d'une certaine taille, précisée dans le fichier assembleur.
- Les cases mémoire attribuées aux variables locales des fonctions sont allouées dans la pile par le code des fonctions. En effet la récursion commande que chaque *appel* de fonction possède ses propres variables distinctes de celles des autres appels de la même fonction. Le compilateur (qui connaît la taille nécessaire aux variables locales) peut donc produire du code qui alloue l'espace en pile au début de l'exécution du corps des fonctions et le rend à la fin (diminuer et augmenter le pointeur de pile, la pile croît vers les adresses mémoire ≪ *basses* ≫). Cette zone de pile qui appartient en propre à chaque appel de fonction s'appelle un *frame* (bloc d'activation).
- Les cases mémoire attribuées aux paramètres formels (et aux éventuelles valeurs à rendre) sont un peu spéciales, elles sont communes à l'appelant (qui y range les paramètres effectifs) et à l'appelé (qui lit les paramètres formels). Mais on conçoit qu'en organisant un peu la pile, les frames de l'appelant et de l'appelé puisssent avoir une partie commune. L'organisation la plus traditionelle introduit un registre supplémentaire, dit *frame-pointer* et noté `fp`. Le frame-pointeur désigne le début du frame d'un appel tandis que le pointeur de pile (noté `sp`) en désigne la fin. Une technique traditionnelle d'appel de fonction peut alors être la suivante :

  1. L'appelant empile les paramètres effectifs (et une place pour ranger la valeur de retour). Le pointeur de pile désigne donc la case du dernier argument empilé.
  2. L'appelant exécute une instruction de saut ad-hoc vers le début du code de l'appelant. Cette instruction de saut empile également l'adresse de retour, c'est à dire l'adresse de

Figure 6.4 – Organisation traditionelle de la pile



l'instruction qui suit l'instruction de saut. Le pointeur de pile désigne donc maintenant la case de pile qui contient l'adresse de retour.

3. L'appelé empile le registre `fp`, puis copie le contenu de `sp` dans `fp`.
4. L'appelé alloue l'espace nécessaire aux variables locales (en diminuant `sp`).
5. L'appelé s'exécute. À cette occasion il peut empiler et dépiler à sa guise (notamment pour appeler d'autres fonctions), à condition de rendre `sp` dans l'état où il l'a trouvé.
6. À la fin de cette exécution, l'appelé rend l'espace de pile des variables locales (et range la valeur à rendre à sa place en pile).
7. Il dépile la valeur du frame-pointer de l'appelé, cette valeur est remise dans `fp` qui retrouve donc sa valeur de l'étape 1.
8. Puis il dépile l'adresse de retour et retourne à l'appelant en sautant à cette adresse (souvent par une instruction ad-hoc qui groupe ces deux opérations).
9. L'appelant dépile les arguments par lui empilés au début : le registre `sp` retrouve sa valeur de l'étape 1.

Dans les détails, cette organisation peut varier un peu : l'appelé peut dépiler les arguments, le jeu d'instruction de la machine peut fournir des instructions qui regroupent les étapes 3 et 4, et les étapes 6 et 7, etc. Mais un principe fondamental demeure : les variables locales (et les paramètres formels) sont repérés par rapport à `fp`, les cases mémoires nécessaires à l'exécution de la fonction sont simplement empilées et dépilées. On voit bien dans la figure 6.4 que la fonction en cours d'exécution (l'appelé), trouve son argument a_i à l'adresse $\texttt{fp} + w * (1 + i)$ et sa variable locale l_j à l'adresse $\texttt{fp} - w * j$ (w est la taille naturelle du mot mémoire). À un instant donné, le frame s'étend donc toujours de `fp`, qui ne bouge pas, à `sp` qui peut bouger.

En fait, le compilateur peut la plupart du temps connaître la taille maximum de pile occupée par le frame d'une fonction. Il suffit *a priori*, si le langage n'autorise pas d'allocation arbitraire en pile (cf. `alloca` de C), de regarder le corps de la fonction. Le code d'une fonction peut donc allouer la totalité de son frame dès le départ de ne le rendre qu'avant de revenir. Dès lors, `fp` est inutile (il vaudra toujours `sp` plus la taille du frame) et c'est ce que nous allons faire dans ce cours. Mais cette organisation de la pile avec deux registres est toujours utilisé :

1. Parce qu'elle est obligatoire pour compiler C (même si on peut se débrouiller pour la limiter aux fonctions qui en ont besoin).
2. Parce que la taille des frames n'est facilement connue que tout à la fin de la compilation et qu'il est bien pratique d'empiler sans se poser de question. Le surcoût à l'exécution strictement attribuable à l'existence de `fp` n'est pas énorme et une des règles non-écrites de la compilation est de mettre en balance la complexité du compilateur et l'effet obtenu au final.
3. La convention de comprendre variables locales et paramètres comme des décalages fixes par rapport à `fp` facilite l'interaction avec un debugger. En outre, le debugger retrouve facilement l'enchaînement des appels en suivant les frame-pointers, comme il est apparent sur la figure 6.4. Il peut alors présenter les appels en cours, voire simuler des retours de fonction.

Méthode moderne

La recherche de l'efficacité conduit à essayer d'attribuer le plus possible des registres aux variables. Prenons quelques exemples :

1. On peut attribuer un registre à une variable globale utilisée très souvent dans le programme. le plus simple est alors de réserver ce registre, il ne peut servir à rien d'autre.
2. Les paramètres formels d'une fonction qui possède peu d'arguments et qui n'appelle pas d'autres fonctions peuvent être mis en registres. C'est un cas fréquent en pratique.
3. Le passage des paramètres (et de la valeur de retour) de toutes les fonctions s'effectue en registres. Ces registres ne sont mis en pile que si nécessaire (appel d'une autre fonction, épuisement des registres disponibles).

Pour bien exploiter les registres un compilateur a besoin d'informations sur l'usage des variables. Une simplification considérable est de limiter les analyses nécessaires au corps des fonctions, analysées indépendamment les unes des autres. C'est plus simple, moins coûteux, et les résultats sont déjà très bons. Notons aussi que l'interaction avec des fonctions compilées par ailleurs (et surtout avec celles qui ont été compilées par un autre compilateur) commande d'adopter des conventions *fixées* au sujet des variables globales et des paramètres. Ces conventions anullent la liberté de mettre variables globales et paramètres formels dans des registres arbitraires. Dans le cas RISC, les conventions d'appel commandent le plus souvent de mettre les paramètres dans des registres convenus, ce qui suffit pour déjà bien profiter des nombreux registres du processeur.

Bref, les environnements sont bien gérés en allouant un frame de pile à chaque appel, mais les décisions de mettre telle variable (ou paramètre) en pile ou en registre sont prises en aval de la génération de code intermédiaire. Nous allons voir comment dans la suite du cours.

6.2 Code intermédiaire

6.2.1 Le code intermédiaire, pourquoi ?

Le code intermédiaire constitue d'abord une interface claire entre le langage et la machine. Son existence augmente la souplesse des compilateurs qui se divisent alors clairement en deux :

— La partie avant *front-end* traduit le langage source en code intermédiaire.
— La partie arrière *back-end* traduit le code intermédiaire en code assembleur.

Dans l'idéal, on peut combiner divers back-ends et front-ends comme on le souhaite. Ainsi si on dispose de deux front-ends, un pour C et un pour Fortran, et de deux back-ends, un pour Mips et un pour Pentium on dispose de quatre compilateurs complets. En pratique, c'est la définition du code intermédiaire qui limite l'intérêt de ces combinaisons, car il doit être suffisamment expressif pour exprimer toutes les constructions des langages sources, mais aussi suffisamment proche des machines réelle pour que les back-ends ne ressemblent pas à des compilateurs complets et surtout qu'ils produisent du code efficace.

Un effet intéressant du code intermédiaire est l'effet unificateur de sa concision. Diverses constructions (voisines) du langage source s'expriment par la même construction du code intermédiaire (appel de fonction et de procédure par exemple), tandis qu'une même construction du code intermédiaire peut regrouper deux opérations voisines de l'assembleur (addition d'une variable entière et d'une constante, ou calcul de l'adresse d'une case de tableau par exemple). Tout travail effectué sur le code intermédiaire est donc très bénéfique, car il factorise un travail qui devrait autrement s'effectuer sur plusieurs constructions distinctes. Évidemment tout travail reposant directement sur la sémantique du langage source (le typage par exemple, ou une optimisation de ≪ haut-niveau ≫) sera mieux fait en amont, et tout travail reposant beaucoup sur un trait spécifique de la machine ciblée (utilisation d'instructions complexes par exemple) sera mieux fait en aval.

Le point de vue équilibré de la combinaison arbitraire des front-ends et des back-ends concerne *a priori* les industriels qui vendent des compilateurs, leur intérêt est bien entendu de proposer le plus de compilateurs possibles pour un travail minimum. On doit aussi considerer le cas des industriels qui produisent des processeurs, leur intérêt est de proposer beaucoup de langages bien compilés sur leur machine. S'ils sont malins, il construiront à grand prix un back-end très efficace et acceptant un langage intermédiaire plutôt d'assez haut-niveau, pour rentabiliser leur énorme investissement. Ils doivent prendre garde à ne pas proposer un code intermédiaire trop proche de leur machine, et donc lutter contre la tentation de mettre en avant les traits distinctifs de leur nouveau processeur.

Les concepteurs de langages de programmation ont un objectif qui semble opposé. Ils souhaitent compiler leur langage, et pouvoir cibler plusieurs machines (car ils souhaitent diffuser leur langage le plus possible). Bizarrement, s'ils sont malins, leur code intermédiaire sera aussi d'assez haut-niveau, (mais ils auront tendance à l'adapter à leur langage). En effet, la compilation efficace demande un gros travail à cause des capacités d'expression limitées des machines et ce travail ne change pas fondamentalement d'une machine à une autre. Il est donc avantageux de procéder à ce travail par transformation du code intermédiaire et de ne se décider pour une machine donnée que le plus tard possible.

6.2.2 Notre code intermédiaire

Il représente en quelque sorte ce que tous les processeurs (ciblés, soyons modestes) ont en commun, sans nous engager trop à cause de ce qu'ils ont de différent. Le type Caml du code intermédiaire est donné par la figure 6.5.

1. Les branchements sont explicites, vers des étiquettes (*labels*, type `Gen.label`)).
2. Le code est arborescent pour les expressions (type `exp`).
3. Le code est linéaire pour les instructions (type `stm`).
4. Il y a un infinité de registres (les temporaires, type `Gen.temp`). Le contenu de ces registres se retrouve après les appels de fonction.
5. L'adressage de la mémoire est explicité (mais les adresses restent des noms : les étiquettes).
6. L'appel de fonction existe en tant que tel.

Les points 1, 3, 4, et 5 traduisent que nous ciblons un processeur, qui exécute des instructions les unes après les autres (certaines de ces instructions sont des branchements) et qui sait faire la différence entre un registre et la mémoire.

Les points 2, 6 et 4 expriment que nous ne souhaitons pas nous engager dès maintenant, ni sur la traduction des expressions en instructions machine, ni sur la traduction des appels de fonctions (car ils dépendent fortement de la machine ciblée), ni sur le nombre réel de registres et surtout sur leur usage (passage d'arguments, *callee-save*, etc.) Il faut bien comprendre que nous pourrions être plus explicites et adopter une représentation plus proche d'une machine réelle. Mais le code

Figure 6.5 – Le code intermédiaire en Caml (interface `code.mli`)

```
type exp =
    Const of int                       (* Entiers et Booléens *)
  | Name of Gen.label                  (* Adresse mémoire nommée *)
  | Temp of Gen.temp                   (* Lecture d'un temporaire *)
  | Mem of exp                         (* Lecture mémoire *)
  | Bin of binop * exp * exp           (* Opération binaire *)
  | Call of Frame.frame * exp list (* Appel de fonction ou appel système *)

and stm =
  | Label of Gen.label                 (* Étiquette (dans le code) *)
  | Move_temp of Gen.temp * exp        (* Écriture dans un temporaire *)
  | Move_mem of exp * exp              (* Écriture en mémoire *)
  | Seq of stm list                    (* Séquence d'instructions *)
  | Exp of exp                         (* Expression évaluée pour son effet *)
  | Jump of Gen.label                  (* Saut non conditionnel *)
  | Cjump of                           (* Saut conditionnel *)
      relop * exp * exp *
        Gen.label * Gen.label

and relop = Req | Rne | Rle | Rge | Rlt | Rgt
and binop = Uplus | Plus | Minus | Times | Div | Lt | Le | Gt | Ge | Eq | Ne
(* Uplus est l'addition non signée pour les calculs d'adresses *)
type code = stm list
```

produit au final exploiterait mal certaines potentialités offertes par un processeur en particulier et serait inefficace.

Plus précisément, les choix des branchements et du code linéaire pour les instructions ne nous limitent pas outre mesure dans notre choix de machines cibles. En revanche, le nombre infini de registres insensibles aux appels de fonction rend compte que nous ciblons des machines dont nous espérons pouvoir bien exploiter les registres. Par exemple, le compilateur décidera *plus tard* de ranger une variable locale dans la pile, si elle doit survivre à un appel de fonction. Si nous ciblions exclusivement des processeurs avec très peu de registres, une telle complication serait inutile, et nous pourrions dès maintenant décider de mettre toutes les variables locales en pile, mais alors nous serions incapable de produire du code efficace pour les processeurs qui ont beaucoup de registres.

Passons maintenant en revue l'environnement de notre code intermédiaire. Les étiquettes et les temporaires doivent être tous distincts, la figure 6.6 donne un extrait de l'interface du module `Gen` qui fournit un type abstrait des étiquettes et quelques fonctionnalités de base.

Le module `Gen` fournit des fonctionnalités similaires pour les temporaires. La génération de code crée beaucoup de temporaires, mais l'allocation de registres qui vient en aval saura très bien les transformer en registres réels, et les temporaires dont la durée de vie est très courte n'auront pas besoin d'être sauvegardés en pile. Donc il faut se retenir de tenter « d'optimiser » l'usage des temporaires dès maintenant. Au contraire, plus on en crée, plus leur durée de vie sera courte et plus facilement ils pourront partager le même registre plus tard. Le mode de pensée à adopter est à l'opposé de celui du programmeur débutant qui a tendance à réutiliser ses variables. . .

Un autre module, `Frame`, entend d'abord définir la représentation en machine des fonctions, donnée par le type abstrait `frame`, dont on peut noter qu'il est argument de l'expression `Call` (cf. la figure 6.5). Comme cette représentation dépend de la machine ciblée, un certain nombre de

FIGURE 6.6 – Gestion des étiquettes, extrait de l'interface `gen.mli`

```
(*******************)
(*  Les étiquettes  *)
(*******************)

type label

(* Renvoie une nouvelle étiquette *)
val new_label: unit -> label

(* retourne une étiquette avec le nom passé en argument.
   Échoue si une étiquette de ce nom existe déjà *)
val named_label : string -> label

(* idem, mais ajoute un suffixe si le nom existe au lieu d'echouer *)
val prefixed_label : string -> label

(* Pour afficher les étiquettes, dans l'assembleur final par exemple *)
val label_string : label -> string
```

détails qui en dépendent directement sont également fournis par le module `Frame`, comme la taille des mots en octets (4 pour le MIPS) donnée par une variable `word_size`. Ce module sera détaillé en temps utile.

Générer un bon code intermédiaire en une seule passe est assez obscur, on va diviser cette opération en trois passes, d'abord générer du code sans soucis d'efficacité ni d'adaptation à la génération de vraies instructions machines, puis corriger cela dans les passes suivantes de linéarisation/canonisation et d'optimisation du flot de contrôle :

$$\mathrm{SA}\overbrace{\underbrace{\xrightarrow[\text{du code}]{\text{Générer}}\ \mathrm{CI}\ \xrightarrow[\text{Linéariser}]{\text{Canoniser}}\ \mathrm{CI}\ \xrightarrow[\text{le contrôle}]{\text{Optimser}}}_{\substack{\text{Code intermédiaire}\\ \text{général}}}}^{(1)}\mathrm{CI}\overbrace{\underbrace{\xrightarrow[\text{d'instruct.}]{\text{Sélection}}\ \mathrm{CA}\ \xrightarrow[\text{de registres}]{\text{Allocation}}}_{\substack{\text{Code machine}\\ \text{Efficace}}}}^{(2)}\mathrm{CA}$$

Cette division en petites étapes, où une passe introduit des inefficacités corrigées par une autre en aval, est robuste, modulable et elle favorise la généralité (et donc la réutilisabilité du code des modules du compilateurs). Un autre avantage est que les passes optmisantes corrigent les inefficacités présentes dans le source en plus de celles introduites par les passes précédentes. Ces avantages compensent plus que largement la relative lenteur du compilateur, surtout à notre époque. La division du back-end en deux (génération du code intermédiaire, puis du code machine) et d'ailleurs un autre exemple du même principe.

6.3 Génération du code intermediaire

On traduit récursivement les expressions et les instructions de Pseudo-Pascal (module `Pp`), en expressions et instructions du code intermédiaire (module `Code`) Nous notons ces traductions respectivement $[\![\_]\!]^e_\rho$ et $[\![\_]\!]^s_\rho$. On notera que les traductions sont paramétrées par un environnement ρ.

6.3.1 Compilation des constructions de Pseudo-Pascal

La traduction des expressions constantes est triviale, remarquons tout de même que les booléens disparaissent au profit de leur realisation par les entiers machines.

$$[\![\texttt{Int}\ n]\!]^e_\rho = \texttt{Const}\ n \quad [\![\texttt{Bool}\ true]\!]^e_\rho = \texttt{Const}\ 1 \quad [\![\texttt{Bool}\ false]\!]^e_\rho = \texttt{Const}\ 0$$

Opérations binaires et séquences de Pseudo-Pascal se traduisent dans les mêmes constructions présentes dans le code intermédiaire.

$$\begin{array}{rcl} [\![\texttt{Bin}\,(op, e_1, e_2)]\!]^e_\rho & = & \texttt{Bin}\,(op, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho) \\ [\![\texttt{Sequence}\,[s_1; \ldots; s_n]]\!]^s_\rho & = & \texttt{Seq}\,[[\![s_1]\!]^s_\rho; \ldots; [\![s_n]\!]^s_\rho] \end{array}$$

La traduction des constructions de tableaux (lecture, écriture) expose les accès à la mémoire. Notons qu'elle est simple parce que les tableaux de Pseudo-Pascal sont sémantiquement des références, et sont donc réalisés par des adresses mémoire. Le calcul de l'adresse de la case d'indice i fait intervenir la taille (en octets) des valeurs rangées dans le tableau ($t + w * i$, où t est l'adresse du tableau). En Pseudo-Pascal, toutes les valeurs occupent un mot machine, w est donc une constante dépendant seulement de la machine ciblée (en pratique ce sera 4 ou 8). Dans le cas géneral cette taille se calcule à partir des types.

$$\begin{array}{rcl} [\![\texttt{Geti}\,(e_1, e_2)]\!]^e_\rho & = & \texttt{Mem}\,(\texttt{Bin}\,(\texttt{Uplus}\,, [\![e_1]\!]^e_\rho, \texttt{Bin}\,(\texttt{Times}\,, \texttt{Const}\ w, [\![e_2]\!]^e_\rho)))) \\ [\![\texttt{Seti}\,(e_1, e_2, e_3)]\!]^s_\rho & = & \texttt{Move\_mem}\,(\texttt{Bin}\,(\texttt{Uplus}\,, [\![e_1]\!]^e_\rho, \texttt{Bin}\,(\texttt{Times}\,, \texttt{Const}\ w, [\![e_2]\!]^e_\rho)), [\![e_3]\!]^e_\rho) \end{array}$$

On notera l'emploi de l'addition non-signée, en fait aucun processeur raisonable ne produit des résultats différents pour une addition signée ou non signée (c'est une bonne propriété de la représentation des entiers en machine par complément à deux). La différence apparaît en cas de débordement, qui est signalé différemment dans les deux cas. La vaste majorité des langages ignorent cette question de débordement et donc peuvent confondre les deux additions, ils ne peuvent pas faire de même pour les comparaisons qui elles produisent des résultats différents dans les cas signé ou le cas non-signé. Or C, par exemple, autorise la comparaison des adresses. Nous ne pouvons donc pas ignorer cette question de signe et l'addition non-signée nous sert d'exemple. Il faut retenir que le meilleur endroit pour introduire la distinction est le code intermédiaire.

La conditionnelle se traduit nécessairement en plusieurs instructions élémentaires, on triche un peu en utilisant la séquence comme une instruction. Lorsque la condition est un test simple, c'est à dire une comparaison d'entiers ($<$, $\leq$, etc.), il convient de court-circuiter la sémantique d'opérateur à valeur booléene, afin d'utiliser directement les instructions test-and-branch des machines. On notera que le test-and-branch du code intermédiaire spécifie deux adresses, il faut brancher vers la première si la condition est vérifiée et vers la seconde si la condition est invalidée. Cette instruction bizarre laisse une grande liberté de réorganiser le code par la suite.

$$[\![\texttt{If}\,(\texttt{Bin}\,(relop, e_1, e_2), s_t, s_f)]\!]^s_\rho = \quad \texttt{Seq} \left[\begin{array}{l} \texttt{Cjump}\,(relop, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho, l_t, l_f); \\ \texttt{Label}\ l_t; [\![s_t]\!]^s_\rho; \texttt{Jump}\ fi; \\ \texttt{Label}\ l_f; [\![s_f]\!]^s_\rho; \texttt{Jump}\ fi; \\ \texttt{Label}\ fi; \end{array} \right]$$

Les tests complexes se traduisent facilement en considérant le résultat nécessairement booléen de la condition et un test simple (ici e_1 différent de *false*, selon la convention adoptée par la sémantique de C).

$$[\![\texttt{If}\,(e_1, s_t, s_f)]\!]^s_\rho = [\![\texttt{If}\,(\texttt{Bin}\,(\texttt{Ne}\,, e_1, \texttt{Const}\ 0), s_t, s_f)]\!]^s_\rho$$

La traduction de la boucle `while` est similaire.

$$[\![\texttt{While}\,(\texttt{Bin}\,(relop, e_1, e_2), s_l)]\!]^s_\rho = \quad \texttt{Seq} \left[\begin{array}{l} \texttt{Label}\ test; \texttt{Cjump}\,(relop, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho, loop, fi); \\ \texttt{Label}\ loop; [\![s_l]\!]^s_\rho; \texttt{Jump}\ test; \\ \texttt{Label}\ fi; \end{array} \right]$$

6.3.2 Compilation des accès aux variables

Attaquons nous d'abord aux accès aux variables, nous verrons plus tard comment les liaisons sont créées. Absolument *toutes* les variables locales et *tous* les paramètres formels sont rangées dans des temporaires (rapellons que les temporaires représentent à la fois les registres et des cases dans la pile). Les variables globales résideront en mémoire (mais en mémoire statiquement allouée). L'environnement ρ doit donc associer les noms des variables à un temporaire ou à une adresse mémoire. Dans le premier cas, $\rho(x)$ est un temporaire t et on a les traductions évidentes.

$$\begin{array}{rcl}
[\![\texttt{Get}\ x]\!]^e_\rho & = & \texttt{Temp}\ t \\
[\![\texttt{Set}\ (x, e)]\!]^s_\rho & = & \texttt{Move_temp}\ (t, [\![e]\!]^e_\rho)
\end{array}$$

Dans le second cas, $\rho(x)$ est une addresse mémoire a et on a encore les traductions évidentes.

$$\begin{array}{rcl}
[\![\texttt{Get}\ x]\!]^e_\rho & = & \texttt{Mem}\ a \\
[\![\texttt{Set}\ (x, e)]\!]^s_\rho & = & \texttt{Move_mem}\ (a, [\![e]\!]^e_\rho)
\end{array}$$

On note que l'accès en lecture examine le contenu de l'adresse, tandis que l'écriture prend cette adresse en argument. Ainsi une variable à gauche ou à droite d'une affectation n'a pas la même interprétation. Dans les langages impératifs (C, Pascal) cette subtile distinction amène une confusion quand on cherche à comprendre le sens des expressions pouvant se trouver à gauche et à droite d'un signe d'affectation (`=` ou `:=`), c'est à dire principalement d'une variable `x`, d'un accès dans un tableau `t[i]`, ou surtout d'un déréférencement de pointeur (`*p` ou `p^`). Dans le premier cas (à gauche) il faut voir l'expression comme le calcul d'une adresse dont on modifie le contenu, dans le second cas il faut comprendre l'expression comme le le contenu de la même adresse. En Pseudo-Pascal la difficulté apparaît moins clairement, car la syntaxe abstraite distingue les deux utilisations autorisées de l'affectation, et les tableaux sont des références. En conséquence, les expressions sont toujours des contenus, y compris dans $\texttt{Geti}\ (e, \_)$ et $\texttt{Seti}\ (e, \_, \_)$ (qui sont respectivement $e\texttt{[\_]}$ et $e\texttt{[\_] := \_}$).

Le cas de la compilation des appels de fonctions est semblable en esprit à celui des variables, les fonctions de Pseudo-Pascal sont désignées par un nom (qui a sa propre catégorie dans l'environnement). L'environnement lie les noms des fonctions à des structures spécifiques les *frames*, que l'instruction d'appel du code intermédiaire prend justement en argument. Nous avons donc, pour les procédures et les fonctions, en notant F le frame de f.

$$\begin{array}{rcl}
[\![\texttt{Procedure_call}\ (f, [e_1; \ldots\ ; e_n])]\!]^s_\rho & = & \texttt{Exp}\ (\texttt{Call}\ (F, [[\![e_1]\!]^e_\rho; \ldots\ ; [\![e_n]\!]^e_\rho])) \\
[\![\texttt{Function_call}\ (f, [e_1; \ldots\ ; e_n])]\!]^e_\rho & = & \texttt{Call}\ (F, [[\![e_1]\!]^e_\rho; \ldots\ ; [\![e_n]\!]^e_\rho])
\end{array}$$

Le traitement des appels de primitives est le même, à la différence que leur frame n'est pas dans l'environnement ρ, mais défini comme une valeur du module `Frame` (`Frame.frame_write`, etc. voir la section 6.3.4). Si les primitives étaient très nombreuses il faudrait sans doute trouver un autre arrangement utilisant un environnement des primitives. Il faudrait alors faire attention aux redéfinitions des noms des primitives.

Il nous reste à voir comment les temporaires, adresses mémoire et frames sont introduits dans les environnements. Commençons par le cas le plus simple des variables globales. On doit, au début du monde et pour chaque variable globale allouer statiquement un mot de mémoire. (une fois encore, si les valeurs du langage n'occupent pas toutes un mot mémoire l'espace alloué dépend du type de la variable). Ensuite, on doit repérer cette addresse, une technique simple est de la repérer par une étiquette, puis d'associer le nom de la variable globale à l'étiquette. Cette technique ne convient pas aux processeurs RISC que nous ciblons (parce que le chargement d'une adresse arbitraire dans un registre prend de l'ordre de deux instructions machine). On repère donc plutôt une variable globale par rapport à une adresse particulière qui est par exemple celle du début de la zone mémoire des globaux. Cette adresse sera chargée dans un registre gp (désigné par le temporaire `Frame.global_register`) au début de l'exécution. Donc l'adresse de la i-ème variable globale sera :

$$\texttt{Uplus}\ (\texttt{Const}\ (w * (i - 1)), \texttt{Temp}\ gp)$$

6.3.3 Les fonctions, représentation, compilation

En Pseudo-Pascal, les variables locales sont introduites exclusivement au début des fonctions (et des procédures, c'est presque pareil). La création des liaisons correspondantes se comprend mieux en exposant comment les fonctions sont compilées.

Conformément à l'idée de ne pas trop nous engager au sujet des registres et de la mémoire allouée en pile, nous allons associer un temporaire frais à chaque variable locale (*i.e.* un temporaire obtenu par un appel à `Gen.new_temp`).

Mais ce n'est pas tout ! Nous refusons aussi de nous engager sur le l'emplacement des arguments. Ainsi, pour une procédure à m arguments nous créons m nouveaux temporaires, et nous ajoutons aussi les liaisons correspondantes à l'environnement avant de compiler le corps. C'est une phase en *aval* entièrement dépendante de la machine ciblée (la sélection d'instructions) qui produira le code qui va chercher les arguments là ou l'appelant les y a mis pour le mettre dans les temporaires associés aux arguments. (Dans le cas du MIPS l'appelé trouve ses quatre premiers arguments dans les registres `a0` à `a3` et les autres sur la pile, mais le générateur de code intermédiaire ne doit surtout pas le savoir.) Dans le cas d'une fonction (par opposition à une procédure), un temporaire frais est également créé pour correspondre à la variable implicite contenant le résultat de la fonction. La sélection d'instructions produira du code pour transférer le contenu du temporaire associé au résultat là où l'appelant l'attend (pour le Mips, dans le registre `v0`). Ces deux bouts de code produits en aval, se nomment respectivement prologue et épilogue de la fonction, ils seront placés au début et à la fin du code de la fonction.

La type `frame` décrit les fonctions dans le back-end. C'est le point de rendez-vous idéal pour les diverses phases du back-end. Voici enfin la définition de ce type extraite de `frame.ml` :

```
type frame = {
      name : Gen.label;          (* Point d'entrée *)
      return_label : Gen.label;  (* Adresse de l'épilogue *)
      args : Gen.temp list;      (* Temporaire des arguments *)
      result : Gen.temp option;  (* Temporaire du résultat (ou rien) *)
      mutable mysize : int;      (* Taille nécessaire sur la pile *)
    }
```

Les deux premiers champs définissent d'abord l'adresse du code de la fonction (utile pour l'appeler) et de son épilogue (techniquement utile à la comunication dans le back-end). Viennent ensuite les temporaires des arguments et du résultat (absent pour les procédures), utiles nous l'avons vu pour communiquer entre le générateur de code intermédiaire et la selection d'instructions. Enfin le dernier champ donne la taille qu'il faudra consacrer à un appel de fonction sur la pile, cette taille est calculée par plusieures phases du back-end.

Le type `frame` des *frames* (blocs d'activation) est abstrait, impossible de travailler directement dessus, on devra passer par les fonctions du module `Frame`. Il y a plusieurs raisons à cela.

— Ce type est destiné à changer. Il dépend en effet de l'architecture ciblée. Ce n'est pas très apparent dans le cas de Pseudo-Pascal compilé vers les machines RISC, mais cela le devient si le langage compilé est plus compliqué et la classe de machines ciblées plus étendue.

— Ce type est compliqué (il le sera encore plus si le langage compilé est complexe) et il n'est pas vraiment utile de l'exposer. Par exemple, considérons la création d'un frame, nous ne voulons pas à ce moment nous préoccuper du champ `mysize`. Or pour créer les frames, une fonction suffit, si elle prend en arguments le nom de la fonction (pour avoir des étiquettes qui le rappellent), la liste de ses paramètres formels (avec leurs types dans le cas général) et une option (`None` pour les procédures, `Some _` pour les fonctions).

```
val named_frame : string -> Pp.var_list -> Pp.type_expr option -> frame
```

Bref, tout de qui concerne l'organisation intime des frames est circonscrit au module `Frame`, afin de bien délimiter ce qui change si cette organisation change. Et nous connaissons maintenant la représentation d'une fonction à ranger dans l'environnement : un *frame*, crée par un appel à `named_frame`.

Le module `Pp` définit les fonctions comme un type enregistrement :

```
definition = {
    arguments : var_list; result : type_expr option;
    (* arguments et type du résultat *)
    local_vars : var_list;
    (* variables locales *)
    body : instruction list;
    (* corps de la fonction *)
  }
```

La compilation d'une fonction de nom f se passe donc en deux temps :

— Création du frame de f, par un appel à `named_frame` auquel on passe f et les contenus des champs `arguments` et `result` (de la définition des fonctions en syntaxe abstraite). C'est cette fonction `named_frame` qui se charge de créer des temporaires frais a_1, a_2, ..., a_n pour les arguments (et un temporaire r pour l'éventuel résultat), une étiquette fraîche pour le point d'entre etc. et de ranger tout ça dans le frame créé. Toutes ces données seront rendues accessibles par le truchement de fonctions idoines du module `Frame` et exportées dans son interface. Dont voici un extrait pertinent :

```
type frame
(* Le type Frame.frame décrit les fonctions (sous-routines) du code intermédiaire *)

val named_frame : string -> Pp.var_list -> Pp.type_expr option -> frame
  (* Creation du frame d'une fonction/procédure, dont le nom est
     passé en premier argument, les paramètres formels en second argument,
     et le (type du) résultat en dernier argument *)
val frame_name : frame -> label
  (* retourne le point d'entrée de la sous-routine *)
val frame_args : frame -> temp list
  (* retourne la liste des temporaires choisis pour recevoir les arguments *)
val frame_result : frame -> temp option
  (* retourne le temporaire choisi pour retourner le résultat d'un vraie fonction, None pour une pro
val frame_return : frame -> label
  (* retourne l'étiquette choisie pour l'épilogue (marque la fin de
  la sous-routine *)
```

— Création d'un environnement dont la partie locale lie les paramètres formels (les noms des arguments) aux a_i (et éventuellement la variable « normale » f à r), et les variables locales à des temporaires frais ; ensuite, compilation du corps ($s_1; s_2; \ldots s_n$ une liste d'instructions) dans cet environnement.

$$\texttt{Seq}\,[[\![s_1]\!]^s_\rho; \ldots ; [\![s_n]\!]^s_\rho]$$

Insistons sur ce que nous n'exprimons pas encore comment on revient des fonctions, c'est la sélection d'instructions qui s'en occupe en aval. Pour le moment le corps de la fonction f est tout simplement une instruction. On y rentre en sautant à l'étiquette `Frame.frame_name` f, et on en sort par un saut vers l'étiquette `Frame.frame_return` f, mais c'est encore implicite.

Note culturelle En Pascal, le corps d'une fonction f spécifie le résultat à rendre en affectant une variable de nom f. Ce n'est pas gênant puisque les fonctions et les variables « normales » appartiennent à des catégories de noms distinctes. Cette convention semble même assez maligne, mais à mon avis elle expose surtout la technique de compilation dans le langage. C'est mal, la conception du langage est inspirée par l'implémentation plutôt que par la recherche d'une bonne expressivité. On comparera avec le `return` de C et Java, bien plus pratique, et parfaitement réalisable selon le même principe par des branchements vers l'épilogue (étiquette `Frame.frame_return` f).

6.3.4 Les fonctions, cas particulier des primitives

Dans notre compilateur les primitives (`write`, `writeln`, `alloc` etc.) sont distinguées dans le front-end, au sens qu'elles sont représentées par des nœuds particuliers de l'arbre de syntaxe abstraite. Cette distinction se justifie surtout dans le cas des primitives `alloc`, dont le deuxième argument est un type justiciable d'une analyse syntaxique bien particulière, et et de `read`, dont l'unique argument est un nom de variable et pas une expression générale. Mais le code intermédiaire ne fait plus cette distinction, pour lui les appels aux primitives sont des appels de fonctions ordinaires et le primitives seront représentés par des frames comme toutes les fonctions. Ces frames particuliers sont fournis par le module `Frame`. Voici l'extrait significatif du fichier d'interface `frame.mli`.

```
(************************)
(* Frames des primitives *)
(************************)

val write_int : frame
val writeln_int : frame
val read_int : frame
val alloc : frame
```

Dans notre compilateur, le code des primitives est en fait un bout d'assembleur qui sera ajouté plus tard à l'assembleur produit par le compilateur, lors d'une phase ultérieure de la compilation. Dans un compilateur plus realiste que le nôtre, les « primitives » seront bien plus nombreuses et assemblées (ou compilées si elles sont par exemple en C) indépendamemnt d'un programme quelconque. C'est alors le travail de l'éditeur de liens (et non plus du compilateur) d'aller chercher le code des primitives.

Une de nos primitives, `read` pose un petit problème. Il faut maintenant la faire passer par le modèle des fonctions ordinaires du code intermédiaire, alors qu'en Pseudo-Pascal, elle est speciale puisque `read` x signifie lire un entier sur la console et ranger sa valeur dans la variable x. C'est à dire que l'appel à la prmitive `read` se compile grosso-modo comme une combinaison d'affectation et d'appel de fonction. Supposons par exemple que $\rho(x)$ est un temporaire t, on a alors :

$$[\![\mathtt{read}\,(x)]\!]^s_\rho = \mathtt{Move\_temp}\,(t, \mathtt{Call}\,(\mathtt{Frame.frame\_read}, [\,]))$$

6.3.5 Compilation d'un programme complet

FIGURE 6.7 – L'interface `trans.mli` du générateur de code intermédiaire.

```
type 'a procedure = Frame.frame * 'a
type 'a program =
    { number_of_globals : int;
      main : 'a procedure;
      procedures : 'a procedure list
    }
val program : Pp.program -> Code.stm program
```

Tout au long du back-end (et donc désormais), Une procédure est une paire composée d'un frame et de quelque chose (ici d'un instructions de code intermédiaire, type `Code.stm`), et un programme est suffisamment décrit par le nombre de ses variables globales la liste de ses fonction et son point d'entrée (une fonctions aussi). Le type des procédures et des programmes est donc un type paramétré.

Il s'agit donc de transformer un programme de syntaxe abstraite (figure 6.8) en programme de code intermédiaire. La démarche de traduction du programme complet est donc :

Figure 6.8 – Définition des programmes Pseudo-Pascal (extrait de `pp.mli`)

```
type program = {
    global_vars : var_list;
    (* variables globales *)
    definitions : (string * definition) list;
    (* fonctions et procédures globales *)
    main : instruction list;
    (* corps principal du programme *)
  }
```

1. Fabriquer l'environnement initial, qui propose des liaisons :
   — pour les variables globales,
   — et pour les fonctions.
   En effet, la sémantique de Pseudo-Pascal autorise l'appel des fonctions *avant* leur définition dans le source.
2. Créer une fonction particulière dont le corps est celui du programme, cette fonction principale (`main`) n'a ni paramètres, ni variables locales et ne sera appelée qu'une fois par un code de lancement ajouté plus tard.
3. Compiler toutes les fonctions à partir de l'environnement « global » de la première étape.

C'est fini, le code est encore loin d'être exécutable, ni même de ressembler à du code machine (il est arborescent), mais il a été produit simplement. Comme la production de ce code réalise la sémantique du langage, cette simplicité est désirable : le programmeur du compilateur peut se concentrer sur le respect de la sémantique et non pas sur une hypothétique adéquation à la machine, d'ailleurs difficile à estimer à ce niveau.

6.4 Linéarisation, canonisation

S'il est un aspect qui est commun à toutes les machines connues, c'est bien que le code est une liste d'instructions. Or, l'instruction Seq du code intermédiaire lui donne une structure arborescente. Nous souhaitons donc la supprimer pour mettre tout le code à plat, ce qui semble assez facile. Cette exigence peut s'exprimer dans l'interface du module `Canon`, chargé (entre autres) de la linéarisation du code.

```
val program : Code.stm Trans.program -> (Code.stm list) Trans.program
```

Autrement dit, on change un programme (au sens de `Trans`, c'est à dire au sens du back-end) dont les fonctions sont des instructions du code intermédiaire, en un programme dont les fonctions sont des listes d'instructions du code intermédiaire.

Toutefois, nous voulons garder les expressions sous forme d'arbre car la selection d'instructions a besoin de ces arbres pour bien fonctionner. Plus précisément, en anticipant un peu, la sélection parcourt l'arbre d'une expression en produisant le code l'évaluant et rangeant le résultat du calcul dans un temporaire r. Or, à partir d'une expression $\texttt{Bin}(\texttt{Plus}, e_1, e_2)$, la sélection peut *a priori* produire l'un ou l'autre de ces codes :

$$\frac{\begin{array}{cc}\text{Code qui calcule } e_1 \text{ dans } r & \text{Code qui calcule } e_2 \text{ dans } r\\ \text{Code qui calcule } e_2 \text{ dans } r' & \text{Code qui calcule } e_1 \text{ dans } r'\end{array}}{\begin{array}{c}\Downarrow\\ r'' \leftarrow r + r'\end{array}}$$

Parfait, mais considérons cette expression :

$$\texttt{Bin}(\texttt{Plus}, \texttt{Call}(f, e_1), \texttt{Call}(g, (\texttt{Call}(h, e_2))))$$

La sémantique (évaluation de gauche à droite) impose le premier choix (code du premier argument d'abord), car les appels de fonctions peuvent faire des effets de bord (ici modifier des variables globales). La selection devra donc connaître un peu de sémantique et ce serait dommage, car elle serait plus compliquée et difficilement réutilisable dans un autre back-end. On peut aussi décider que l'ordre d'évaluation des arguments n'est pas spécifié dans la sémantique (C, Caml), j'aime bien cette solution, mais je ne peux plus changer la sémantique de Pseudo-Pascal maintenant. De toute façon, il y a un autre problème, bien plus grave, considérons cette expression :

$$\texttt{Call}\,(f, \texttt{Call}\,(g, e_1), \texttt{Call}\,(h, e_2))$$

Supposons que les arguments sont passés en registres, arguments dans a_0, a_1, etc et résultat dans r_0. Une des missions de la selection est justement de ranger les valeurs des paramètres effectifs dans ces registres. Alors (ordre gauche-droite), la sélection simple donnera ce genre de code :

$$\begin{array}{ll} a_0 \leftarrow e_1 & \#\ \text{argument de } g \\ \texttt{call}\ g & \#\ \text{le résultat de } g\ldots \\ a_0 \leftarrow r_0 & \#\ \text{est le premier argument de } f. \\ a_0 \leftarrow e_2 & \#\ \text{argument de } h \\ \ldots & \# \end{array}$$

Ça ne fonctionne pas, le passage de l'argument de h détruit le premier argument de de f en attente dans a_0. On pourrait compliquer la sélection, mais encore une fois ce serait dommage, car il y a une sélection par processeur ciblé et nous ne voulons pas dupliquer nos efforts.

Le plus simple est de mettre le code intermédiaire sous forme *canonique*, ainsi définie.

— Il n'y a pas d'instructions Seq.

— Et surtout : les appels de fonctions ne peuvent apparaître qu'au sommet des expressions. Par exemple, (l'instruction `Move_tmp` est abrégé en `Move`) :

$$\texttt{Seq}\,[\texttt{Move}\,(t_0, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Call}\,(f, e_1), \texttt{Call}\,(g, (\texttt{Call}\,(h, e_2))))); \ldots]$$

devrait être transformée en la suite d'instructions :

$$\begin{array}{l} \texttt{Move}\,(t_1, e_1); \\ \texttt{Move}\,(t_2, \texttt{Call}\,(f, \texttt{Temp}\ t_1)); \\ \texttt{Move}\,(t_3, e_2); \\ \texttt{Move}\,(t_4, \texttt{Call}\,(h, \texttt{Temp}\ t_3)); \\ \texttt{Move}\,(t_5, \texttt{Call}\,(g, \texttt{Temp}\ t_4)); \\ \texttt{Move}\,(t_0, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Temp}\ t_1, \texttt{Temp}\ t_5)); \\ \quad\ldots \end{array}$$

La transformation introduit beaucoup de temporaires, gardons confiance dans suite du back-end pour les mettre dans des registres.

Mettre le code intermédiaire en forme canonique revient donc à effectuer un dernier travail sémantique sur le code intermédiaire. Réaliser ce travail dans une phase séparée simplifie à la fois le générateur de code intermédiaire et la sélection d'instructions. Sur le code canonique, la sélection d'instruction sera libre d'arranger le code des expressions comme elle l'entend, elle se contentera de respecter la contrainte d'ordre évidente exprimée par la liste d'instructions qu'elle reçoit en argument.

On peut exprimer la canonisation (et la linéarisation du code) par un ensemble de règles de réécriture du code intermédaire. Les règles de transformation des expressions se notent $e \longrightarrow s \oplus c$ et se lisent, une expression e se tranforme en un code (une liste d'instructions) canonique s et une expression canonique résiduelle c. Des règles possibles sont données par la figure 6.9. Dans cette figure t désigne un temporaire frais. La règle la plus intéressante est de loin celle des appels de fonction. (La traduction de ces règles en un programme Caml, même truffé de concaténations de listes, est un exercice intéressant.)

Ces règles respectent bien l'ordre d'évaluation des expressions et tous les appels de fonctions sont « remontés ». Mais ces règles en font beaucoup trop : il n'y a plus d'arbres du tout, enfin

FIGURE 6.9 – Canonisation naïve

$$\texttt{Const}\ \_ \longrightarrow\ \oplus\, \texttt{Const}\ \_ \qquad \texttt{Temp}\ \_ \longrightarrow\ \oplus\, \texttt{Temp}\ \_ \qquad \texttt{Name}\ \_ \longrightarrow\ \oplus\, \texttt{Name}\ \_$$

$$\frac{e \longrightarrow s \oplus c}{\texttt{Mem}\ e \longrightarrow s \oplus \texttt{Mem}\ c} \qquad \frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad e_2 \longrightarrow s_2 \oplus c_2}{\texttt{Bin}\,(op, e_1, e_2) \longrightarrow s_1; \texttt{Move}\,(t, c_1); s_2 \oplus \texttt{Bin}\,(op, \texttt{Temp}\ t, c_2)}$$

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad \cdots \qquad e_n \longrightarrow s_n \oplus c_n}{\begin{array}{c}\texttt{Call}\,(f, [e_1; \ldots\ ; e_n]) \longrightarrow \\ s_1; \texttt{Move}\,(t_1, c_1); \ldots\ ; s_n; \texttt{Move}\,(t_n, c_n); \texttt{Move}\,(t_{n+1}, \texttt{Call}\,(f, [\texttt{Temp}\ t_1; \ldots\ ; \texttt{Temp}\ t_n])) \oplus \texttt{Temp}\ t_{n+1}\end{array}}$$

presque plus, les arbre penchent maintenant systématiquement à droite ! Il faut en fait adopter des règles plus fines pour les opérateurs et les fonctions. Considérons maintenant la règle des opérateurs, cette règle introduit un temporaire frais et une instruction `Move`, uniquement parce que nous devons évaluer c_1 avant d'évaluer e_2. On dit en général que deux expressions *commutent* quand l'évaluation de l'une et de l'autre peuvent être effectuées dans n'importe quel ordre, sans perturber la sémantique. Or si nons savons que e_1 et e_2 commutent, nous voudrions bien évaluer c_1 après l'exécution de s_2 c'est à dire adopter cette règle :

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad e_2 \longrightarrow s_2 \oplus c_2 \qquad e_1 \text{ et } e_2 \text{ commutent}}{\texttt{Bin}\,(op, e_1, e_2) \longrightarrow s_1; s_2 \oplus \texttt{Bin}\,(op, c_1, c_2)}$$

Il semble intuitivement clair qu'exécuter s_1 (une partie de e_1) puis s_2 (une partie de e_2), puis évaluer c_1 et c_2 (ce qui reste de e_1 et e_2) dans n'importe quel ordre doit être possible, si e_1 et e_2 peuvent justement être évalués dans ≪ n'importe quel ordre ≫.

Mais penchons nous de plus près sur la correction de cette règle. Les expressions c_1 et c_2 sont ultra-canoniques : ce ne sont jamais des appels de fonction (plus généralement, elle ne font pas d'effets de bord) et elles commutent certainement entre elles. On peut donc envisager sereinement une expression $\texttt{Bin}\,(op, c_1, c_2)$ dans tous les cas. La commutation de e_1 et e_2 se réduit donc maintenant à savoir si on peut évaluer c_1 *après* l'exécution de s_2, au lieu du contraire commandé par la sémantique. Il est facile de fournir des approximations, d'abord triviales (s_2 est vide, ou c_1 est une constante), puis un peu plus raffinées en confrontant les temporaires lus par c_1 aux temporaires écrits par s_2, et en considérant aussi si c_1 lit la mémoire et si s_2 écrit dans la mémoire. La règle devient en tout cas :

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad e_2 \longrightarrow s_2 \oplus c_2 \qquad c_1 \text{ et } s_2 \text{ commutent}}{\texttt{Bin}\,(op, e_1, e_2) \longrightarrow s_1; s_2 \oplus \texttt{Bin}\,(op, c_1, c_2)}$$

Je propose le code de la figure 6.10 pour réaliser la canonisation des expressions. Notons que, dans le cas de Pseudo-Pascal, l'approximation triviale suffit pour ne pas toucher aux arbres qui ne contiennent pas d'appel de fonction, et c'est bien là l'essentiel. Enfin, la règle affinée par la commutation se généralise aux appels de fonction. Ici, il faudra vérifier que l'argument canonique c_i commute avec le code canonique $s_{i+1}; \ldots\ ; s_n$.

La canonisation donne aussi lieu à des règles à appliquer aux instructions, mais elles sont bien moins intéressantes. Elles reviennent à remplacer $s_1; \texttt{Seq}\,[s_2; \ldots s_3]; s_4$ par $s_1; s_2; \ldots\ ; s_3; s_4$ (on note au passage les libertés prises avec les notations des listes...), ainsi qu'à appeler la canonisation des expressions. Pour clarifier un peu voici une fonction `flatten_stm` possible qui supprime les `Seq` de l'instruction passée en argument :

FIGURE 6.10 – Canonisation des expressions

```
(* test de commutation simple *)
let commute s c = match s,c with
| Seq [],_ -> true
| _,(Name _ |Const _) -> true (* vive les or-pats *)
| _,_ -> false

(* Autant profiter de Seq *)
let stm_append s1 s2 = match s1, s2 with
| Seq [],_ -> s2
| _,Seq [] -> s1
| _,_      -> Seq [s1 ; s2]

(* Écrit selon les règles de réécriture *)
let rec canon_exp e = match e with
| (Name _ | Temp _ | Const _) -> Seq [], e
| Mem e ->
    let s,c = canon_exp e in
    s, Mem e
| Bin (op, e1, e2) ->
    let s1,c1 = canon_exp e1 and s2,c2 = canon_exp e2 in
    if commute s2 c1 then
      stm_append s1 s2, Bin (op, c1, c2)
    else
      let t = Gen.new_temp () in
      stm_append s1 (stm_append (Move_tmp (t, c1)) s2), Bin (op, Temp t, c2)
| Call (f,args) ->
    let s, cs = canon_exps args in
    let t = Gen.new_temp () in
    stm_append s (Move_temp (t, Call (f, cs))), Temp t

and canon_exps es = ... (* généralisation du cas « Bin » *)


let rec do_noseq i k = match i with
| Seq is  -> do_noseqs is k
| i       -> i::k

and do_noseqs is k = match is with
| []     -> k
| i::rem -> do_noseq i (do_noseqs rem k)

let flatten_stm i = do_noseq i []
let flatten_stms is = do_noseqs is []
```

On pourrait assez logiquement appeler `flatten_stm` sur le résultat d'une fonction `canon_stm`, de type `stm -> stm`, chargée d'appeler la fonction `canon_exp` de la figure 6.10). On pourrait aussi mélanger supression des instructions `Seq` et canonisation mais ce serait moins clair et guère plus efficace.

6.5 Optimisation du contrôle

Une fois canonisé (et mis à plat), le code intermédiaire comprend encore une instruction étrange : l'instruction de test-and-branch `Cjump`, qui spécifie les deux étiquettes où brancher. En langage machine les instructions test-and-branch spécifient seulement l'etiquette de branchement en cas de condition valide, autrement l'execution du code se poursuit en séquence. Cette contrainte est parfaitement expressible dans le code canonique, il suffit d'imposer que l'étiquette ≪ condition invalide ≫ suive toujours le test-and-branch :

$$\texttt{Cjump}\,(relop, e_1, e_2, l_t, l_f);\\ \texttt{Label}\ l_f$$

Obtenir cette situation peut entraîner de nier le test *relop*, si c'est l_t qui suit l'instruction $Cjump$, ou même d'introduire une nouvelle étiquette et un saut vers l_f, si ni l_t ni l_f ne suivent le $Cjump$.

De fait, avec ce test-and-branch bi-étiquette, nous avons introduit une complication, complication qui sert surtout à autoriser l'analyse et la transformation du code, ou plus précisément du flot de son exécution (*control flow*).

Par définition, un *bloc de base* (*basic block*) est une suite (maximale) d'instructions qui est nécessairement exécutée de son début à sa fin. Lorsque l'on ne s'interesse qu'au flot de l'exécution on peut voir un bloc de base comme une grosse instruction (pensez-y deux minutes). En première approximation les blocs de base commencent par une étiquette et se terminent par un saut (conditionnel ou pas) et ne contiennent ni étiquette ni saut. Une étiquette isolée n'est pas un problème, car on peut moralement la faire précéder d'un saut vers elle-même. La génération de code intermédiaire provoque d'ailleurs fréquemment cette situation dans le cas par exemple de la compilation de la conditionnelle. Pour expliciter les blocs de base, il suffit de parcourir une liste d'instructions et de la découper dès que l'on voit une construction qui marque une limite inférieure de bloc (figure 6.11). Notez la régularité de la représentation : les étiquettes du prologue et de l'épilogue marquent les deux extrêmités de la liste d'instructions découpée en blocs.

Mais un processeur n'exécute pas une suite de blocs de bases. On doit lui présenter une suite d'instructions, que nous pouvons bien voir alors comme une suite de blocs de base mis bout-à-bout. Le code était d'ailleurs bien dans cet état avant de passer par la moulinette `code_to_blocks` de la figure 6.11. Et nous pouvons bien imaginer une fonction réciproque `block_to_code` qui défait le travail (figure 6.12, qui utilise `fold_right` du module `List`). Notez que la première étiquette du premier bloc est enlevée, on suppose donc qu'il s'agit toujours de l'étiquette ajoutée précedemment. En revanche, on laisse en place les sauts vers l'épilogue (on pourra enlever un éventuel saut final vers l'épilogue plus tard).

Pourquoi se fatiguer autant ? La structure en liste des blocs de bases n'exprime en fait rien de particulier. Les blocs sont bien mieux organisés selon un graphe de flot (d'exécution) dont les sommets sont les blocs. Il y a un arc du bloc b_1 au bloc b_2 quand l'exécution peut à la fin de b_1 se poursuivre par l'exécution de b_2. Le graphe de flot représente toutes les exécutions possible d'un bout de code. Cette représentation est idéale pour les optimisations du contrôle. Prenons l'exemple assez simple de deux conditionnelles imbriquées et

$$\texttt{If}\,(\texttt{Bin}\,(op_1, e_1, e_2), s_t, \texttt{If}\,(op_2, e_3, e_4, s_{ft}, s_{ff}))$$

En compilant, selon notre schéma simple, on obtient une belle pagaille d'étiquettes et de sauts :

$$
\begin{array}{l}
\texttt{Label}\ start; \texttt{Cjump}\,(op_1, \_, \_, l_t, l_f);\\
\texttt{Label}\ l_t; \ldots\ ; \texttt{Jump}\ fi_1;\\
\texttt{Label}\ l_f;\\
\quad \texttt{Cjump}\,(op_2, \_, \_, l_{ft}, l_{ff});\\
\quad \texttt{Label}\ l_{ft}; \ldots\ ; \texttt{Jump}\ fi_2\\
\quad \texttt{Label}\ l_{ff}; \ldots\ ; \texttt{Jump}\ fi_2\\
\quad \texttt{Label}\ fi_2\\
\texttt{Jump}\ fi_1;\\
\texttt{Label}\ fi_1;
\end{array}
$$

Figure 6.11 – Fabriquer une liste de blocs de base pour le code d'une fonction f

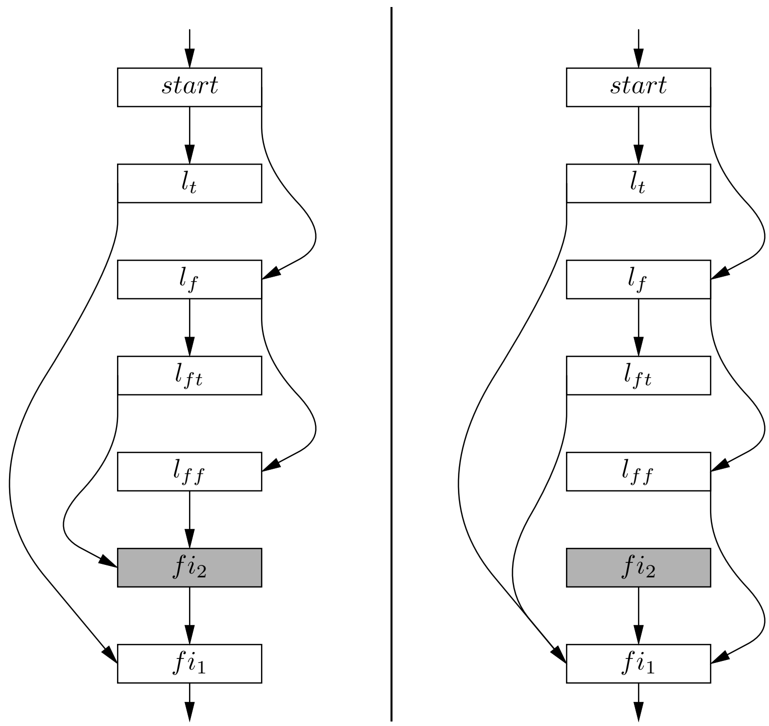
```
type basic_block = {enter:Gen.label ; mutable succ:stm ; body:stm list}

(* Poursuivre la construction d'un bloc commençant par l'étiquette cur_lab
   suivie des instructions cur_ydob (à l'envers) *)
let rec in_block f cur_lab cur_ydob = function
(* Une étiquette termine le bloc *)
| Label lab::rem ->
    let r = in_block f lab [] rem in
    {enter=cur_lab ; succ=Jump lab ; body=List.rev cur_ydob}::r
(* Un saut termine le bloc *)
| (Jump _ | Cjump (_,_,_,_,_)) as stm::rem ->
    let r = start_block f rem in
    {enter = cur_lab ; succ = stm ; body = List.rev cur_ydob}::r
(* Tout autre instruction est à ajouter au bloc *)
| stm::rem ->
    in_block f cur_lab (stm::cur_ydob) rem
 (* Dernier bloc, ajouter un saut vers l'épilogue *)
| []       ->
    [{enter = cur_lab ; succ = Jump (Frame.frame_return f) ;
      body = List.rev cur_ydob}]

(* Commencer un bloc ici *)
and start_block f stms = match stms with
| Label lab::rem -> in_block f lab [] rem
| _              -> assert false (* code mal généré *)

let code_to_blocks f code = in_block f (Frame.frame_name f) [] code
```

Figure 6.12 – Retrouver le code

```
let blocks_to_code blocks = match blocks with
| {body=stms ; succ=stm}::rem ->
    stms @
    stm ::
    List.fold_right
      (fun {enter=lab; body=stms; succ=stm} r ->
        Label lab::stms @ stm :: r)
      rem []
| _ -> assert false
```

La situation s'éclaircit en peu si on regarde le graphe de flot correspondant (à gauche dans la figure 6.13) Dans cette figure, les blocs sont désignés par leurs étiquettes d'entrée et l'ordre de

FIGURE 6.13 – Deux graphes de flot pour les conditionnelles imbriquées



présentation initial des blocs est conservée. Les flèches entrante et sortante disent le début et la fin de n'importe quelle exécution. En machine on peut représenter un graphe de flot par la liste des sommets (une liste de `Gen.label`) et une table de hachage (module `Hashtbl`) qui associe des blocs aux sommets. On retrouve alors facilement les successeurs d'un sommet (figure 6.14).

Une trace est une exécution possible du code, c'est à dire un parcours possible de l'entrée à la sortie du graphe de flot, ici nous avons en tout trois traces : $(start, l_t, fi_1)$, $(start, l_f, l_{ft}, fi_2, fi_1)$ et $(start, l_f, l_{ff}, fi_2, fi_1)$. Mais le bloc fi_2 (grisé) est « vide », c'est à dire qu'il ne contient pas d'instruction et qu'une seule flèche en sort. On peut donc l'enlever de toutes les traces sans rien changer à l'effet de ces traces. Cela revient à court-circuiter le bloc fi_2 dans le graphe de flot. On obtient le graphe de droite de de la figure 6.13, on notera que le bloc fi_2 n'est plus dans aucune trace, il n'est plus atteignable à partir de l'entrée, il constitue du *code mort*. Le court-circuitage des blocs vides revient à enlever des sauts vers les sauts, une optimisation qui est toujours gagnante. Il est assez facile à réaliser, il suffit de parcourir les sommets du graphe de flots, on suit alors chaque arc sortant jusqu'à trouver un bloc non-vide et on remplace l'arc sortant par un arc vers ce bloc non-vide. Le code donné à la figure 6.15 réalise cette opération sur une liste de blocs acompagnée de la table de hachage produite à la figure 6.14 (Notez que ce code boucle si il y a des cycles de blocs vides dans le graphe de flot.)

Les graphes de flot autorisent des optimisations plus complexes. Prenons l'exemple simple du code généré pour la boucle `while` :

$$\begin{array}{l} \texttt{Label}\ test; \texttt{Cjump}\,(relop, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho, loop, fi); \\ \texttt{Label}\ loop; \ldots\ ; \texttt{Jump}\ test; \\ \texttt{Label}\ fi; \end{array}$$

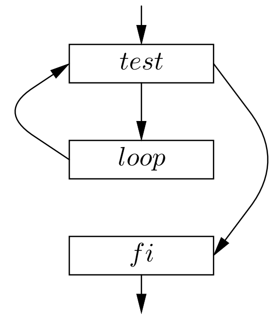
On obtient le graphe de flot :

Figure 6.14 – Réalisation du graphe de flot

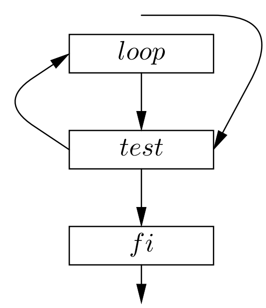
```
let blocks_to_flowgraph blocks =
  let t = Hashtbl.create 17 in
  let labs =
    List.map
      (fun b ->
        let lab = b.enter in
        Hashtbl.add t lab b ;
        lab)
      blocks in
  labs, t

let get_block t lab = Hashtbl.find t lab
let get_succ b = match b.succ with
| Jump lab -> [lab]
| Cjump (_,_,_,lab1,lab2) -> [lab1 ; lab2]
|   _ -> assert false
```

Figure 6.15 – Court-circuitage des blocs vides

```
let rec shorten_lab t lab =
  try
    let b = get_block t lab in
    match b with
    | {body=[]; succ=Jump olab} -> shorten_lab t olab
    | _ -> lab
  with
  | Not_found -> lab

let shorten_block t b = match b.succ with
| Jump lab -> b.succ <- Jump (shorten_lab t lab)
| Cjump (op,e1,e2,lab1,lab2) ->
    b.succ <- Cjump (op, e1, e2, shorten_lab t lab1, shorten_lab t lab2)
| _ -> ()

let shorten_blocks t blocks = List.iter (shorten_block t) blocks
```



Ici nous avons une infinité de traces, commençant par *test*, suivi d'un nombre arbitraire (au point de pouvoir être nul) d'enchaînements de *loop* et de *test*, suivis d'un fi. Nous pouvons parfaitement changer l'ordre de présentation des blocs et obtenir le graphe de flot suivant, équivalent au précédent (il admet les mêmes traces, en fait c'est le même graphe dessiné autrement) :



Retraduisons ensuite la liste de blocs en code :

$$\begin{array}{l}\texttt{Jump}\ \mathit{test}; \texttt{Label}\ \mathit{loop}; \ldots\ ; \texttt{Jump}\ \mathit{test};\\ \texttt{Label}\ \mathit{test}; \texttt{Cjump}\,(\mathit{relop}, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho, \mathit{loop}, fi);\\ \texttt{Label}\ fi;\end{array}$$

Appliquons la simplification évidente de supprimer le saut dans `Jump` *test*; `Label` *test* (mais pas l'étiquette).

$$\begin{array}{l}\texttt{Jump}\ \mathit{test}; \texttt{Label}\ \mathit{loop}; \ldots\ ;\\ \texttt{Label}\ \mathit{test}; \texttt{Cjump}\,(\mathit{relop}, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho, \mathit{loop}, fi);\\ \texttt{Label}\ fi;\end{array}$$

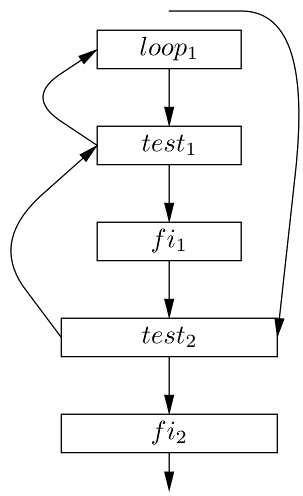
Le code final est presque certainement meilleur (et sans doute pas pire) que le code de départ. En effet, nous avons ameilloré l'exécution d'une infinité de traces en supprimant un saut par passage dans la boucle. Du point de vue des traces, on peut dire que le corps de boucle ainsi présenté *loop*, *test* est *dans sa trace*.

De façon plus générale, en considérant des boucles imbriquées, on conçoit l'importance de présenter les boucles dans leur trace (figure 6.16). Malheureusement c'est assez difficile à réaliser sur un graphe de flot arbitraire et encore plus si le graphe est issu d'un langage qui possède `goto`.

Je présente donc un algorithme plus simple de production d'un arrangement. Cet algorithme construit un ensemble de traces T, en construisant gloutonnement des traces t et un ensemble de blocs atteignables A.

1. Soit b le premier bloc (celui où pointe la flèche entrante). On démarre avec $t = [b]$.
2. Considérer un bloc qui peut suivre le dernier bloc b_t de t. Il y deux cas possibles.
   (a) Il n'y en pas, parce que tous les blocs qui peuvent suivre b_t sont déjà dans les traces de T. Alors, ranger t dans T, choisir un bloc b au hasard parmi ceux de A (si il n'y en a plus c'est fini aller en 3), l'enlever de A et recommencer en 2 en posant $t = [b]$.
   (b) Il y en a un b (il peut y en avoir deux, auquel cas, on choisit et on met l'autre dans A). Alors enlever b de A, le mettre à la fin de t et recommencer en 2.

FIGURE 6.16 – Bon arrangement de la boucle 1 imbriquée dans la boucle 2



3. Produire l'arrangement en mettant bout-à-bout les traces de T.

(Je ne suis vraiment pas malin, il y a un `goto` dans cet algorithme informel). L'algorithme a l'avantage d'éliminer le *code mort* qui n'est pas atteignable à partir de l'entrée (*i.e.* qui n'est dans aucune trace). Cela paraît bien excessif, mais cela arrive (en C, mélange de `break`, `return`, etc.). Par ailleurs le court-circuitage des blocs vides introduit du code mort. Le petit nettoyage qui élimine les blocs vides court-circuités en même temps que tout le code mort a un petit air élégant. L'idée qui est derriere la démarche gloutonne est que si les traces sont les plus longues possibles, alors l'exécution aura tendance à se faire plutôt en séquence et que donc on pourra supprimer des sauts.

Mais en fait je n'aime pas du tout cet algorithme qui saccage l'ordre de présentation des traces. En effet, comme Pseudo-Pascal est structuré (pas de `goto`) on peut dès la génération de code produire un arrangement convenable des traces des boucles. Il suffit de compiler la boucle `while` selon le schéma qui produit le bon arrangement du graphe de flot (que nous avions retrouvé à partir du bon arrangement). Même si on n'utilise pas ce schéma (contestable à cause de problèmes d'alignement des adresses de code), les boucles imbriquées seront correctement placées les unes par rapport aux autres, reflétant la bonne structuration présente dans le code source. Ensuite, on peut se contenter de supprimer les blocs inatteignables de la liste initiale de blocs, *sans en changer l'ordre*. C'est assez facile à faire, on calcule l'ensemble des sommets atteignables en un parcours du graphe de flot en profondeur d'abord, puis on ne retient que les blocs atteignables (figure 6.17, qui emploie la fonction `filter` du module `List`). Ce code suppose donnée une réalisation des ensembles d'étiquettes (`empty` est l'ensemble vide, `mem` teste l'appartenance, `add` ajoute une étiquette à un ensemble).

Une fois les blocs de base réarrangés on peut les retransformer en code à plat, en leur appliquant la fonction `blocks_to_code` de la figure 6.12. Il reste ensuite à se débarrasser des sauts vers une étiquette immédiatement succesive et à garantir que les test-and-branch sont bien suivis de l'étiquette correspondant au cas *false* du test. On réalise ce dernier petit travail par une optimisation « *trou de serrure* » (*peephole*), particulièrement facile à programmer en Caml par un filtrage de la liste d'instructions (figure 6.18, le code est un peu compliqué par la gestion des sauts vers l'épilogue). Il reste ensuite à mettre tout le code ensemble, production des blocs (figure 6.11) et du graphe de flot (figure 6.14), court-circuitage des blocs vides (figure 6.15), suppression des blocs inatteignables (figure 6.17), retour au code (figure 6.12), et finalement nettoyage (figure 6.18). Ce que fait le code de la figure 6.19.

FIGURE 6.17 – Suppression des blocs inatteignables

```
let remove_unreachable t labs =

  let rec dfs r lab =
    if mem lab r then r
    else
      let r = add lab r in
      if Hashtbl.mem t lab then
        let succ = get_succ (get_block t lab) in
        List.fold_left dfs r succ
      else
        r in

  match labs with
  | start::rem ->
      let reach = dfs empty start in
      let labs = start::List.filter (fun lab -> mem lab reach) rem in
      List.map (get_block t) labs
  | _ -> assert false
```

Figure 6.18 – Nettoyage final du code

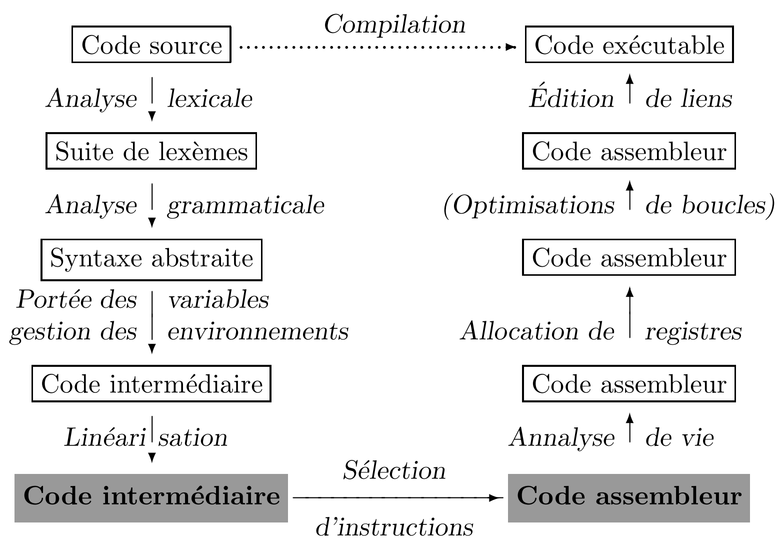
```
let neg = function
  | Req -> Rne | Rne -> Req | Rle -> Rgt | Rge -> Rlt | Rlt -> Rge | Rgt -> Rle
  | _ -> assert false

let rec peephole f = function
(* Cas particulier des sauts vers l'épilogue *)
  | [Cjump (relop, e1, e2, lab1 , lab2)] when lab1 = Frame.frame_return f ->
       [Cjump (neg relop, e1, e2, lab2 , lab1)]
  | [Cjump (relop, e1, e2, lab1 , lab2)] when lab2 = Frame.frame_return f ->
       [Cjump (relop, e1, e2, lab1 , lab2)]
  | [Jump lab] when lab=Frame.frame_return f -> []
(* Garantir que « Label l2 » suit immédiatement *)
  | Cjump (relop, c1, c2, l1, l2) :: t ->
      begin match t with
      | Label l3 :: t3 when l1 = l3 ->
          Cjump (neg relop, c1, c2, l2, l1) ::
          Label l3 :: peephole f t3
      | Label l3 :: t3 when l2 = l3 ->
          Cjump (relop, c1, c2, l1, l2) ::
          Label l3 :: peephole f t3
      | _ -> (* cas embêtant, il faut ajouter une nouvelle étiquette *)
          let l4 = Gen.new_label () in
          Cjump (binop, c1, c2, l1, l4) ::
          Label l4 :: Jump l2 :: peephole f t
      end
(* Saut inutile *)
  | Jump l1 :: Label l2 :: t when l1 = l2 -> Label l2 :: peephole f t
(* Cas normaux *)
  | h :: t -> h :: peephole f t
  | []     -> []
```

Figure 6.19 – Optimisation des traces

```
let opt_trace f c =
  let blocks = code_to_blocks f c in
  let labs,t = blocks_to_flowgraph blocks in
  shorten_blocks t blocks ;
  let blocks = remove_unreachable t labs in
  peephole f (blocks_to_code blocks)
```

Chapitre 7

Sélection des instructions



7.1 Principes

Le source de la passe de sélection des instructions est donc le code intermédiaire canonisé et linéarisé (voir le chapire précédent), sa cible est le langage assembleur de la machine ciblée, c'est à dire dans notre cas celui du MIPS (voir la section 2.2).

Les instructions du MIPS opèrent principalement sur le schéma « trois-adresses », c'est à dire que la plupart des instructions effectuent une opération sur deux registres machines et rangent le résulat dans un troisième. Par exemple l'addition :

add r_1 , r_2 , r_3

Nous noterons ce style d'instruction $r_1 \leftarrow r_2 + r_3$. En fait la deuxième source r_3 peut aussi être un petit entier sur 16 bits. Il d'agit d'une instruction machine différente, même si elle a le même mnémonique `add`, on la note $r_1 \leftarrow r_2 + i_{16}$. Deux autres instructions importantes du MIPS sont l'écriture et la lecture de la mémoire (*store* et *load*).

sw r_1 , i_{16}(r_2) # *charge le contenu de* r_1 *dans la case mémoire d'adresse* $i_{16} + r_2$
lw r_1 , i_{16}(r_2) # *idem pour lire la mémoie*

Nous les noterons $[i_{16} + r_2] \leftarrow r_1$ et $r_1 \leftarrow [i_{16} + r_2]$ (rappelons que l'adresse mémoire écrite ou lue est la somme du petit entier i (seize bits) et du contenu du registre r_2 (trente-deux bits). Le chargement en registre d'une constante ou d'une adresse (une étiquette) s'effectue par les instructions :

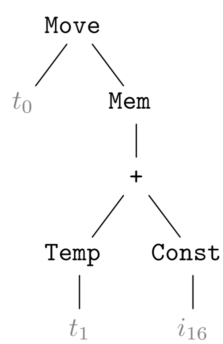
li r_1, i
`la` r_1, ℓ

Notées $r_1 \leftarrow i$ et $r_1 \leftarrow \ell$. Du point de vue strictement machine, ces instructions sont en fait identiques : charger un entier en registre. Le mnémonique `la` indique simplement à l'assembleur de la présence d'une étiquette à resoudre. Toutefois, si l'entier tient sur 16 bits, il s'agit d'une unique instruction machine et de deux sinon. Pour distinguer ces deux cas, donnons nous deux instructions $r_1 \leftarrow i_{16}$ et $r_1 \leftarrow i_{32}$ (pour l'instruction `la` nous ne pouvons rien faire). Enfin, il reste une dernière instruction importante, celle qui transfère le contenu d'un registre r_2 dans un autre r_1

move r_1, r_2

La sélection opérée sur une instruction du code intermédiaire consiste à parcourir l'arbre de cette instruction en émettant la ou les instructions machines qui la réalisent. Il s'agit essentiellement d'un parcours postfixe, on compile les arguments avant d'emettre l'instruction machine qui realise de calcul demandé par un nœud. Chaque bout de code rend son résultat dans un temporaire frais alloué pour l'occasion.

Soit par exemple l'instruction `Move` $(t_0,$ `Mem` (`Bin`(`Plus`, `Temp` t_1, `Const` i_{16}))), présentée sous forme d'arbre avec la simplification de montrer l'addition comme un arbre binaire :



On obtient alors ces trois codes possibles, qui tous sont corrects.

$t_2 \leftarrow i_{16}$ $t_3 \leftarrow t_1 + t_2$ $t_4 \leftarrow [0 + t_3]$ $t_0 \leftarrow t_4$	$t_3 \leftarrow t_1 + i_{16}$ $t_0 \leftarrow [0 + t_3]$	$t_0 \leftarrow [i_{16} + t_1]$

Le premier code prend au pied de la lettre la notion de sélection comme parcours postfixe, le second remarque l'existence d'une instruction d'addition dont la seconde source est un entier et que l'on peut éviter un transfert de registre, le dernier code se rend compte d'entrée de jeu de tout le pouvoir de l'instruction de chargement en mémoire du MIPS.

Une bonne façon de voir est de regrouper les nœuds de l'arbre qui se retrouvent réalisés par une même instruction machine. On appelle traditionellement ces regroupement des *tuiles* (*tiles*). On obtient alors logiquement les recouvrements de l'arbre en quatre, deux ou une tuiles de la figure 7.1.

On notera que les temporaires (et l'entier i_{16}) sont laissés en dehors des tuiles. En effet, ils n'ont pas d'intérêt, seule compte l'instruction sélectionnée. Les malins remarqueront que, pour bien recouvrir tout les nœuds de l'arbre, il nous faudrait une tuile supplémentaire recouvrant le nœud `Temp` et correspondant à aucune instruction. Sur notre exemple nous pouvons dresser deux tableaux des tuiles correspondant à chaque instruction machine (voir figure 7.2). Il y a un tableau pour les expressions du code intermédiaire, et un autre pour ses instructions.

La sélection revient à couvrir les constructions du code intermédiaire à l'aide de tuiles définies à partir du jeu d'instruction de la machine. Pour une instruction donnée, le jeu de tuile associé

FIGURE 7.1 – Trois recouvrements de la même instruction du code intermédiaire.

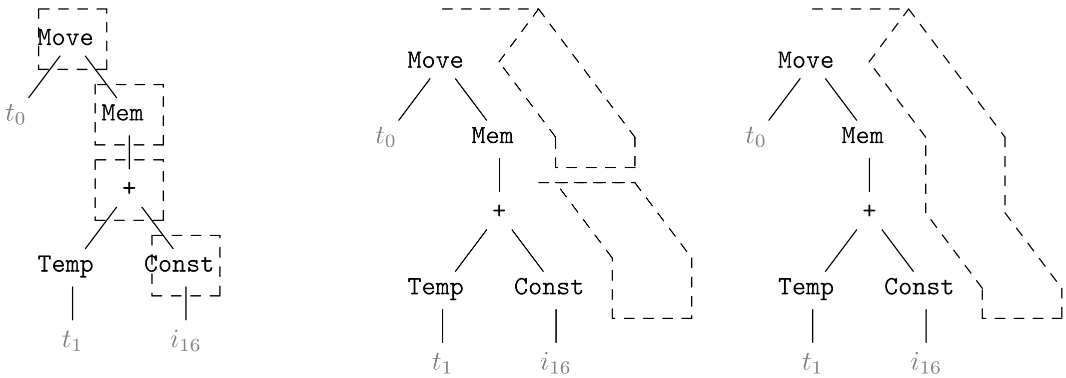
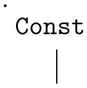
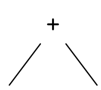
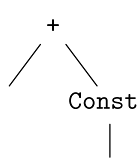
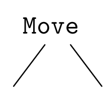
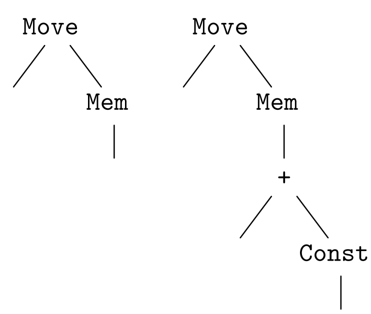
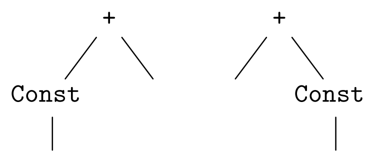


FIGURE 7.2 – Les tuiles utilisées dans notre exemple

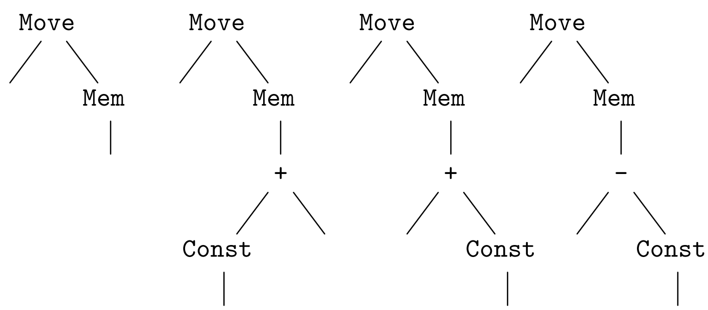
Tuiles à recouvrir les expressions	
$r_1 \leftarrow i_{16}$:	
$r_1 \leftarrow r_2 + r_3$:	
$r_1 \leftarrow r_2 + i_{16}$:	
$r_1 \leftarrow [i_{16} + r_2]$:	

Tuiles à recouvrir les instructions	
$r_1 \leftarrow r_2$:	
$r_1 \leftarrow [i_{16} + r_2]$:	

exprime son « pouvoir couvrant ». Ainsi, en tenant compte de la commutativité de l'addition, le jeu de tuiles « à couvrir les expressions du code intermédiaire » de l'instruction machine $r_1 \leftarrow r_2 + i_{16}$ est constitué de deux tuiles.



Tandis que l'instruction machine $r_1 \leftarrow [i_{16} + r_2]$ possède quatre tuiles à couvrir les instructions du code intermédiaire.



La dernière tuile provient de l'identité $x - i = x + (-i)$. Une tuile du même genre ne semble pas utile dans le cas de l'instruction $r_1 \leftarrow r_2 + i_{16}$ car la tuile de l'instruction de soustraction immédiate $r_1 \leftarrow r_2 - i_{16}$, couvre ce cas. Si les coûts de l'addition et de la soustractions étaient différents, nous pourrions adopter l'une ou l'autre tuile [1].

Une fois les tuiles définies, le jeu consiste donc à en recouvrir les arbres du code intermédiaire. Si on associe à chaque instruction machine (et donc aux tuiles) un coût, on peut distinguer deux classes de recouvrement d'intérêt particulier.

— Les recouvrements optimaux, tels qu'on ne peut pas regrouper deux tuiles adjacentes pour produire une nouvelle tuile de coût inférieur à la somme des deux tuiles fusionnés.

— Les recouvrements optimums, tels que la somme du coût de leurs tuiles est mimimale.

Le coût des instructions machine peut par exemple s'estimer en considérant le nombre de cycles du processeur nécessaires pour les exécuter. Dans le cas du MIPS il suffit alors de compter le nombre d'instructions machines des expansions des instructions de l'assembleur (celles que nous sélectionnons en fait). Cette estimation ne rend bien entendu pas compte exactement du temps d'exécution d'une instruction, qui dépend de nombreux autres facteurs (état des caches, du *pipeline*, ...). Une estimation encore plus simple est d'associer un coût unité à chaque instruction de l'assembleur, il y a alors confusion entre coût, nombre de tuiles et longueur de la séquence d'instructions assembleur générée. Toutes ces estimations sont grossièrement fausses dans le cas de la multiplication et de la division qui coûtent toujours plus qu'une instruction ordinaire. Pour le moment gardons juste ce point en mémoire.

Selon les définitions, un optimum est forcément optimal mais pas le contraire. De fait, l'optimal est un optimun local. La différence de coût entre optimal et optimum n'apparaît que sur des recouvrement plus compliqués que le nôtre et surtout pour un jeu d'instructions plus étendu. Cette différence est de toute façon rarement significative en pratique. Or, il existe un algorithme simple et efficace pour atteindre un optimal, tandis que trouver un optimum demande un algorithme de programmation dynamique plus compliqué et coûteux. Nous nous cantonerons donc à l'optimal, ce qui n'est pas le cas des concepteurs de générateurs de sélecteurs d'instructions. Ces générateurs sont de véritables compilateurs qui transforment des jeux de tuiles (et de coûts) en des programmes réalisant la sélection d'instructions.

1. En fait je mens, les tuiles de l'addition et de la soustraction immédiates ne couvrent pas exactement les mêmes les mêmes arbres... (section 7.4.1)

L'intérêt de ces outils n'est pas évident dans le cas des processeurs RISC.
- L'estimation du coût individual des instructions ne rend pas très bien compte des temps effectifs d'exécution.
- La différence entre optimum et optimal se fait surtout lorsque qu'il y a choix possible entre deux instructions coûteuses différentes, une situation exceptionelle dans le cas d'un processeur RISC.
- La recherche de l'optimal se programme assez facilement en ML.

Ainsi, quand le jeu d'instruction est simple, on rechigne à se donner la peine d'apprendre le fonctionnement d'un outil qui réalise un travail que l'on peut faire soi-même assez facilement. On comparera par exemple avec l'analyse grammaticale : la complexité de l'analyse des grammaires LALR(1) et leur expressivité justifient l'emploi d'un outil genre `yacc`.

En effet, la recherche d'un recouvrement optimal se fait à partir de la racine de l'arbre, on essaie d'abord toutes les tuiles possibles en retenant celle de moindre coût parmi les tuiles qui convienent ; dans le modèle simple de coût, la tuile à choisir est celle qui a le plus nœuds. Ensuite, on opère récursivement sur les sous-arbres qui dépassent de la tuile. Cela semble peut être un peu compliqué, mais l'idée des tuiles précède la venue des langages genre ML, qui possèdent la construction du filtrage (*pattern matching*). Une tuile est réellement une description des nœuds du sommet d'un arbre avec des trous, c'est à dire exactement un motif (*pattern*) au sens de ML, et la recherche de l'optimal se fait donc par une bête fonction récursive qui filtre son argument selon les tuiles.

Par exemple, le programme Caml de la figure 7.3 génère du bon code et n'est guère moins concis qu'une spécification équivalente pour un générateur de sélecteurs. En fait, à condition de présenter les motifs les plus spécifiques en premier, c'est le compilateur Caml qui fait le plus gros travail : la compilation du filtrage On notera aussi que l'emploi de la clause `when` qui résout oportunément le problème des entiers sur 16 bits.

7.2 La sélection en pratique

7.2.1 Les registres

Les registres réels du processeurs sont connus de la sélection, pour émettre certaines instructions qui opèrent sur des registres fixés, mais aussi pour réaliser les conventions d'appel. Ces registres sont représentés par des temporaires, le module `Gen` définissant un tableau de temporaires `registers` à cet effet. Le sélecteur assigne un usage à ces temporaires pré-alloués. Le plus pratique est de ranger chaque temporaire de registre machine dans une variable qui porte son nom conventionel. À l'intention de l'assembleur on se donne aussi un tableau de chaînes des noms conventionnels. (figure 7.4). On regroupe ensuite facilement les registres par catégories. Un fois encore, il s'agit de conventions que nous pouvons changer, afin par exemple de tester l'allocateur de registres sous pression. Toutefois, le registre `ra` doit impérativement être inclus dans la liste des callee-saves, nous verrons pourquoi dans la section sur les fonctions.

7.2.2 Les instructions assembleur

Nous devons sélectionner les instructions mais pas encore choisir les registres. Un type `Ass.instr` explicitant les temporaires et leur usage mais paramétré par les mnémoniques permet cette opération étrange à première vue (figure 7.5). Examinons un peu le principe de l'instruction la plus générale `Oper`. Soit donc la presqu'instruction MIPS `add` t_1`,` t_2`,` t_3. Ce n'est pas tout à fait une instruction de l'assembleur en raison des temporaires qui prennent la place des registres. On la représente par :

`Oper ("add  ^d0, ^s0, ^s1", [`t_2 `;` t_3`], [`t_1`], None)`

La chaîne est l'instruction de l'assembleur, avec les registres arguments de remplacés par `^d`i ou `^s`i. Ces chapeaux étranges désignent l'emplacement dans l'instruction des registres lus et écrits par elle. L'entier i désigne le i plus unième registre de chaque catégorie. La numérotation s'entend par rapport aux listes de temporaires qui suivent, la première liste contient les registres lus (ou sources, d'où le « s ») la seconde les registres écrits (ou destinations, d'où le « d »). Enfin le

FIGURE 7.3 – Un petit sélecteur d'instructions en Caml

```
(* Les entiers relatifs de l'intervalle [−2^{b−1} ... 2^{b−1}[ sont représentables sur b bits *)
let seize_bits i = -(1 lsl 15) <= i && i < (1 lsl 15)

let emit s = Printf.printf "%s\n"

let rec emit_exp e = match e with
| Temp _  -> ()   (* cas de base, pas d'instruction *)
(* Constante entière *)
| Const i -> emit "r1 ← i"
(* l'assembleur distingue r1 ← i16 et r1 ← i32 *)
(* Addition *)
| Bin (Plus, Const i, e2) when seize_bits i ->
    emit_exp e2 ; emit "r1 ← r2 + i16"
| Bin (Plus, e1, Const i) when seize_bits i ->
    emit_exp e1 ;  emit "r1 ← r2 + i16"
| Bin (Plus, e1, e2) ->
   emit_exp e1 ; emit_exp e2 ; emit "r1 ← r2 + r3"
(* Accès mémoire *)
| Mem (Bin (Plus, Const i, e2)) when seize_bits i ->
   emit_exp e2 ;  emit "r1 ← [i16 + r2]"
| Mem (Bin (Plus, e1, Const i)) when seize_bits i ->
   emit_exp e1 ; emit "r1 ← [i16 + r2]"
| Mem (Bin (Sub , e1, Const i)) when seize_bits (-i) ->
   emit_exp e1 ; emit "r1 ← [i16 + r2]" (* i16 est −i *)
| Mem e ->
   emit_exp e ; emit "r1 ← [0 + r2]"
| ...

and emit_stm stm = match stm with
 ...
```

dernier argument indique les sauts effectués par l'instruction, ici il n'y en pas, donc c'est `None`. Plus précisément, `None` signifie que le contrôle passe nécessairement à l'instruction suivante.

Selon cette représentation l'instruction d'addition « immédiate » `add` t_1`,` t_2`, 20` sera donc :

`Oper ("add  ^d0, ^s0, 20",` $[t_2]$`,` $[t_1]$`, None)`

Les sauts s'expriment aussi avec `Oper`. Il faut d'abord bien voir que les étiquettes ont une représentation conforme à nos besoins (type `Gen.label`) et une représentation externe conforme aux exigences lexicales de l'assembleur (genre que des caractères alphanumériques). On passe de la repésentation interne à la représentation externe par la fonction `Frame.string_label`. Soit donc une étiquette ℓ de représentation externe `L123`. Alors, un saut vers cette étiquette s'exprime comme :

`Oper ("b    L123", [], [], Some [`ℓ`])`

Pour l'instruction de saut conditionnel `beq` t_1`,` t_2`, L123` on aura donc :

`Oper ("beq ^s0, ^s1  L123", [`t_1` ; `t_2`], [], Some [`ℓ` ; `ℓ'`])`

Surprise ! Une deuxième étiquette apparaît dans les branchements, c'est celle de la condition invalidée, qui est le dernier argument de l'instruction `Cjump` du code intermédiaire.

Après canonisation, la définition de cette seconde étiquette suit nécessairement en séquence.

FIGURE 7.4 – Les registres du MIPS dans la sélection.

```
let r = Array.sub registers 0 32 (* Le MIPS a 32 registres *)

let zero = r.(0) and at = r.(1) and v0 = r.(2)
... and fp = r.(30) and ra = r.(31)

let name_of_register = [|
  "zero"; "at"; "v0"; "v1"; "a0"; "a1"; "a2"; "a3";
  "t0"; "t1"; "t2"; "t3"; "t4"; "t5"; "t6"; "t7";
  "s0"; "s1"; "s2"; "s3"; "s4"; "s5"; "s6"; "s7";
  "t8"; "t9"; "k0"; "k1"; "gp"; "sp"; "fp"; "ra";
|]

(* Usage standard des registres MIPS *)
let arg_registers          = [a0; a1; a2; a3]
and res_registers          = [v0]
and caller_save_registers = [t0; t1; t2; t3; t4; t5; t6; t7; t8; t9]
and callee_save_registers = [ra ; s0; s1; s2; s3; s4; s5; s6; s7]
(* Une autre convention, 3 registres disponibles
let arg_registers          = [a0]
and res_registers          = [v0]
and caller_save_registers  = []
and callee_save_registers  = [ra]
*)
```

Voilà une occasion de découvrir l'encodage de la pose des étiquettes dans le code assembleur. Soit donc `L321` la représentation externe de l'étiquette ℓ', on aura :

```
Label ("L321:", ℓ')
```

On note le « : » suffixant l'étiquette.

Enfin l'insruction `move` t_1`,` t_2 de transfert de registre à registre est particularisée, en raison de son importance lors de l'allocation de registres, dont une des missions est justement de supprimer les moves si il est arrivé à assigner le même registre machine à t_1 et t_2.

```
Move ("move  ^d0, ^s0", t2, t1)
```

Ne vous laissez pas surprendre par l'échange des arguments, toutes les conventions ont leurs défauts.

7.2.3 Sélection pour les expressions

Les instructions assembleur seront au final donées en argument à une fonction `emit` comme dans le code de la figure 7.3. Mais là où le selecteur théorique affichait l'instruction selectionnée, le selecteur réel doit renvoyer une liste d'instructions machine. Plutôt que de faire renvoyer la liste d'instruction par le selecteur, je préfère conserver une programmation impérative (et oui ça m'arrive) et donc continuer d'appeler la « fonction » `emit` pour son effet de bord. La fonction `emit` doit donc maintenant accumuler les instructions dans une structure de donnée quelconque, que j'appelle une *table* (figure 7.6).

La table est une structure impérative, on définira `emit` ainsi :

Figure 7.5 – Le type des instructions assembleur, interface `ass.mli`

```
type temp = Gen.temp
type label = Gen.label

type instr =
  | Oper of string * temp list * temp list * label list option
(* Oper (mnémonique, sources, destinations, sauts) *)
  | Move of string * temp * temp
  | Label of string * label
```

Figure 7.6 – Interface du module `Table`.

```
type 'a t

val create : 'a -> 'a t  (* Créer une table *)

val emit : 'a t -> 'a -> unit (* Ajouter un element à la fin de la table *)

val trim_to_list : 'a t -> 'a list (* Vider la table dans une liste *)
```

```
let nop = Oper ("nop", [], [], None) (* instruction qui ne fait rien *)

let my_table = Table.create nop      (* le typage de Caml exige cet argument *)

let emit ins = Table.emit my_table ins
```

Il se révèle pratique de regrouper les cas semblables à l'aide de fonctions d'émission spécifiques. Par exemple, pour les opérations arithmétiques on peut écrire :

```
let memo_of_op = function
| Uplus -> "addu" (* addition non signée, pour les calculs d'adresse *)
| Plus  -> "add "
| Minus -> "sub "
  ...
| Eq -> "seq "
| Ne -> "sne " (* opérations booléennes *)

let emit_op3 op d s0 s1 =
  emit (Oper (memo_of_op op^" ^d0, ^s0, ^s1"),[s0 ; s1], [d], None)

let emit_op2 op d s i =
  emit (Oper (memo_of_op op^" ^d0, ^s0, "^string_of_int i),[s], [d], None)

let emit_move d s = emit (Move ("move ^d0, ^s0", s, d))
```

Enfin, la fonction de sélection `emit_exp` doit maintenant rendre le temporaire destination de l'instruction émise. La figure 7.7 décrit un selecteur complet pour les expressions. On peut comparer avec la figure 7.3 qui décrit l'algorithme employé, donc un sélecteur « théorique ». Les temporaires renvoyés sont pour la plupart des temporaires frais, sauf pour les cas particuliers d'un temporaire `Temp` t (t est renvoyé) et de la constante entière 0 (le registre `zero` est renvoyé). On notera l'astuce employée pour exploiter l'éventuel seconde source entière et l'usage de fonc-

Figure 7.7 – Un sélecteur effectif

```
let rec emit_exp e = match e with
(* Temporaire *)
| Temp t -> t
(* Constantes *)
| Const 0 -> zero (* c'est un registre *)
| Const i ->
   let d = new_temp () in
   emit (Oper ("li   ^d0, "^string_of_int i, [], [d], None)) ; d
| Name l ->
   let d = new_temp () in
   emit (Oper ("la  ^d0, " ^label_string l, [], [d], None)) ; d
(* Opérations *)
| Bin ((Plus|Times|Uplus) as op, Const i, e2) -> emit_binop op e2 (Const i)
| Bin (op, e1, e2)                            -> emit_binop op e1 e2
(* Accès mémoire *)
| Mem e   ->
   let d = new_temp () and s,i = emit_addr e in
   emit (Oper ("lw  ^d0, "^string_of_int i^"(^s0)", [s], [d], None)) ; d
(* Les expressions sont canoniques *)
| Call (_,_) -> assert false

and emit_binop op e1 e2 = match e2 with
| Const i when seize_bits i ->
   let s = emit_exp e1 and d = new_temp () in
   emit_op2 op d s i ; d
| _ ->
   let s0 = emit_exp e1 and s1 = emit_exp e2 and d = new_temp () in
   emit_op3 op d s0 s1 ; d

and emit_addr  e = match e with
(* seuls cas intéressants à repérer *)
| Bin (Uplus, Temp r, Const i) when seize_bits i -> r,i
| Bin (Uplus, Const i, Temp r) when seize_bits i -> r,i
(* cas général *)
| _      -> emit_exp e, 0
```

FIGURE 7.8 – Appel de fonction, passage des arguments en registres

```
let emit_jal lab sources dests =
  emit (Oper ("jal "^lab, sources, dest, None))

let emit_call2 f e1 e2 =
  emit_move a0 (emit_exp e1) ; emit_move a1 (emit_exp e2) ;
  let lab =  Gen.label_string (Frame.frame_name f) in
  emit_jal lab [a0; a1]  (ra::v0::args_registers@caller_save_registers)
  let is_fun =
    match Frame.frame_result f with Some _ -> true | None  -> false in
  if is_fun then Some v0 else None
```

tions appelées quand le sommet de la tuile identifie l'instruction **lw** ou le groupe des instructions arithmétiques. On remarquera aussi que le selecteur n'essaie pas d'effectuer les opérations dont les deux arguments sont connus. Cette mission est dévolue à des phases d'optimisation (avec notre représentation des instructions machines ce type d'optimisation n'est possible qu'en amont). Enfin, j'ai un peu triché avec les additions signées et non-signées pour avoir plus de tuiles dans le sélecteur théorique.

La selection sur les instructions du code intermédiaire ne sera pas décrite en détail : c'est exactement la même chose que pour les expressions. La seule différence notable est que les instructions du code intermédiaire n'ont pas de valeur. La fonction `emit_stm` se contentera donc d'émettre le code machine qui exécute son argument et renverra void (`()` de type `unit`).

```
val emit_stm : Code.stm -> unit
```

7.2.4 Les fonctions

Le sélecteur est chargé de réaliser les conventions d'appel de la machine ciblée. Les conventions qui nous intéressent ici sont surtout celles qui déterminent les rapports entre l'usage des registres et les fonctions. Ces conventions pour le MIPS sont décrites à la section 2.4.6. On a principalement.

- Une fonction prend ses quatre premiers arguments dans les registres `a0`, `a1`, `a2` et `a3`.
- Une fonction rend son résultat dans `v0`.
- Les registres `s0` à `s7` sont les *callee-saves*. C'est à dire que leur contenu n'est pas affecté par l'appel de fonction. En pratique cela veut dire qu'avant d'écrire dans un callee-save, une fonction doit sauvegarder son ancien contenu, afin de le remettre dans le callee-save avant de retourner. D'où le nom, *callee* voulant dire « fonction appelée ».
- Les registres `t0` à `t9` sont les *caller-saves*. Le contenu d'un caller-save peut être détruit par un appel de fonction. En pratique cela veut dire que si on range une valeur dans un caller-save et que l'on souhaite encore l'utiliser après un appel de fonction, alors il faudra sauvegarder le contenu du caller-save avant l'appel. D'où le nom, *caller* voulant dire « fonction appelante ».

Appels

Après canonisation, l'appel de fonction n'apparaît plus que dans les instructions du code intermédiaire. Suposons donc un appel de procédure (de fonction) à deux arguments.

$$\texttt{Exp}\,(\texttt{Call}\,(f, e_1, e_2)) \qquad\qquad \texttt{Move\_temp}\,(t, \texttt{Call}\,(f, e_1, e_2))$$

(En fait l'argument de `Call` est une liste d'expressions.)

Nous devons d'abord émettre les instructions qui calculent la valeur de e_1 et la rangent dans `a0`, puis celles qui calculent la valeur de e_2 dans `a1`. Ensuite nous pouvons émettre l'instruction d'appel de sous-routine **jal** (figure 7.8).

Examinons d'un peu plus près l'instruction **jal**. L'adresse de la sous-routine est extraite du frame de la fonction, à l'aide de la fonction idoine du module `Frame`. On constate ensuite que, du point de vue de son dernier argument, l'instruction ne branche pas : elle s'exécute en séquence. Le point de vue est donc de voir l'appel de sous-routine comme une instruction ordinaire. En fait, cette instruction ordinaire remplace les instructions du corps de la fonction f, que nous ne pouvons pas connaître en général puisqu'elles peuvent être émises après l'émission de l'appel. Les sources de cette instruction sont les deux registres arguments `a0` et `a1`, même si ces registres n'apparaissent pas explicitement dans le mnémonique (pas de `^s0`, ni de `^s1`). On découvre ensuite une liste impressionante de destinations. La présence de `ra` ne surprend pas car l'instruction **jal** écrit dedans, la présence de `v0` non plus car si f est une fonction, elle y rangera son résultat, comme le dit `emit_call2` elle même. Mais si f est une procédure ? Et bien f *peut* aussi écrire dans `v0`, si par exemple f appelle une autre fonction. Il en va de même pour tous les registres arguments et tous les registres caller-save[2]. Les callee-saves sont exclus de la liste parce que, même si f écrit dedans, elle doit les rendre dans l'état où elle les a trouvés, ce qui vu de l'extérieur revient à ne pas écrire dedans. Il en va de même pour tous les autres temporaires, car la sémantique des temporaires ordinaires est de ne pas être concernés par l'appel de fonction (section 6.2.2). Il faut bien comprendre que nous sommes justement en train de réaliser cette sémantique. Elle est à voir comme une donnée qui aide à comprendre comment le sélecteur réalise les conventions d'appel.

En fait, les sources et les destinations des instructions sont une information destinées à l'analyse de durée de vie (*liveness*) préalable à l'allocation des registres. De son point de vue, les sources sont des temporaires nécessaires à l'exécution d'une instruction et les destinations sont les temporaires dont cette execution détruit le contenu. Ou plus exactement, comme cette information ne peut être totalement connue, les sources comprennent au moins les temporaires nécessaires et les destinations au moins les temporaires détruits. Adopter ce point de vue dès maintenant aide à comprendre les ≪ lus ≫ et les ≪ ecrits ≫ du type `Ass.instr`.

Les primitives sont un cas particulier, elles sont réalisées par de petites fonctions écrites en assembleur qui effectuent au final les appels système (qui bizarrement ne détruisent aucun registre, même pas `ra`) On connaît donc exactement les registres utiles et détruits de ces fonctions. Il convient de profiter de ce cas particulier. Par exemple, voici le source assembleur de la primitive `alloc` :

```
sw   $a0, 0($fp)
sll  $a0, $a0, 2
addu $v0, $fp, 4
addu $fp, $v0, $a0
j    $ra
```

La sous-routine `alloc` renvoie dans `v0` l'adresse d'une zone de `a0` mots de mémoire allouée dynamiquement. Le registre `fp` est réservé pour servir de pointeur vers la zone de mémoire encore libre. L'examen du code révèle que la sous-routine `alloc` lit le registre `a0` et écrit dans les registres `a0` et `v0`. Notons au passage que `fp` est réservé, c'est à dire que le code produit ne peut pas l'utiliser, il est donc totalement ignoré. Nous confions désormais le soin de determiner les registres détruits par une sous-routine à une fonction `trash` qui prend une fonction (un frame) en argument et renvoie la liste des registres potentiellement détruits par cette fonction. Nous nous livrons ici à un comportement de gagne-petit. Dans un compilateur normal, on peut éviter ce genre de suppositions peu rénumératrices et dangereuses (si on réécrit un bout d'assembleur).

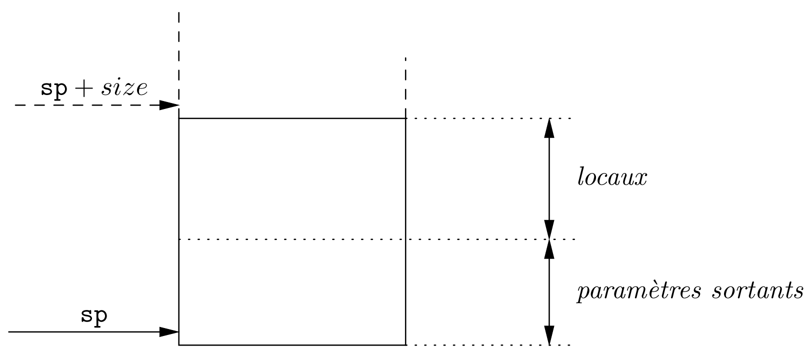
Si la fonction f a plus de quatre arguments l'appelant doit empiler les arguments en excès. Pour simplifier supposons plutôt que seul le premier argument est passé en registre et que f a trois arguments. Rappelons que le frame de l'appelant de f s'étend de son frame-pointeur au pointeur de pile (registre `sp`). Si le frame-pointer reside dans un registre `fp`, l'appelant est libre de modifier `sp`, il se contente donc d'empiler les paramètres effectifs. Mais nous avons reservé `fp` pour un autre usage. Qu'à cela ne tienne, nous devons ranger les deuxième et troisième arguments au sommet de la pile, et nous pouvons donc nous repérer par rapport à `sp`. Mais pour être bien sûrs de ne rien écraser d'important à cette occasion, nous devons signaler à tout le back-end que le sélecteur a

2. Nous ne pouvons en fait pas connaître les registres lus écrits par le corps de f, car à ce stade les instructions lisent et écrivent dans temporaires et non pas dans des registres.

Figure 7.9 – Appel de fonction, passage de deux arguments sur la pile.

```
let emit_store_sp_offset o t =
  emit (Oper ("sw   ^s0,"^string_of_int o^"($sp)", [t], [], None))

let emit_call_1_2 caller_frame f e1 e2 e3 =
  emit_move_to a0 (emit_exp e1) ;
  emit_store_sp_offset 0 (emit_exp e2) ;
  emit_store_sp_offset 4 (emit_exp e3) ;
  Frame.make_space_for_args caller_frame 2 ;
  ...
```

Figure 7.10 – Partage du frame entre fond (*locaux*) et sommet (*paramètres sortants*).



besoin de deux mot au sommet de la pile. On opère en avertissant le frame de l'*appelant* qui doit donc être passé en argument à `emit_call` (et donc aussi à `emit_stm` qui appelle `emit_call`). La fonction `make_space_for_args` enregistre la demande en ajustant la taille du sommet du frame de l'appelant en conséquence (il se souvient de la demande maximale). On note donc au passage que le type `frame` définit une structure de donnée légèrement impérative.

La figure 7.10 résume la situation, à la sélection des appels de fonction nous sommes en train de calculer la taille de la zone des paramètres sortants, zone à allouer au sommet du frame de l'appelant. La taille de la zone des locaux, à allouer au fond du frame, sera calculée par l'allocation de registres. La taille totale du frame, *size*, ne sera donc connue que tout à la fin de la compilation.

Sélection des instructions du corps

La sélection des instructions du corps d'une fonction s'opère normalement, en itérant `emit_stm`. Le gros morceau est l'insertion du prologue au début et de l'épilogue à la fin (voir section 6.3.3). Programmatiquement nous avons donc :

```
let emit_fun f body = (* f est le frame *)
  let saved_callees =  emit_prolog f in
  List.iter (fun i -> emit_stm f i) body ;
  emit_epilog f saved_callees ;
  Table.trim_to_list my_table
```

Prologue et épilogue réalisent notre modèle de gestion des environnements des fonctions. Le prologue commence par l'étiquette du point d'entrée de la fonction f. Il procède succesivement aux opérations suivantes :

1. Allouer le frame en pile (diminuer le pointeur de pile).

2. Transférer tous les registres callee-saves dans des temporaires frais, dits sauvegardes des callee-saves.
3. Copier les arguments de leurs positions définies par les conventions d'appel vers les temporaires définis pour eux lors de la génération du code intermédiaire.

Pour l'épilogue, qui commence par son étiquette (connue de la structure `frame` représentant f) la repose est à l'inverse de la dépose, selon la formule irritante de la Revue technique automobile.

1. Si f n'est pas une procédure, copier le temporaire résultat défini lors de la génération du code intermédiaire dans le registre résultat de la convention d'appel (pendant de 3 du prologue).
2. Transférer les sauvegardes des callee-saves dans les callee-saves (repose de 2 du prologue).
3. Rendre l'espace alloué en augmentant le pointeur de pile (repose de 1 du prologue).
4. Revenir de la fonction par l'instruction idoine (repose de l'instruction d'appel).

Les étapes 3 du prologue et 1 de l'épilogue sont logiques, elles assurent l'indépendance de la génération de code intermédiaire vis à vis de du processeur ciblé. Elle ne posent qu'un léger problème technique dans le cas des arguments en excès passés en pile.

Les étapes 2 du prologue et de l'épilogue sont plus troublantes. Pour les réaliser l'émetteur du prologue renvoie la liste des sauvegardes des callee-saves qui est donnée en argument à l'émetteur de l'épilogue. La fonction f a la responsabilité de rendre les callee-saves dans l'état où elle les a trouvés. Le fonctionnement du processeur ne garantit rien à ce sujet, puisque toute écriture dans un registre est visible de partout. Mais la sémantique des temporaires garantit qu'un temporaire reste insensible aux appels de fonctions que f peut effectuer. Donc, comme le code de f se gardera bien de toucher aux sauvegardes des callee-saves, la combinaison de l'étape 2 du prologue et de l'étape 2 de l'épilogue garantit que f fait face à ses responsabilités. Soit `s0` un callee-save, en pratique on s'attend à l'un où à l'autre des scénarios suivants.

— Si le code de f ne touche pas au registre `s0`, alors son temporaire de sauvegarde sera un registre et idéalement ce registre sera `s0` lui même. Dès lors, les transferts entre `s0` et sa sauvegarde seront éliminés du code.
— Si le code de f touche au registre `s0`, alors son temporaire de sauvegarde sera une case de la zone des locaux du frame de f. La sauvegarde de `s0`, s'effectuera donc en pile.

L'allocateur de registres se dénommerait donc moins publicitairement, allocateur de registres ou de cases de piles.

Reste enfin la dernière étape 4 de l'épilogue. L'emission de l'instruction de retour d'une fonction qui rend son résultat dans `v0` se fait ainsi :

```
emit (Oper "j   $ra", v0::callee_save_registers, [], Some [])
```

(Pour une procédure les temporaires sources de l'instruction de retour ne comprennent pas `v0`)

On remarque d'abord que les sauts indiqués sont `Some []`, ce qui signifie qu'il y a bien saut mais que la destination est inconnue. L'instruction de retour lit effectivement le seul registre `ra` qui est inclus dans la liste des calle-saves. Et il est assez logique de considérer que f doit présenter à sa dernière instruction le registre `ra` dans l'état où elle l'a trouvé.

Mais du point de vue de l'usage qui peut être fait des registres machine, l'instruction de retour remplace toutes les instructions qui peuvent suivre au cours de l'exécution. Or, la suite du code de l'appelant peut selon les conventions d'appel lire `v0` et les (vrais) callee-saves en toute confiance. Plus précisément, l'appelant doit, quand il lira ces registres, y trouver ce qu'il croit qu'ils contiennent, à savoir le resultat de l'appel pour `v0` le cas échéant, et un contenu inchangé pour les callee-saves. Les autres registres (arguments et caller-saves) ne sont pas un problème car l'appelant sait toujours selon ces mêmes conventions que leur contenu n'est plus fiable. (voir les registres « écrits » par l'instruction d'appel).

Enfin, on peut aussi inclure les registres spéciaux (genre `zero`, `sp`, etc.) dans la liste des sources de l'instruction de retour. Je n'en voit pas trop l'intérêt, car ces registres sont réservés c'est à dire que leur usage n'est pas contrôlé selon le mécanisme général des temporaires lus et écrits par les instructions d'appel et de retour de sous-routine. Par exemple, l'usage correct du registre `sp` est garanti par la les diminutions et augmentations symétriques du prologue et de l'épilogue

L'allocation et la libération du frame de f (étapes 1 du prologue et 3 de l'épilogue) posent seulement un problème technique : la taille du frame ne sera connue qu'en aval, après l'allocation de registres. Nous pourions laisser des trous dans le code et aller les remplir ensuite (technique connue sous le nom de *back-patching*). Mais l'assembleur nous autorise une solution moins compliquée à mettre en œuvre. En effet, le programme qui s'appelle assembleur comprend souvent les *constantes symboliques*, c'est à dire les définitions de noms quelconques comme des entiers. Lorsque la taille de f sera connue (par exemple 12 octets), on pourra insérer une définition du nom `f_size` dans le code assembleur. Mais pour l'heure nous nous contentons d'utiliser ce nom (figure 7.11). Dans le cas où `f_size` se révèle finalement nul on peut souhaiter supprimer les instructions d'ajustement du pointeur de pile. Oublions ce détail.

Il reste à examiner le passage d'arguments en pile. Supposons que f prend trois arguments, dont le premier est passé en registre et les deux suivants en pile. Les arguments sont récupérés après l'allocation du frame (étape 1 du prologue), c'est à dire que le pointeur de pile est déjà diminué de `f_size`, taille du frame de f. Ici encore, on peut désigner la position en pile des arguments entrants en s'aidant de la constante symbolique `f_size` (figure 7.12). En supposant un unique vrai callee-save `s0`, le code d'émission du prologue f est donné par la figure 7.13. Bien sûr, dans le sélecteur que vous allez écrire, vous devez écrire une fonction `emit_prolog` générale qui traite le cas de toutes les fonctions, quelque soit leur nombre d'arguments.

7.3 Un exemple simple

Soit la fonction facorielle écrite en Pseudo-Pascal.

```
function fact (n : integer) : integer;
begin
  if n <= 1 then
    fact := 1
  else
    fact := n * fact (n - 1)
end ;
```

Le code intermédiaire généré est le suivant :

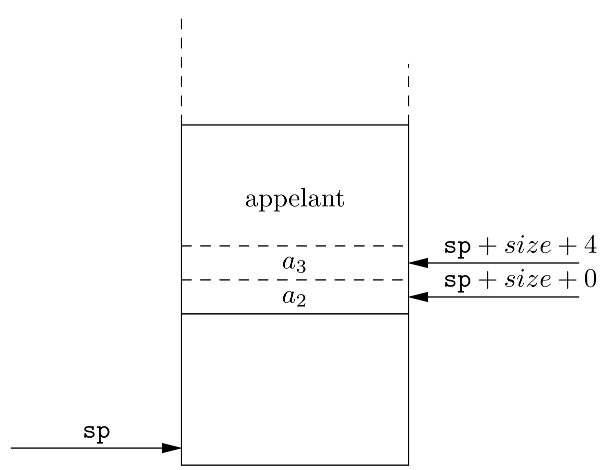
```
function fact
args =  $t105
result = $t104
  Cjump L12 L13 (<= $t105 1)
L13:
  (set $t107 $t105)
  (set $t106 (call fact (- $t105 1)))
  (set $t104 (* $t107 $t106))
  Jump fact_end
L12:
  (set $t104 1)
```

Dans le code assembleur produit par le sélecteur (en se donnant un seul callee-save `s0`), on notera particulièrement l'apparition du prologue et de l'épilogue.

Figure 7.11 – Allocation et libération du frame de f à l'aide d'une constante symbolique.

```
    f_size = 12          # mis là après l' allocation de registres

# Code produit par le sélecteur
f: # prologue de f
    subu $sp, $sp, f_size
    ...

f_end: # épilogue de f
    ...
    addu $sp, $sp, f_fize
    j    $ra
```

Figure 7.12 – Passage des arguments en pile vu de l'appelé.



```
fact:
    subu $sp, $sp, fact_f
    move $111, $ra
    move $112, $s0
    move $105, $a0
    li   $113, 1
    ble  $105, $113, L12
L13:
    move $107, $105
    sub $114, $105, 1
    move $a0, $114
    jal fact
    move $106, $v0
    mul $115, $107, $106
    move $104, $115
    b fact_end
L12:
    li  $104, 1
fact_end:
    move $v0, $104
    move $ra, $111
    move $s0, $112
    addu $sp, $sp, fact_f
    j   $ra
```

Il y a de nombreux temporaires et transferts entre temporaires qui sembleraient gréver l'efficacité finale du code. Mais le compilateur peut très bien au final produire ce code :

```
    fact_f=8
fact:
    subu $sp, $sp, fact_f
    sw
$ra, 0($sp)  # store $111
    sw
$s0, 4($sp)  # store $112
    li  $v0, 1
    ble $a0, $v0, L12
L13:
    move $s0, $a0
    sub $a0, $a0, 1
    jal fact
    mul $v0, $s0, $v0
    b fact_end
L12:
    li  $v0, 1
fact_end:
    lw  $ra, 0($sp)  # load
$111
    lw  $s0, 4($sp)  # load
$112
    addu $sp, $sp, fact_f
    j  $ra
```

Dans ce code final, les temporaires `$111` et `$112` se retrouvent en pile. On note que les temporaires argument (`$105`) et résultat (`$104`) se retrouvent respectivement dans les registres argument (`a0`) et résultat (`v0`), ainsi que l'allocation du registre `s0` au temporaire `$107` qui est à la racine de la bonne allocation des registres.

7.4 Quelques détails

7.4.1 Sur les opérations immédiates et la multiplication

Nous avons pris soin de selectionner les instructions « immédiates » (c'est à dire celles qui prennent une seconde source qui est un entier sur 16 bits) dès que c'était possible et ceci pour toutes les instructions qui opèrent sur un mode trois adressses. Or, une étude attentive du jeu d'instructions du processeur MIPS (et non de toutes les instructions acceptées par l'assembleur) révèle que parmi les instructions qui nous intéressent, les seules qui peuvent prendre une deuxième source immédiate sont en fait **add**, **addu**, **slt** (opération <), **sll** et **sra** (décalages). Mais, l'assembleur sait traiter toutes les instructions immédiates que nous sélectionnons, dans le cas de la soustraction il saura même remplacer une soustraction immédiate inexistante en machine par l'addition immédiate équivalente (sauf si la constante est $-(2^{15})$ ...). Dans le cas général l'assembleur remplacera l'instruction immédiate par un chargement préalable dans l'un de ses deux registres réservés et l'instruction trois-adresses correspondante. Nous pouvons donc sans nous fatiguer sélectionner toutes ces instructions immédiates, nous aurions même pu nous éviter de verifier que l'argument entier tient sur 16 bits, Mais tous les assembleurs ne sont pas aussi sympathiques.

Toutefois il est un cas où nous devons travailler nous mêmes, il s'agit de la multiplication et de la division. Ces instructions prennent plus de temps à exécuter que les autres et il convient, quand le deuxième argument est constant de tenter de les remplacer par une ou plusiseures instructions « normales ». C'est particulièrement important dans le cas des multiplications par la taille naturelle du mot introduites à foison par les accès dans les tableaux. Or, une multiplication par 2^b est équivalente à un décalage à gauche de b positions (**sll**), tandis qu'une division (signée) par 2^b est équivalente à un décalage dit arithmétique à droite de b positions (**sra**). Dans ce dernier type de décalage, le bit de signe (le plus à droite) est répliqué b fois afin de combler le trou laissé par le décalage. Bref, on peut, grâce au repérage effectué par `emit_exp`, traiter quelques cas particuliers fréquents et significatifs dans `emit_binop` (cf. figure 7.7). Idéalement, une transformation aussi simple, revenant à remplacer une opération coûteuse (une multiplication) par une autre certainement moins coûteuse (un décalage), est valable pour tous les processeurs, et on aurait du y procéder en amont. Des réductions plus ambitieuses sont possibles, c'est à dire que l'on peut transformer une multiplication par une constante en k instructions. Mais on retrouve alors un problème de dépendance au processeur, notamment pour connaître la limite supérieure de k en fonction des coût des instructions. On notera aussi que ce type de transformation profitera surtout aux programmes qui font beaucoup de multiplications, et non pas à tous les programmes en général. On ne s'attaquera donc à cette optimisation que si le besoin s'en fait sentir.

7.4.2 Quelques problèmes posés par le Pentium

Les processeurs Intel se distinguent par leur jeu d'instruction moins régulier que celui du MIPS (d'où plus de tuiles) et aussi par leurs opérations « deux-adresses ».

Ainsi, une addition s'écrit :

addl r_2`,` r_1 `#` $r_1 \leftarrow r_1 + r_2$

(Attention à l'inversion des arguments !). Ce n'est pas réellement une difficulté, il suffit de continuer à considérer les opération comme « trois adresses » (genre $t_1 \leftarrow t_2 + t_3$) et d'emettre le code :

movl t_2`,` t_1 `#` $t_1 \leftarrow t_2$
addl t_3`,` t_1 `#` $t_1 \leftarrow t_1 + t_3$

On compte sur l'allocateur de registres pour attribuer si possible le même registre machine aux temporaires t_1 et t_2.

Dans les anciens processeurs Intel, l'instruction de multiplication est très contrainte. Le multiplicateur est nécessairement dans `%eax`, et le résultat est sur 64 bits, poids faibles dans `%eax` et poids forts dans `%edx`. Pour réaliser une multiplications trois-adresses $t_1 \leftarrow t_2 * t_3$, on émettra :

movl t_2`, %eax #` `%eax` $\leftarrow t_2$
imull t_3 `#` `%eax` $\leftarrow$ `%eax` $* t_3$`, %edx` $\leftarrow ?$
movl `%eax,` t_1 `#` $t_1 \leftarrow$ `%eax`

Et surtout ou oubliera pas `%edx` dans la liste des temporaires « écrits » par l'instruction `imull`. Ici encore on peut espérer une allocation des temporaires t_1 et t_2 dans le registre `%eax`, mais c'est faire preuve d'optimisme. Dans les processeurs Intel plus modernes, on a une multiplication deux adresses, mais le problème demeure pour la division.

La majorité des opérations peut aussi opérer en mémoire. Ainsi on peut écrire :

addl t_2`, 4(`t_1`) #` $[4 + t_1] \leftarrow [4 + t_1] + t_2$

L'effet est d'additionner t_2 au contenu de la case mémoire adressée par $4 + t_1$. Il n'y a pas de problème majeur pour identifier la tuile associée si le code intermédiaire exhibe ce motif et de toute façon ce sera un cas bien rare (incrément d'une case de tableau). Mais si les accès mémoire résultent de la mise en pile d'un temporaire en pile (mise en pile décidée après la sélection), on aura plutôt ce style de code :

movl `4(%esp),` t `#` $t \leftarrow [4 + \texttt{\%esp}]$
addl t_2`,` t `#` $t \leftarrow t + t_2$
movl t`, 4(%esp) #` $[4 + \texttt{\%esp}] \leftarrow t$

Ce n'est en fait pas bien grave car ce second code a un temps d'exécution théorique identique au premier code en une instruction. La pénalitée payée est un code un peu moins compact et un temporaire en plus (t), qui compte tenu de sa durée de vie très courte sera alloué en registre.

Enfin on peut souhaiter exploiter les instructions d'empilage et de dépilage qui on un effet de bord (décrément ou incrément de la taille du mot) sur le registre `%esp`.

pushl t `#` $\texttt{\%esp} \leftarrow \texttt{\%esp} - 4$ `;` $[0 + \texttt{\%esp}] \leftarrow t$
popl t `#` $t \leftarrow [0 + \texttt{\%esp}]$ `;` $\texttt{\%esp} \leftarrow \texttt{\%esp} + 4$

On peut, pour un léger gain en vitesse, employer ces instructions de façon ad-hoc. Je pense surtout à l'empilage des arguments en excès lors des appels de fonction. Cela n'a en fait aucun intérêt dans le cas de notre compilateur qui regroupe les mouvements du pointeur de pile dans le prologue et l'épilogue. C'est un peu plus pertinent dans le cas d'un frame-pointeur en registre.

FIGURE 7.13 – Émission du prologue de f qui prend ses deux derniers arguments en pile

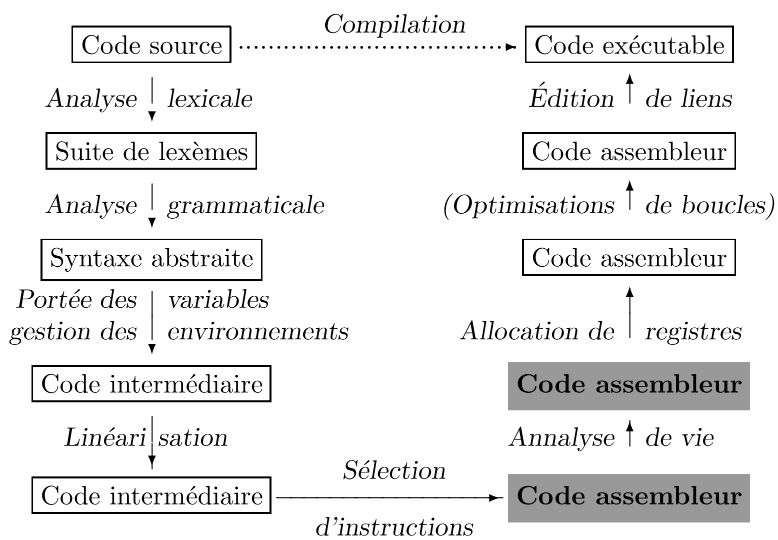
```
let emit_load_arg d o =
  emit (Oper "lw    ^d0,"^o^"($sp)", [], [d], None)

let emit_prolog_1_2 f =
  let f_size = Gen.label_string (Frame.frame_label f)^"_size" in
(* point d'entrée et allocation du frame *)
  emit_label (Frame.frame_label f) ;
  emit (Oper ("subu $sp, $sp, "^f_size, [], [], None)) ;
(* sauvegarde des callee-saves *)
  let saved_ra = new_temp () and saved_s0 = new_temp () ;
  emit_move saved_ra ra ; emit_move saved_s0 s0 ;
(* récupérer les arguments *)
  let [t1 ; t2 ; t3] = Frame.frame_args f in
  emit_move t1 a0 ;
  emit_load_arg t2 ("0+"^f_size) ;
  emit_load_arg t3 ("4+"^f_size) ;
(* rendre les sauvegardes des callee-saves, pour l'épilogue *)
  [saved_ra ; saved_s0]
```

FIGURE 7.14 – Multiplication et division immédiates

```
and emit_binop op e1 = function
  | Const i ->
      let s = emit_exp e1 and d = new_temp () in
(* selection ad-hoc pour quelques cas importants *)
      begin match op,i with
      | Times,2 -> emit_sll d s 1
      | Times,4 -> emit_sll d s 2
      | Div, 2  -> emit_sra d s 1
      | Div, 4  -> emit_sra d s 2
      | _,_     -> emit_op2 op d s i
      end ;
      d
  | e2 ->
      let s0 = emit_exp e1
      and s1 = emit_exp e2
      and d = new_temp () in
      emit_op3 op d s0 s1 ; d
```

Chapitre 8

Analyse de durée de vie



Le but de l'analyse de durée de vie (*liveness analysis*) est de déterminer les information de durée de vie des temporaires. Cette phase est un préalable indispensable à l'allocation de registres par coloriage de graphe qui sera l'objet de la leçon suivante.

8.1 Durées de vie

8.1.1 Temporaires vivants

Intuitivement, un temporaire est vivant en un point donné du code, si son contenu *en ce point* peut être lu par la suite. Le mot « vivant » signifie donc surtout utile. Or, si un temporaire est vivant en un point donné du code, son contenu doit se trouver quelque part, idéalement dans un registre machine. Il est clair que si deux temporaires t_1 et t_2 sont vivants en un même point, alors ces deux temporaires ne peuvent pas représenter le même registre machine. En revanche, si deux temporaires ne sont vivants simultanément en aucun point du code, alors ils peuvent représenter le même registre machine.

Précisons un peu sur un exemple. Soient les deux bout de code suivants, exprimés avec la syntaxe des instructions machine du début chapitre précédent :

C_1
$t_1 \leftarrow 1$
$\{t_1\}$
$t_2 \leftarrow 2$
$\{t_1, t_2\}$
$t_2 \leftarrow t_1 + t_2$

C_2
$t_1 \leftarrow 1$
$\{t_1\}$
$t_2 \leftarrow t_1 + 2$
$\{t_2\}$
$t_1 \leftarrow 3$
$\{t_1, t_2\}$
$t_2 \leftarrow t_2 + t_1$

Les « points donnés du code » se situent entre deux instructions, où sont montrés les temporaires vivants. On remarquera, dans le code C_2, que t_1 n'est pas vivant au deuxième point du programme. Une lecture de t_1 suit effectivement (dernière instruction), mais la valeur lue n'est pas celle que t_1 contient entre la deuxième et la troisième instruction. On peut donc reformuler la définition des temporaires vivants en disant qu'un temporaire est vivant en un point du programme lorsque, par la suite, il est lu avant d'être écrit. Enfin les points du code sont plutôt repérés par rapport aux instructions, on distingue alors l'entrée d'une instruction et sa sortie.

Pour chaque instruction i, on définit :

— $\texttt{Use}\,(i)$ l'ensemble des temporaires lus (ou encore utilisés) par i,
— $\texttt{Def}\,(i)$ l'ensemble des temporaires écrits (ou encore définis) par i,
— $\texttt{Succ}\,(i)$ l'ensemble des instructions qui peuvent suivre immédiatement i dans une exécution,
— $\texttt{In}\,(i)$ l'ensemble des temporaires vivants à l'entrée de i,
— $\texttt{Out}\,(i)$ l'ensemble des temporaires vivants à la sortie de i.

Les ensembles $\texttt{Use}\,(i)$, $\texttt{Def}\,(i)$ et $\texttt{Succ}\,(i)$ se définissent instruction par instruction et ce sont justement les informations mises en valeur par le type bizarre `Ass.instr` de la figure 7.5.

On peut définir $\texttt{Out}\,(i)$ formellement ainsi :

$$\texttt{Out}\,(i) = \left\{ t \;\middle|\; \exists i_1 \in \texttt{Succ}\,(i), \ldots, i_n \in \texttt{Succ}\,(i_{n-1}), \bigwedge \left\{ \begin{array}{l} t \in \texttt{Use}\,(i_n) \\ \forall k \in [1, n-1], t \notin \texttt{Def}\,(i_k) \end{array} \right. \right\}$$

C'est le pendant exact de la définition des temporaires vivants comme étant ceux qui seront lus avant d'être écrits. De même on définit :

$$\texttt{In}\,(i) = \left\{ t \;\middle|\; \exists i_1 = i, i_2 \in \texttt{Succ}\,(i_1), \ldots, i_n \in \texttt{Succ}\,(i_{n-1}), \bigwedge \left\{ \begin{array}{l} t \in \texttt{Use}\,(i_n) \\ \forall k \in [1, n-1], t \notin \texttt{Def}\,(i_k) \end{array} \right. \right\}$$

En corollaire de ces deux définitions on a les deux égalités :

$$\texttt{In}\,(i) = \texttt{Use}\,(i) \cup (\texttt{Out}\,(i) \setminus \texttt{Def}\,(i)) \qquad \texttt{Out}\,(i) = \bigcup_{i' \in \texttt{Succ}\,(i)} \texttt{In}\,(i')$$

Ces deux égalités permettent de calculer `Out` et `In` par itération. Avant de le prouver examinons le cas d'une séquence d'instructions. Si les instructions i_1 puis i_2 se suivent en séquence (*i.e.* on a $\texttt{Succ}\,(i_1) = \{i_2\}$), alors, les définitions entraînent immédiatement :

$$\texttt{Out}\,(i_1) = \texttt{In}\,(i_2) = (\texttt{Out}\,(i_2) \setminus \texttt{Def}\,(i_2)) \cup \texttt{Use}\,(i_2)$$

Par conséquent, dans le cas d'une séquence d'instructions, les temporaires vivants se calculent facilement en remontant le sens de l'exécution. Dans le cas de nos codes C_1 et C_2 on a donc en fait :

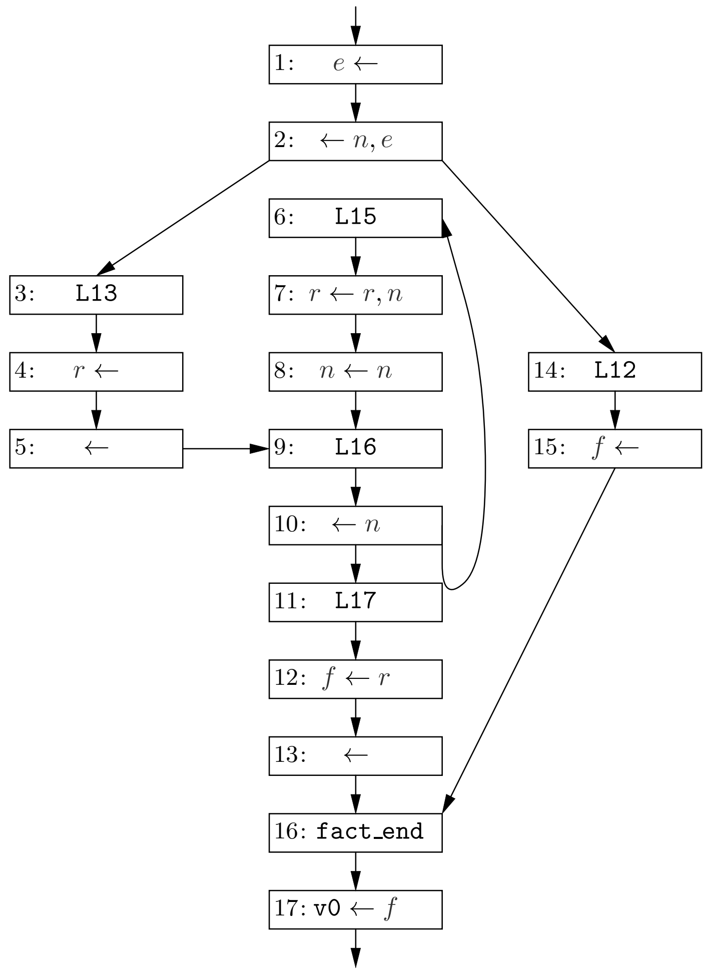
C_1
$T \setminus \{t_1, t_2\}$
$t_1 \leftarrow 1$
$(T \setminus \{t_2\}) \cup \{t_1\}$
$t_2 \leftarrow 2$
$(T \setminus \{t_2\} \cup \{t_1, t_2\}$
$t_2 \leftarrow t_1 + t_2$
T

C_2
$T \setminus \{t_1, t_2\}$
$t_1 \leftarrow 1$
$(T \setminus \{t_1, t_2\}) \cup \{t_1\}$
$t_2 \leftarrow t_1 + 2$
$(T \setminus \{t_1, t_2\}) \cup \{t_2\}$
$t_1 \leftarrow 3$
$(T \setminus \{t_2\}) \cup \{t_1, t_2\}$
$t_2 \leftarrow t_2 + t_1$
T

Où T est l'ensemble des temporaires vivants en sortie du code. On remarque au passage que les temporaires vivants en entrée de C_1 et C_2 se déduisent directement de ceux en sortie de ces codes. Ici il suffit d'enlever les temporaires détruits par les codes, qui par ailleurs ne lisent pas les contenus initiaux de t_1 et t_2.

FIGURE 8.1 – Graphe de flot adapté au calcul des durées de vies

```
1        li   e, 1
2        ble  n, e, L12
3   L13:
4        li   r, 1
5        b L16
6   L15:
7        mul  r, r, n
8        sub  n, n, 1
9   L16:
10       bgt  n, $zero, L15
11  L17:
12       move f, r
13       b fact_end
14  L12:
15       li   f, 1
16  fact_end:
17       move $v0, f
```



Considérons maintenant l'exemple plus compliqué du code de la figure 8.1. Un graphe de flot met en avant les informations pertinentes pour le calcul des durées de vies, temporaire lus et écrits, contrôle. Sur ce graphe, on constate par exemple que le temporaire e est vivant entre les instructions 1 et 2, tandis que temporaire f est vivant en entrée de l'instruction 17, en raison des séquences 15 (écriture), 16, 17 et 12 (écriture), 13, 16, 17. Le temporaire n est vivant de l'entrée de l'instruction 1 à la sortie de l'instruction 10, mais n'est plus vivant en entrée de l'instruction suivante 11 car aucune instruction ne le lit plus par la suite.

On notera qu'utiliser le graphe de flot revient à approximer les séquences d'instructions réellement exécutées par excès. On suppose que tous les chemins du graphe de flot seront pris lors de l'exécution. Il n'appartient pas à l'analyse de durée de vie de chercher à identifier quels sont les chemins réellement utiles, cette tâche est dévolue à d'autres analyses. Ces autres analyses ne pourront elle-même produire qu'une approximation du contrôle, mais cette approximation sera plus fine que l'approximation grossière qui consiste à considérer que tous les chemins peuvent être pris à l'exécution.

8.1.2 Calcul

Un code de longueur n $c_n = [i_1; \ldots; i_n]$ est une suite de n instructions, qui peuvent s'enchaîner lors d'une exécution du programme. On note $[\,]$ pour la séquence vide. Pour $n > 0$, on écrit librement $c_n = [c_{n-1}; i_n]$ ou $c_n = [i_1; c_{n-1}]$.

Pour une instruction i et un entier naturel k posons :

$$\mathtt{Succ}^0(i) = \{[\,]\} \qquad\qquad \mathtt{Succ}^{k+1}(i) = \bigcup_{i_1 \in \mathtt{Succ}(i)} \left(\bigcup_{c_k \in \mathtt{Succ}^k(i_1))} \{\, [i_1; c_k] \,\} \right)$$

Autrement dit, $\mathtt{Succ}^k(i)$ est l'ensemble des codes de longueur k qui peuvent être exécutés après l'instruction i. $\mathtt{Out}^n(i)$ est défini comme l'ensemble des temporaires vivants en sortie de k en ne considérant que les séquences de code de longueur au plus égale à n.

$$\mathtt{Out}^n(i) = \bigcup_{k=0}^{n} \left(\bigcup_{[c_{k-1}; i_k] \in \mathtt{Succ}^k(i)} \left\{ t \,\middle|\, \bigwedge \left\{ \begin{array}{l} t \in \mathtt{Use}(i_k) \\ \forall k' \in [1, k-1], t \notin \mathtt{Def}(i_{k'}) \end{array} \right. \right\} \right)$$

De même on définit $\mathtt{In}^n(i)$ en ne retenant que les séquences de longueur limitée par n dans la définition de $\mathtt{In}(i)$.

$$\mathtt{In}^n(i) =$$

$$\bigcup_{k=0}^{n} \left(\bigcup_{c_{k-1} \in \mathtt{Succ}^{k-1}(i)} \left\{ t \,\middle|\, \bigwedge \left\{ \begin{array}{l} t \in \mathtt{Use}(j_k) \\ \forall k' \in [1, k-1], t \notin \mathtt{Def}(j_{k'}) \end{array} \right. , \text{ avec } [j_1; \ldots; j_k] = [i; c_{k-1}] \right\} \right)$$

Avec deux abus de notation caractérisés, on a aussi défini $\mathtt{Out}^0(i) = \emptyset$ ($\mathtt{Succ}^0(i)$ ne contient pas de séquence de la forme $[c_{k-1}; i_k]$) et $\mathtt{In}^0(i) = \emptyset$ ($\mathtt{Succ}^{-1}(i) = \emptyset$). Ces définitions entraînent que les $\mathtt{Out}^n(i)$ et les $\mathtt{In}^n(i)$ sont des suites d'ensembles croissantes au sens large et dont les limites sont $\mathtt{Out}$ et $\mathtt{In}$.

$$\mathtt{Out}(i) = \bigcup_{n \in \mathbb{N}} \mathtt{Out}^n(i) \qquad\qquad \mathtt{In}(i) = \bigcup_{n \in \mathbb{N}} \mathtt{In}^n(i)$$

Par ailleurs, on a les deux égalités :

$$\mathtt{In}^{n+1}(i) = \mathtt{Use}(i) \cup (\mathtt{Out}^n(i) \setminus \mathtt{Def}(i)) \qquad\qquad \mathtt{Out}^n(i) = \bigcup_{j \in \mathtt{Succ}(i)} \mathtt{In}^n(j)$$

Montrons par exemple la première égalité. Soit donc t dans $\mathtt{In}^{n+1}(i)$. En procédant par équivalences et en notant $[i; c_{k-1}]$ comme $[j_1; j_2; \ldots; j_k]$, il vient :

$$\exists\, k \in [1, n{+}1], \exists\, c_{k-1} \in \texttt{Succ}^{k-1}\,(i),\ \Big(t \in \texttt{Use}\,(j_k) \bigwedge \forall k' \in [1, k-1], t \notin \texttt{Def}\,(j_{k'})\Big)$$

$$\Updownarrow$$

$$\bigvee \left\{ \begin{array}{l} t \in \texttt{Use}\,(i), \quad (k=1) \\ \exists\, k \in [2, n{+}1], \exists\, c_{k-1} \in \texttt{Succ}^{k-1}\,(i),\ (t \in \texttt{Use}\,(j_k) \bigwedge \forall k' \in [1, k-1], t \notin \texttt{Def}\,(j_{k'})) \end{array} \right.$$

$$\Updownarrow$$

$$\bigvee \left\{ \begin{array}{l} t \in \texttt{Use}\,(i), \quad (k=1) \\ \\ \exists\, k \in [2, n{+}1], \exists\, c_{k-1} \in \texttt{Succ}^{k-1}\,(i),\ \bigwedge \left\{ \begin{array}{l} t \in \texttt{Use}\,(j_k) \\ t \notin \texttt{Def}\,(i), \quad (j_1 = i) \\ \forall k' \in [2, k-1], t \notin \texttt{Def}\,(j_{k'}) \end{array} \right. \end{array} \right.$$

$$\Updownarrow$$

$$\bigvee \left\{ \begin{array}{l} t \in \texttt{Use}\,(i), \quad (k=1) \\ \bigwedge \left\{ \begin{array}{l} t \notin \texttt{Def}\,(i), \\ \exists\, k \in [1, n], \exists\, c_k \in \texttt{Succ}^{k}\,(i),\ t \in \texttt{Use}\,(i_k) \bigwedge \forall k' \in [1, k-1], t \notin \texttt{Def}\,(i_{k'}) \end{array} \right. \end{array} \right.$$

Où, dans la dernière proposition, on a effectué un audacieux changement d'indice et noté $c_k = [i_1; \ldots; i_k]$.

Nous pouvons calculer $\texttt{In}^k\,(i)$ et $\texttt{Out}^k\,(i)$ pour tous les entiers k et toutes les instructions du programme. Il suffit de partir de $\texttt{Out}^0\,(i) = \emptyset$ (La valeur de $\texttt{In}^0\,(i)$ est indifférente) et d'utiliser constructivement les équations :

$$\texttt{In}^{n+1}\,(i) = \texttt{Use}\,(i) \cup (\texttt{Out}^{n}\,(i) \setminus \texttt{Def}\,(i)) \qquad \texttt{Out}^{n+1}\,(i) = \bigcup_{j \in \texttt{Succ}\,(i)} \texttt{In}^{n+1}\,(j)$$

Or, les suites $\texttt{In}^k\,(i)$ et $\texttt{Out}^k\,(i)$ sont croissantes au sens large, d'après leur définitions, et bornées, puisqu'il y a un nombre fini de temporaires dans un programme donné. Par conséquent ces suites sont stationnaires à partir d'un certain rang où elles valent leurs limites respectives $\texttt{In}\,(i)$ et $\texttt{Out}\,(i)$.

Autrement dit, il est temps de remarquer que $\texttt{In}\,(i)$ est une commodité et que $\texttt{Out}\,(i)$ est le plus petit point fixe de la fonction :

$$O(i) \longrightarrow \bigcup_{j \in \texttt{Succ}\,(i)} F_j(O(j))$$

En notant $F_i(X) = \texttt{Use}\,(i) \cup (X \setminus \texttt{Def}\,(i))$.

Et de fait, nous venons que montrer que $\texttt{Out}\,(i)$ et $\texttt{In}\,(i)$ sont la plus petite solution des équations :

$$\texttt{In}\,(i) = \texttt{Use}\,(i) \cup (\texttt{Out}\,(i) \setminus \texttt{Def}\,(i)) \qquad \texttt{Out}\,(i) = \bigcup_{j \in \texttt{Succ}\,(i)} \texttt{In}\,(j)$$

Parfois on se contente donc de *définir* $\texttt{Out}\,(i)$ et $\texttt{In}\,(i)$ comme la plus petite solution de ces équations, vues comme le point fixe d'une fonction. Il faut bien reconnaître que l'intuition est un peu enfouie, en échange d'une définition plus concise. On note aussi qu'il n'est pas immédiatement apparent que la récursion est bien fondée, c'est à dire que la fonction dont on calcule le point fixe est monotone.

Dans cette seconde définition, on aurait tort d'oublier de spécifier *plus petit* point fixe. En effet, ces équations admettent de nombreuses autres solutions. Soit $T(i)$ des ensemble de temporaires pour le moment arbitraires. On a alors :

$$\texttt{Use}\,(i) \cup ((\texttt{Out}\,(i) \cup T(i)) \setminus \texttt{Def}\,(i)) = \texttt{Use}\,(i) \cup (\texttt{Out}\,(i) \setminus \texttt{Def}\,(i)) \cup (T(i) \setminus \texttt{Def}\,(i)) =$$

$$\texttt{In}\,(i) \cup (T(i) \setminus \texttt{Def}\,(i))$$

$$\bigcup_{j \in \texttt{Succ}\,(i)} (\texttt{In}\,(j) \cup T_j) = \texttt{In}\,(i) \cup \left(\bigcup_{j \in \texttt{Succ}\,(i)} T_j \right)$$

Il suffit alors, pour obtenir une autre solution de trouver des $T(i)$ dont un au moins est non-vide et pour lesquels on a :

$$T(i) = T(i) \setminus \texttt{Def}\,(i) \qquad\qquad T(i) = \bigcup_{j \in \texttt{Succ}\,(i)} T(j)$$

Ce qui est faisable, en posant par exemple, $T(i) \neq \emptyset$ et $T(i) \cap \texttt{Def}\,(i) = \emptyset$ pour une instruction sans successeur, puis en calculant tous les $T(i)$ par point fixe... On peut aussi donner une solution triviale en posant $T(i) = t$ pour toutes les instructions i, où t est un temporaire qui n'appartient à aucun $\texttt{Def}\,(i)$.

8.1.3 Calcul en pratique

Nous ne calculons pas les $\texttt{Out}^k\,(i)$ pour eux-mêmes, mais pour leurs limites. Supposons que les instructions i_1, i_2, ... i_n de notre programme $\mathcal{P}$ sont indicées par les entiers d'un intervalle $[1, n]$. Cela revient à ordonner totalement les instructions et à les désigner par leur rang selon l'ordre choisi. Le calcul de la suite $\texttt{Out}^k$ telle que définie demande de maintenir deux tableaux et de procéder selon cet algorithme décrit en Caml de cuisine :

```
for i=1 to n  do Out(i) <- ∅ done ;
do
  for i=1 to n do Out'(i) <- Out(i)  done ;
  for i=1 to n do
    Out(i) <- (⋃_{j∈Succ(i)} (Out'(j) \ Def(j)) ∪ Use(j))
  done
until ∀i ∈ [1,n], Out(i) = Out'(i)
```

C'est à dire que, à chaque tour de la boucle `do...until` on calcule les $\texttt{Out}^{k+1}\,(i)$ en fonction des $\texttt{Out}^k\,(i)$, conformément à la définition. Mais considérons une nouvelle suite $\texttt{Speed}^k\,(i)$ calculée à l'aide d'un seul tableau :

```
for i=1 to n do Speed(n) <- ∅ done ;
let encore = ref true in
while !encore do
  encore := false ;
  for i=1 to n do
    let prev = Speed(i) in
    Speed(i) <- (⋃_{j∈Succ(i)} (Speed(j) \ Def(j)) ∪ Use(j)) ;
    encore := !encore || (prev <> Speed(i))
  done
done
```

La définition algorithmique de la suite $\texttt{Speed}^k\,(i)$ est de loin la plus naturelle, mais on peut aussi lui donner la définition plus formelle suivante :

$$\texttt{Speed}^0(i) = \emptyset$$

$$\texttt{Speed}^{k+1}(i) = \left(\bigcup_{j \in \texttt{Succ}(i),\, j<i} F_j(\texttt{Speed}^{k+1}(j))\right) \bigcup \left(\bigcup_{j \in \texttt{Succ}(i),\, j\geq i} F_j(\texttt{Speed}^{k}(j))\right)$$

Avec, rappelons le, $F_j(X) = (X \setminus \texttt{Def}(j)) \cup \texttt{Use}(j)$. Par monotonie des F_j on a alors :

$$\texttt{Out}^k(i) \subseteq \texttt{Speed}^k(i) \subseteq \texttt{Out}(i)$$

C'est à dire que la suite des $\texttt{Speed}^k(i)$ converge également vers $\texttt{Out}(i)$. Par un bon choix de l'ordonnancement des instructions du programme $\mathcal{P}$, on accélère notablement la convergence. Le bon choix est d'ordonner les instructions à l'inverse de l'ordre d'exécution de façon à augmenter la taille de l'ensemble des $j \in \texttt{Succ}(i)$, $j < i$.

Examinons l'effet produit sur l'exemple simple d'un code C exécuté en séquence :

C	$\texttt{Out}^0$	$\texttt{Out}^1$	$\texttt{Out}^2$	$\texttt{Out}^4$	$\texttt{Out}^5$
$t_1 \leftarrow 1$	$\emptyset$	$\emptyset$	$\emptyset$	$\{t_1\}$	$\{t_1\}$
$t_2 \leftarrow 2$	$\emptyset$	$\emptyset$	$\{t_1, t_2\}$	$\{t_1, t_2\}$	$\{t_1, t_2\}$
$t_2 \leftarrow t_1 + t_2$	$\emptyset$	$\{t_1, t_2\}$	$\{t_1, t_2\}$	$\{t_1, t_2\}$	$\{t_1, t_2\}$
$t_3 \leftarrow t_2 * t_1$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

i	C	$\texttt{Speed}^0$	$\texttt{Speed}^1$	$\texttt{Speed}^2$
4	$t_1 \leftarrow 1$	$\emptyset$	$\{t_1\}$	$\{t_1\}$
3	$t_2 \leftarrow 2$	$\emptyset$	$\{t_1, t_2\}$	$\{t_1, t_2\}$
2	$t_2 \leftarrow t_1 + t_2$	$\emptyset$	$\{t_1, t_2\}$	$\{t_1, t_2\}$
1	$t_3 \leftarrow t_2 * t_1$	$\emptyset$	$\emptyset$	$\emptyset$

Avec le choix de $\texttt{Succ}(i) = i - 1$, on voit que, dans le cas d'un code en séquence de n instructions, la stabilisation est atteinte en $n + 1$ étapes pour la suite $\texttt{Out}^k$ et 2 étapes pour la suite $\texttt{Speed}^k$.

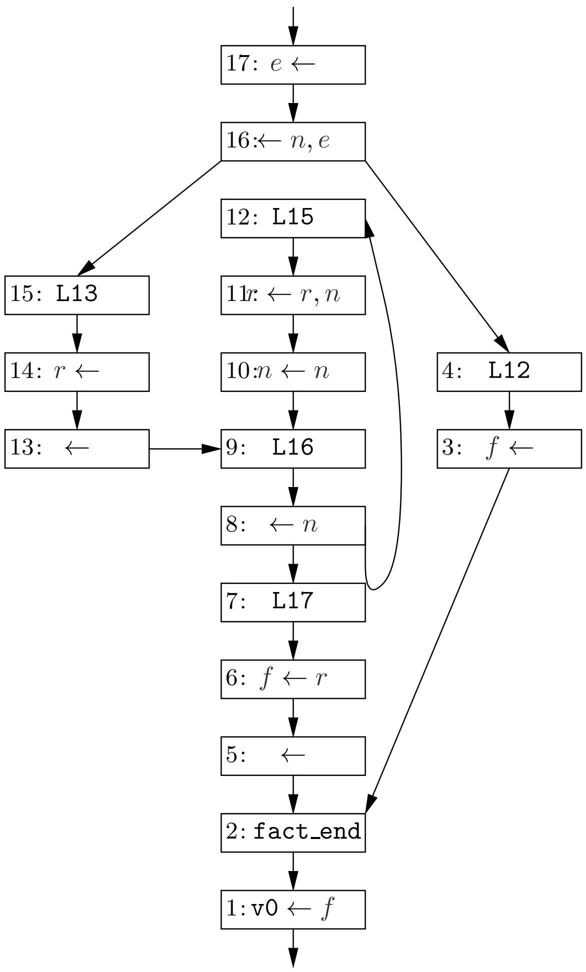
Dans le cas d'un contrôle plus complexe, nous avons encore intérêt à ordonner les instructions le plus possible selon l'ordre inverse de leur exécution. Mais satisfaire complètement la contrainte $j \in \texttt{Succ}(i)$ entraîne $j < i$ n'est plus possible en raison des boucles. On se contente alors d'un ordre (quasi-)topologique inverse, de sorte que les successeurs sont généralement traités avant leur prédécesseurs. De toute évidence, il convient au moins de traiter les instructions exécutées en séquence dans l'ordre inverse de leur exécution. On peut donc se contenter de l'ordre inverse de la présentation des instructions du programme. Dans le cas de notre compilateur qui ne réordonne pas les traces, cet ordre inverse de la présentation correspond d'ailleurs à ordre (quasi-)topologique inverse.

De fait, une telle numérotation des sommets du graphe de flot de la figure 8.1 permet de calculer les $\texttt{Out}(i)$ en trois itérations, comme le montre la figure 8.2. Ici, on a presque toujours $j \in \texttt{Succ}(i)$ entraîne $j < i$ sauf pour $\texttt{Succ}(8) = \{7, 12\}$. On a donc, puisque $\texttt{Speed}^1(7) = \{r\}$ et $\texttt{Speed}^0(12) = \emptyset$, $\texttt{Speed}^1(8) = ((\{r\}\setminus) \cup \emptyset) \cup (\emptyset\setminus) \cup \emptyset) = \{r\}$. À la même itération on a $\texttt{Speed}^1(12) = \{n, r\}$ puisque le successeur 11 de 12 lit ces deux registres. Dès lors à itération suivante on a, $\texttt{Speed}^1(8) = ((\{r\}\setminus) \cup \emptyset) \cup (\{n, r\}\setminus) \cup \emptyset) = \{n, r\}$. Le point fixe est atteint comme on s'en rendrait compte en calculant $\texttt{Speed}^3$ (non-montré).

Les $\texttt{In}(i)$ sont montrés pour mémoire. On notera que le programme présenté est un bout de fonction. Si on avait analysé le code complet, alors le registre `v0` serait également vivant en sortie de la dernière instruction (numérotée 1), puisque ce registre est ≪ lu ≫ par l'instruction de retour **j** `$ra` qui vient ensuite.

Le coût du calcul des durées de vie est potentiellement assez élevé. Soit un code de taille n et contenant n temporaires distincts. Que l'on choisisse le calcul des $\texttt{Out}^k$ ou des $\texttt{Speed}^k$. Un passage dans la boucle principale se solde par de l'ordre de n opérations ensemblistes, qui chacune coûte disons de l'ordre de n opérations élémentaires. Soit un coût en n^2 pour une itération. Dans le pire

FIGURE 8.2 – Calcul accéléré des temporaires vivants



i	Speed^0	Speed^1	Speed^2	In
1	$\emptyset$	$\emptyset$	$\emptyset$	$\{f\}$
2	$\emptyset$	$\{f\}$	$\{f\}$	$\{f\}$
3	$\emptyset$	$\{f\}$	$\{f\}$	$\emptyset$
4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
5	$\emptyset$	$\{f\}$	$\{f\}$	$\{f\}$
6	$\emptyset$	$\{f\}$	$\{f\}$	$\{r\}$
7	$\emptyset$	$\{r\}$	$\{r\}$	$\{r\}$
8	$\emptyset$	$\{r\}$	$\{n, r\}$	$\{n, r\}$
9	$\emptyset$	$\{n, r\}$	$\{n, r\}$	$\{n, r\}$
10	$\emptyset$	$\{n, r\}$	$\{n, r\}$	$\{n, r\}$
11	$\emptyset$	$\{n, r\}$	$\{n, r\}$	$\{n, r\}$
12	$\emptyset$	$\{n, r\}$	$\{n, r\}$	$\{n, r\}$
13	$\emptyset$	$\{n, r\}$	$\{n, r\}$	$\{n, r\}$
14	$\emptyset$	$\{n, r\}$	$\{n, r\}$	$\{n\}$
15	$\emptyset$	$\{n\}$	$\{n\}$	$\{n\}$
16	$\emptyset$	$\{n\}$	$\{n\}$	$\{e, n\}$
17	$\emptyset$	$\{e, n\}$	$\{e, n\}$	$\{n\}$

des cas et à chaque itération un seul des ensembles croît d'un seul élément, et on peut donc itérer au pire n^2 fois. Soit un coût en n^4 pour l'ensemble du calcul des durées de vie. Heureusement comme nous l'avons vu, ce coût est rarement atteint en pratique. Selon notre mesure rapide et dans le cas du code en séquence, le coût du calcul naïf des $\mathtt{Out}\ k$ est en n^3 et celui des $\mathtt{Speed}^k$ est en n^2.

On peut aussi se débrouiller pour diminuer fortement le coût effectif des opérations ensemblistes. Un truc très classique consiste à représenter les ensembles par des vecteurs de bits, les opérations ensemblistes ont alors des pendants directs dans les opérations logiques sur les entiers, (l'union est le ou logique etc.). Dès lors, si le nombre de temporaires distincts est faible (genre inférieur à 32 ou 64) et au prix d'un encodage des temporaires dans les petits entiers, on peut même parler un peu abusivement de coût constant pour les opérations sur les ensembles. En pratique nous n'utiliserons pas cette représentation des ensembles, car elle manque un peu de souplesse.

Un autre gain important en pratique est obtenu en calculant les durées de vie d'abord sur un graphe de flot dont les sommet sont les blocs de base (les blocs de base sont les suites maximales d'instructions nécessairement exécutées en séquence voir section 6.5). En effet, du point de vue des durées de vie les blocs de base se comportent comme de grosses instructions.

Soit donc un bloc $b = [i_1; i_2; \ldots; i_n]$ avec $\mathtt{Succ}\,(i_k) = \{i_{k+1}\}$ pour $k \in [1, n-1]$. $\mathtt{In}\,(b) = \mathtt{In}\,(i_i)$ se calcule alors à partir de $\mathtt{Out}\,(b) = \mathtt{Out}\,(i_n)$ en constatant $\mathtt{Out}\,(i_k) = \mathtt{In}\,(i_{k+1})$ dans le bloc et en écrivant donc :

$$\texttt{In}\,(b) = (\texttt{Out}\,(i_1) \setminus \texttt{Def}\,(i_1)) \cup \texttt{Use}\,(i_1) \ldots \qquad \texttt{Out}\,(i_k) = (\texttt{Out}\,(i_{k+1}) \setminus \texttt{Def}\,(i_{k+1})) \cup \texttt{Use}\,(i_{k+1}) \ldots$$

$$\texttt{Out}\,(i_n) = \texttt{Out}\,(b)$$

Soit plus directement $\texttt{In}\,(b) = (\texttt{Out}\,(b) \setminus \texttt{Def}\,(b)) \cup \texttt{Use}\,(b)$, avec :

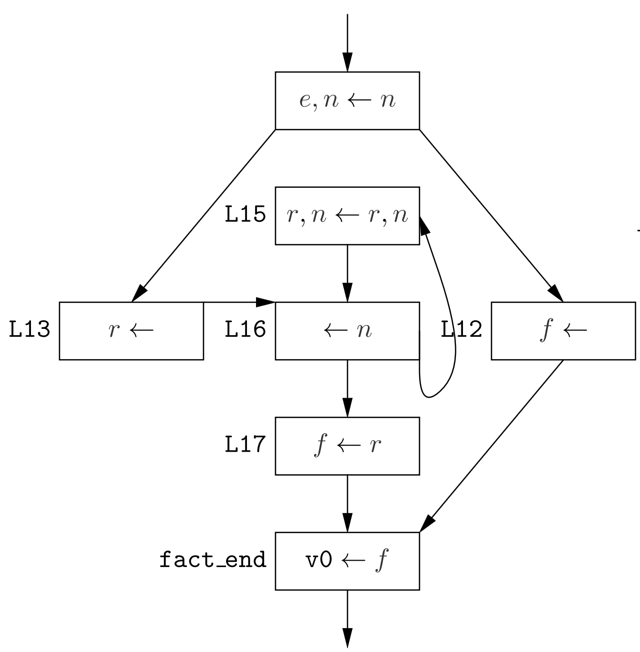
$$\texttt{Def}\,(b) = \bigcup_{k=1}^{n} \texttt{Def}\,(i_k), \qquad \texttt{Use}\,(b) = \left(\bigcup_{k=1}^{n} \texttt{Use}\,(i_k) \setminus \left(\bigcup_{j=1}^{k-1} \texttt{Def}\,(i_j) \right) \right)$$

L'apparente complexité de la formule des `Use` ne doit pas troubler, elle se comprend bien si on se rappelle que les temporaires présentés en entrée du bloc ne sont effectivement lus par une instruction i_k du bloc, que si leur contenu n'a pas été changé par une instruction précédant i_k.

Au prix donc d'un calcul préalable des blocs de base, des `Def` et des `Use` de chaque bloc, puis d'une reconstitution des durées de vie instruction par instruction, on diminue notablement le nombre de sommets du graphe sujet à itération. En gros, on divise la nombre de sommets par la taille moyenne des blocs. Le calcul sur le graphe des blocs de base s'impose naturellement dans le cas fréquent d'un compilateur qui maintient la structuration du code selon les blocs de base durant tout le back-end.

Ainsi, dans le cas du code la figure 8.1 on obtient le graphe des blocs de base de la figure 8.3, qui comprend seulement 7 sommets, à comparer aux 17 sommets du graphe de flot des instructions de la figure 8.2.

FIGURE 8.3 – Calcul des temporaires vivants sur les blocs de base



b	Speed^0	Speed^1	Speed^2	In
end	$\emptyset$	$\emptyset$	$\emptyset$	$\{f\}$
L12	$\emptyset$	$\{f\}$	$\{f\}$	$\emptyset$
L17	$\emptyset$	$\{f\}$	$\{f\}$	$\{r\}$
L16	$\emptyset$	$\{r\}$	$\{n, r\}$	$\{n, r\}$
L15	$\emptyset$	$\{n, r\}$	$\{n, r\}$	$\{n, r\}$
L13	$\emptyset$	$\{n, r\}$	$\{n, r\}$	$\{n\}$
start	$\emptyset$	$\{n\}$	$\{n\}$	$\{n\}$

8.2 Graphe d'interférence

Les informations de durée de vie sont utilisables par plusieurs optimisations du compilateur. Par exemple, si nous avons une instruction $t \leftarrow \ldots$, où $\ldots$ ne fait pas d'effet de bord (typiquement, si nous ignorons les débordements arithmétiques) et que le temporaire t n'est pas vivant en sortie de l'instruction, alors nous pouvons éliminer cette instruction.

Mais, en ce qui nous concerne les durées de vie des temporaires servent à l'allocation de registres. Nous disposons de n registres machines r_1, r_2, … , r_n à répartir entre m temporaires t_1, t_2, … , t_m.

Nous disons que deux temporaires *interfèrent* si on ne peut pas leur allouer le même registre. La relation d'interférence est certainement non-reflexive, symétrique et non-nécessairement transitive. Il est clair que si deux temporaires t_i et t_j sont vivants en un même point du programme, on doit leur allouer des registres différents. Notons qu'un temporaire t peut être un registre r (le registre `a0` du premier argument exemple). Il n'y a pas lieu, dans ce cas, d'allouer un registre à $t = r$. Mais on doit évidemment considérer les temporaires qui interfèrent avec t comme ne pouvant pas occuper le registre r.

Le recouvrement des durées de vie est la principale cause d'interférence entre temporaires. Mais il y en a d'autres. Considérons d'instruction d'appel de sous-routine **jal** f. Cette instruction écrit dans le registre `ra`. Pire elle écrit potentiellement dans tous les registres arguments, résultat et tous les caller-saves. Ces conditions interdisent d'allouer un des registres précités à tout temporaire qui est vivant à travers l'appel de sous-routine. Et ceci même si nous ne sommes pas stricto-sensu en présence de durées de vies non-disjointes. Il peut se trouver des interférence encore plus éloignées des durées de vie, supposons que la machine ciblée possède une instruction $t \leftarrow \ldots$ dont le résultat ne peut pas être rangé dans un registre particulier r_1, alors le temporaire t interfère avec le registre r_1. Ce type d'interférence se produit par exemple dans le cas des processeurs Motorola 680X0, qui possèdent deux classes de registres, une pour les données et l'autre pour les adresses, toutes les opérations ne pouvant pas utiliser tous les registres indifféremment.

Mais dans le cas du MIPS, les registres sont réellement d'usage général. Dès lors, et grâce aux ≪ lus ≫ et≪ écrits ≫ étendus mis en place lors de la sélection d'instructions, on peut calculer les interférences entre temporaires par un simple parcours des instructions i, c'est à dire des sommets du graphe de flot. Ce parcours distingue deux cas :

— Si i est une instruction qui n'est pas un transfert entre temporaires. Alors les temporaires de $\texttt{Def}\,(i)$ interfèrent avec tous les temporaires de $\texttt{Out}\,(i) \setminus \texttt{Def}\,(i)$.

— Si i est un transfert du temporaire s vers le temporaire d. Alors d interfère avec tous les temporaires de $\texttt{Out}\,(i) \setminus \{d, s\}$. Un transfert $d \leftarrow s$ ne crée pas d'interférence entre s et d. En effet, en sortie de l'instruction, rien n'empêche les temporaires s et d d'occuper le même registre machine, bien au contraire.

Ce simple parcours permet de détecter (et d'enregistrer) toutes les interférences d'un programme en assembleur opérant sur les temporaires. En effet, si une instruction produit un résultat dans un temporaire, alors ce temporaire ne peut pas occuper le même registre que tout autre temporaire vivant en sortie de l'instruction (sauf dans le cas du transfert et de sa source, si cette dernière est encore en vie après le transfert). En revanche, une destination de l'instruction peut occuper le même registre que tout temporaire qui n'est pas vivant en sortie de l'instruction.

La relation d'interférence est idéalement représentée par un graphe non-orienté dont les sommets sont les temporaires et dont les arcs expriment l'interférence de deux temporaires. Puisque la relation d'interférence est par définition non-réflexive, il n'y a jamais d'arcs d'un sommet vers lui-même.

On profite du calcul de la relation d'interférence pour calculer une autre relation : deux temporaires sont reliés par un transfert si il existe une instruction `move` de l'un vers l'autre. On peut représenter cette nouvelle relation sur le graphe d'interférence en ajoutant des arcs distincts des arcs d'interférence.

Reprenons le code en séquence C de la section précédente, ainsi que les informations pertinentes des `Def` et des `Out`.

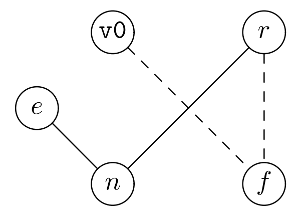
i	C	`Def`	`Out`
4	$t_1 \leftarrow 1$	t_1	t_1
3	$t_2 \leftarrow 2$	t_2	t_1, t_2
2	$t_2 \leftarrow t_1 + t_2$	t_2	t_1, t_2
1	$t_3 \leftarrow t_2 * t_1$	t_3	

On voit alors que les deux temporaires t_1 et t_2 interfèrent, en raison par exemple de $\texttt{Def}\,(2) = \{t_2\}$ et $\texttt{Out}\,(2) = \{t_1, t_2\}$.

Dans le cas de la figure 8.2 et après résumé des informations pertinentes, on obtient le graphe d'interférence suivant.

i	`move`	`Def`	`Out`
1	`v0` $\leftarrow f$	`v0`	`v0`
2			f
3	$f \leftarrow 1$	f	f
4			
5			f
6	$f \leftarrow r$	f	f
7			r
8			n, r

i	`move`	`Def`	`Out`
9			n, r
10		n	n, r
11		r	n, r
12			n, r
13			n, r
14	$r \leftarrow 1$	r	n, r
15			n
16			n
17	$e \leftarrow 1$	e	n



Les arcs des « `move` » sont en pointillés.

On constate d'abord que n et e interfèrent (à cause de $e \in \texttt{Def}\,(17)$ et de $n \in \texttt{Out}\,(17)$), puis que n et r interfèrent (pour trois raisons, voir les instructions 10, 11 et 14). Au vu du graphe d'interférence, on voit que f, r et e peuvent tous occuper le registre `v0`. Tandis qu'un autre registre est nécessaire pour n. On notera que les arcs `move` suggèrent puissamment d'allouer un même registre à `v0`, f et r, tandis que ce sont bien les arcs d'interférence qui nous disent que c'est possible.

8.3 Réalisation

La programmation sur les graphes est notoirement un peu difficile. Cela tient souvent au manque de séparation entre les structures de données qui représentent les graphes et les fonctions qui calculent sur les graphes. En effet il n'est pas évident de définir les graphes par des structures de données abstraites et de garder une bonne efficacité, en raison notamment de la grande variété en pratique des structures de « graphes ». Je m'essaie pourtant ici à un tel exercice dans un souci de clarté.

8.3.1 Environnement de programmation

Nous avons d'abord besoin de manipuler des ensembles de temporaires. Supposons donné un module `Smallset` qui réalise les opérations courantes sur les ensembles (figure 8.4). Les ensembles `Smallset` sont réalisés par des listes ordonnées, ce qui, par rapport à la réalisation par arbre binaires équilibrés de la bibliothèque standard, pénalise les opérations `mem`, `add` etc. au profit des opérations entre ensembles `union`, `diff` etc. et surtout de la simplicité et de l'efficacité sur les petits ensembles.

Le gros morceau est la représentation des graphes. Soit donc le module `Graph` dont l'interface est donnée à la figure 8.5. L'interface est suffisament commentée. On remarquera que le module `Graph` se charge de garantir l'existence d'au plus un arc entre deux sommets. Le type des sommets (ou nœuds) `'a node` est abstrait et paramétré par le type `'a` des informations associées aux sommets. On passe donc logiquement en argument une valeur de type `'a` à la fonction `new_node` de création des sommets. Le typage de Caml impose de passer une information bidon lors de la création du graphe (fonction `create`).

8.3.2 Calcul des durées de vie

Dans le cas du graphe de flot, les informations à associer aux sommets sont, une instruction, et ses ensembles `Use`, `Def`, `In` et `Out`. On définit donc le type suivant :

Figure 8.4 – Interface du module `Smallset` des ensembles (de temporaires).

```
(*
  Petite réalisation des ensembles.
   - Les ensembles sont encodés comme des listes ordonnées, selon
     l'ordre générique de Caml « < »
   - Toutes les opérations sont en gros linéaires en fonction du cardinal
     des ensembles passés en argument.
*)

type 'a set

val eqset : 'a set -> 'a set -> bool
   (* égalité sur les ensembles, ne pas supposer que « = » fonctionne, même si c'est le cas *)

val choose : 'a set -> 'a option
    (* renvoie un élément quelconque *)
val singleton : 'a -> 'a set
    (* créer un singleton *)
val of_list : 'a list -> 'a set
    (* "of_list l" crée l'ensemble dont les éléments sont l, coût en n Log(n) *)
val to_list : 'a set -> 'a list
    (* "to_list s" remvoie la liste des éléments de s *)

(* Les fonctions suivantes parlent d'elles-mêmes *)
val empty : 'a set
val is_empty : 'a set -> bool
val mem : 'a -> 'a set -> bool
val union : 'a set -> 'a set -> 'a set
val union_list : 'a set list -> 'a set
(* "union_list l" renvoie la réunion des ensembles de l, coût en n Log(n) *)
val diff : 'a set -> 'a set -> 'a set
val inter : 'a set -> 'a set -> 'a set

val add : 'a -> 'a set -> 'a set
val remove : 'a -> 'a set -> 'a set

val iter : 'a set -> ('a -> unit) -> unit
    (* "iter f s" applique f une fois à chaque élément de s, ordre indéfini *)
```

FIGURE 8.5 – Interface du module Graph des graphes orientés.

```
(*******************)
(* Graphes orientés *)
(*******************)

(* en cas de malheur *)
exception Error of string

(* Types de sommets et des graphes *)
type 'a node
and 'a t

val create : 'a -> 'a t
(* Créer un nouveau graphe, initialement vide, les sommets contiendront l'information 'a *)

val new_node : 'a t -> 'a -> 'a node
(* « new_node g i » ajoute un noeud d'information i au graphe g par effet de bord *)

val new_edge : 'a t -> 'a node -> 'a node -> unit
(* « new_edge g n1 n2 » ajoute un arc de n1 vers n2, si il n'existe pas déjà *)

val nodes : 'a t -> 'a node list
(* Tous les noeuds, dans l'ordre de leur création *)

val info : 'a t -> 'a node -> 'a
(* Le contenu d'un noeud *)

val succ : 'a t -> 'a node -> 'a node list
(* Les successeurs d'un noeud *)

val iter : 'a t -> ('a node -> unit) -> unit
(* « iter g f » itère la fonction f sur les noeuds (ordre de création) *)

val debug : out_channel -> (out_channel -> 'a node -> unit) -> 'a t -> unit
(* Affichage pour le debug *)
```

```
type flowinfo = {
    instr : Ass.instr ;               (* instruction *)
    def : temp set;  use : temp set; (* détruits et lus *)
    mutable live_in : temp set;       (* sans commentaire *)
    mutable live_out : temp set;
  }

type flowgraph = flowinfo Graph.t
(* Type des graphes de flots  décorés des  live-in/live-out *)

val flow : Ass.instr list -> flowgraph
(* Fabrication  du graphe de  flot ,  décoré par  les  durées de vie  *)
```

Les champs `live_in` et `live_out` sont mutables, car ils ne sont pas connus à la création du graphe, mais calculés par la suite. De fait, la fonction `flow` est très simple : création du graphe, puis calcul des durées de vie.

```
let flow code =
  let g = mk_graph code in
  fixpoint g ;
  g
```

Pour fixer un peu les idées voici une fonction `mk_graph` possible. On crée le graphe en deux temps, d'abord les sommets :

```
open Smallset

let mk_info i = match i with
| Oper (_, src, dest, _) ->
   {instr=i ; def = of_list dest ; use = of_list src ;
    live_in = empty ; live_out = empty}
| Move (_,src, dest, _) ->
   {instr = i ; def = singleton dest ; use = of_list src ;
    live_in = empty, live_out = empty}
| Label (_,_) ->
   {instr = i ; def = empty ; use = empty ;
    live_in = empty, live_out = empty}

let lab2node = Hashtbl.create 17

let rec mk_nodes g = function
| []     -> ()
| i::rem ->
    let n = Graph.mk_node g (mk_info i) in
    begin match i with
    | Label (_,lab) -> Hashtbl.add lab2node lab n
    | _             -> ()
    end ;
    mk_nodes g rem
```

C'est la fonction `mk_nodes` ci-dessus qui ajoute les sommets au graphe `g` passé en argument. Elle procède par un simple parcours de la liste d'instructions (assembleur) passée en second argument. À chaque somment correspond des informations, regroupant l'instruction elle même (champ `instr`) les temporaires lus et écrits (champs `use` et `def`). Les informations de *liveness* (champs `live_in` et `live_out`) sont initialisées à l'ensemble vide. Au passage, on remplit une table de hachage lab2node qui réalise une association des étiquettes aux nœuds, utile pour créer les arcs résultant des sauts dans une seconde étape.

Les arcs sont créés par un parcours de la liste des sommets du graphe (Obtenue à partir du graphe g, par Graph.nodes g) qui, rappelons le, portent une instruction dans leur champ instr de leur information La création des arcs diffère selon que cette instruction est un saut (et alors il faut aller chercher les sommets cibles à partir de leurs étiquettes et à l'aide de la table de hachage de l'étape précédente) ou une instruction qui transfère le contrôle en séquence (et alors la cible de l'arc est l'instruction suivante, qui doit exister). Il est donc critique que la liste des sommetes du graphe reflète l'ordre de création des sommets, qui est ici l'ordre des instructions dans la liste initiale.

```
let rec mk_edges g nodes = match nodes with
| [] -> ()
| n::rem ->
    begin match (Graph.info g n).instr with
    | Oper (_,_,_,Some labs) -> (* Saut *)
       List.iter
         (fun lab ->
           let jump_to =
             try Hashtbl.find lab2node lab with Not_found -> assert false in
           Graph.new_edge g n jump_to)
         labs
    | _ -> (* Contrôle en séquence *)
       match rem with
       | next::_ -> Graph.new_edge g n next
       | _       -> assert false
    end ;
    mk_edges g rem
```

On posera ensuite tout simplement :

```
let mk_graph code =
  let g = Graph.create (mk_info (Oper ("", [], [] ,None))) in
  mk_nodes g code ;
  mk_edges g (Graph.nodes g) ;
  g
```

En ce qui concerne la fonction fixpoint, le plus simple est sans doute de partir des « définitions » par point fixe des ensembles In et Out.

$$\mathtt{In}\,(i) = \mathtt{Use}\,(i) \cup (\mathtt{Out}\,(i) \setminus \mathtt{Def}\,(i)) \qquad\qquad \mathtt{Out}\,(i) = \bigcup_{j \in \mathtt{Succ}\,(i)} \mathtt{In}\,(j)$$

Le calcul accéléré (section 8.1.3) vient naturellement en transformant les équations en affectations des champs mutables live_in et live_out. Il est laissé en exercice.

8.3.3 Graphe d'interférence

Le graphe d'interférence est non-orienté, il possède deux sortes d'arcs (pour les interférence et les move) et il n'y pas d'arcs d'un sommet vers lui-même. Ces différences justifient un nouveau module des graphes Sgraph (figure 8.6). On remarquera surtout que le type des graphes ('a, 'b) Sgraph.t est maintenant paramétré à la fois par le type du contenu des sommets et celui des arcs. Fort logiquement les fonctions new_edge (création des arcs) et adj (retrouver les voisins d'un sommet) prennent un argument de type 'b qui identifie la sorte d'arcs concernée.

Ainsi le type des graphes d'interférence est le suivant :

```
(* Sortes d'arcs du graphe d'interférence *)
```

Figure 8.6 – Interface du module `Sgraph` des graphes non-orientés.

```
(***************************************)
(* Graphes non-orientés                *)
(* - Les sommets contiennent des 'a    *)
(* - Les arcs sont étiquetés par des 'b *)
(***************************************)

type ('a, 'b) t
type 'a node

exception Error of string
(* Erreur, avec un message *)

(* Créer un nouveau graphe *)
val create : 'a -> ('a, 'b) t

val new_node : ('a, 'b) t -> 'a -> 'a node
(* créer un nouveau sommet ajouté graphe par effet de bord *)

val new_edge : ('a, 'b) t -> 'a node -> 'a node -> 'b ->  unit
(*
   « new_edge g n1 n2 l » , ajoute un arc étiqueté par l entre n1 et n2.
   L'arc n'est pas crée lorsque :
    - Il existe déjà,
    - ou n1=n2
*)

val exists_edge : ('a, 'b) t -> 'a node -> 'a node -> 'b -> bool
(* exists_edge g n1 n2 l, teste l'existence d'un arc entre n1 et n2 et étiquetté par l *)

val iter : ('a, 'b) t -> ('a node -> unit) -> unit
(* itérer une fonctions sur tous les sommets du graphe (ordre de création) *)

(*****************)
(* Lire le graphe *)
(*****************)
val nodes : ('a, 'b) t -> 'a node list
(* liste de tous les noeuds, (ordre de creation) *)
val info : ('a, 'b) t -> 'a node -> 'a
(* contenu d'un sommet *)
val adj : ('a, 'b) t -> 'a node -> 'b -> 'a node list
(* voisins d'un sommet *)
```

```
type ilab = Inter | Move_related

(* Contenu des noeud du graphe d'interférence *)
type interference = {
    temp : temp;                  (* un temporaire *)
(* Les champs suivants sont utiles pour l'allocation de registres *)
    mutable color : temp option ;
    mutable occurs : int ;
    mutable degree : int ;
    mutable elem : ((interference Sgraph.node) Partition.elem) option ;
  }

type igraph = (interference, ilab) Sgraph.t

val interference :  flowgraph  -> igraph
```

La fonction `interference` prend en argument le graphe de flot décoré par les durées de vie et renvoie le graphe d'interférence. Elle procède par un simple parcours des sommets du graphe de flot. Pour chaque instruction i du programme (*i.e.* chaque sommet du graphe de flot), on a deux cas.

— Si i est une instruction qui n'est pas un transfert entre temporaires, alors ajouter des arcs d'interférence entre les temporaires de `Def` (i) et ceux de `Out` $(i) \setminus$ `Def` (i).

— Si i est un transfert du temporaire s vers le temporaire d, alors ajouter des arcs d'interférence entre d et les temporaires de `Out` $(i) \setminus \{d, s\}$. En outre ajouter un arc `move` entre s et d.

8.3.4 Un détail

Le code assembleur contient en plus des temporaires et des registres d'usage général, des références aux registres spéciaux (genre `gp`, `zero`, ...). Ces registres dits speciaux sont exclus du mécanisme général d'allocation des registres par nécessité (`zero`, ...) ou par choix (`sp`, `fp`, ...). Il sont définis dans le module `Spim`.

Il y a au moins deux façons de traiter proprement les registres spéciaux. Je choisis de les exclure complètement, c'est à dire de ne calculer ni leurs durées de vie, ni leur interférences. Il convient alors de ne pas inclure de registres spéciaux dans les ensembles `Def` et `Use` du graphe de flot et de ne pas les introduire dans le graphe d'interférence. Cette solution a l'avantage de produire des graphes plus lisibles.

L'autre solution est de ne pas distinguer registres spéciaux et d'usage général, jusqu'à l'allocation de registres qui doit dans ce cas traiter différemment les deux classes de registres.

8.4 Un exemple complet

Considérons un petit programme Pseudo-Pascal.

```
function fact (n : integer) : integer;
var r : integer ;
begin
  if n <= 1 then
    fact := 1
  else begin
    r := 1 ;
    while n > 0 do begin
      r := r * n;
      n := n-1
    end ;
    fact := r
  end
end;
```

Il s'agit de la fonction `fact` qui nous a déjà servi d'exemple (cf. la figure 8.1).

Voici le graphe de flot complet de la fonction `fact`. On se limite à quatre registres, `v0`, `ra`, `a0` (argument), et `s0` (callee-save).

```
fact:                      #      <=            # $a0 $s0 $ra
    subu $sp, $sp, fact_f  #      <=            # $a0 $s0 $ra
    move $112, $ra         # $112 <= $ra        # $a0 $s0 $112
    move $113, $s0         # $113 <= $s0        # $a0 $112 $113
    move $108, $a0         # $108 <= $a0        # $108 $112 $113
    li   $114, 1           # $114 <=            # $108 $112 $113 $114
    ble  $108, $114, L12   #      <= $108 $114  # $108 $112 $113
L13:                       #      <=            # $108 $112 $113
    li   $109, 1           # $109 <=            # $108 $109 $112 $113
    b    L16               #      <=            # $108 $109 $112 $113
L15:                       #      <=            # $108 $109 $112 $113
    mul  $109, $109, $108  # $109 <= $108 $109  # $108 $109 $112 $113
    sub  $108, $108, 1     # $108 <= $108       # $108 $109 $112 $113
L16:                       #      <=            # $108 $109 $112 $113
    bgt  $108, $zero, L15  #      <= $108       # $108 $109 $112 $113
L17:                       #      <=            # $109 $112 $113
    move $107, $109        # $107 <= $109       # $107 $112 $113
    b    fact_end          #      <=            # $107 $112 $113
L12:                       #      <=            # $112 $113
    li   $107, 1           # $107 <=            # $107 $112 $113
fact_end:                  #      <=            # $107 $112 $113
    move $v0, $107         # $v0  <= $107       # $v0 $112 $113
    move $115, $112        # $115 <= $112       # $v0 $113 $115
    move $s0, $113         # $s0  <= $113       # $v0 $s0 $115
    addu $sp, $sp, fact_f  # <=                 # $v0 $s0 $115
    j    $115              # <= $v0 $s0 $115 #
```

Pour chaque instruction sont montrés les `Def`, les `Use` et les temporaires vivants en sortie. Les points notables sont :

— Les temporaires `109` (variable `r`) et `108` (paramètre formel `n`) sont vivants à travers toute la boucle (des étiquettes `L13` à `L17`). Toutefois, `108` n'est plus vivant en sortie (et donc en entrée) de l'étiquette `L17`, alors qu'il est vivant en sortie du test de boucle (instruction qui suit l'étiquette `L16`), en raison du saut possible vers le debut du corps de la boucle (étiquette `L15`).

— Le temporaire `114` a une durée de vie très brève, limitée à l'intervalle entre deux instructions.
— Les sauvegardes des callee-saves `112` (pour `ra`) et `113` (pour `s0`) sont vivantes à travers presque tout le code. C'est la dernière instruction qui fait tout l'intérêt de la restauration du callee-save `s0`, en affirmant lire ce registre.

Voici ensuite graphe d'interférence :

```
$112 <=> $v0 $107 $109 $114 $108 $113 $s0 $a0
$ra  <=>
$a0  <=> $113 $112
$s0  <=> $115 $v0 $112
$113 <=> $115 $v0 $107 $109 $114 $108 $112 $a0
$108 <=> $109 $114 $113 $112
$114 <=> $113 $112 $108
$109 <=> $113 $112 $108
$107 <=> $113 $112
$v0  <=> $s0 $115 $113 $112
$115 <=> $s0 $113 $v0
```

On remarque :
— En raison de leur longue durée de vie, les sauvegardes des callee-saves (`112` et `113`) interfèrent avec presque tous les temporaires et registres. Mais ils n'interfèrent pas avec les registres sauvegardés (`ra` et `s0`).
— De façon générale les registres machine interfèrent peu avec les vrais temporaires, car les registres machine n'apparaissent ici que dans le prologue et l'épilogue. Il n'en serait pas de même en cas d'appel de fonction dans le code.

Enfin voici le graphe des `moves` :

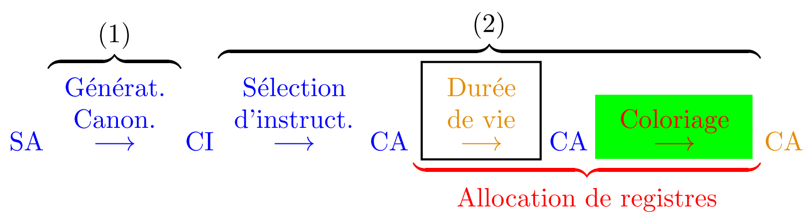
```
$112 <=> $115 $ra
$ra  <=> $115 $112
$a0  <=> $108
$s0  <=> $113
$113 <=> $s0
$108 <=> $a0
$114 <=>
$109 <=> $v0 $107
$107 <=> $v0 $109
$v0  <=> $109 $107
$115 <=> $ra $112
```

Ce graphe suggère nettement comment répartir les registres entre les temporaires. Par exemple, on aura, dans le code final, intérêt à remplacer les temporaires `109` (variable `r`) et `107` (variable `fact`) par `v0`. Le graphe d'interférence indique que c'est possible car ces temporaires n'interfèrent pas avec `v0`.

Par rapport à la définition directe ≪ il y a un arc `move` entre t et t' quand il y a un transfert entre t et t' ≫ le graphe présenté représente en fait une fermeture de la relation de transfert simple. Deux temporaires sont en relation quand il y a une chaîne de transfert entre eux, dont aucun temporaire intermédiaire n'est un registre machine. On le voit par exemple dans le cas du temporaire `109` qui est en relation avec `v0` par l'intermédiaire du temporaire `107`.

Chapitre 9

Allocation de registres



La mission de l'allocation de registres est de transformer les temporaires arbitraires du code assembleur produit par la sélection (voir le chapitre 7) en registres de la machine ciblée. Grâce à l'analyse de durée de vie, chapitre 8 il est relativement facile d'obtenir que les contenus des temporaires ne se mélangent pas, c'est à dire d'obtenir du code correct au final.

Malheureusement, il est parfois impossible de réaliser un temporaire à l'aide d'un registre machine, il faut alors réaliser le temporaire en pile, c'est à dire lui allouer non plus un registre machine, mais une case mémoire en pile. Le but d'une bonne allocation de registres est de minimiser le nombre de temporaires alloués en pile. Un but secondaire est de minimiser les transferts entre registres. En effet si le code assembleur comprend une instruction de transfert entre deux temporaires t_1 et t_2, on a alors intérêt (si c'est possible) à allouer le *même* registre à t_1 et t_2, afin d'éliminer l'instruction de transfert.

Nous sommes donc tout à la fin de chaîne de compilation. Par ailleurs le bon emploi des registres est crucial pour l'efficacité du code produit par le compilateur. Pour ces deux raisons la phase d'allocation de registres est sans doute la plus connue des phases du back-end. Et de fait, la majorité des bizarreries rencontrées lors des phases précédentes trouvent maintenant leurs explications.

Dans ce cours je vais décrire l'allocation de registre par coloriage de graphe.

9.1 Allocation d'un temporaire en pile

Avant d'entrer dans le vif du sujet, commençons par examiner pourquoi nous sommes sûrs de produire du code exécutable au final.

Considérons un bout de code quelconque encore paramétré par deux temporaires.

add t_1, t_2, 2
mul t_1, t_1, t_2

Décidons d'allouer t_1 et t_2 en pile (*to spill*. mot anglais que je vais utiliser comme synonyme de « allouer un temporaire en pile »). Il suffit d'adresser cette demande au frame de la fonction en cours de compilation (voir la section 6.3.3) à l'aide de la fonction idoine (`Frame.alloc_local`) qui alloue et renvoie une position en pile repérée par rapport au pointeur de pile. De fait nous allouons maintenant la zone des locaux du frame, voir la figure 7.10.

Donc ici, supposons que t_1 dans la première case de pile (indice 0) et t_2 dans la deuxième (indice 4). Il faut maintenant réécrire le code pour ajouter autour des opérations des lectures et écritures en mémoire dans et vers des registres dits *auxiliaires de spill.* Pour le moment, supposons que nous disposons de deux registres machine réservés à cet usage, `t8` et `t9`.

```
lw $t8, 4($sp) # load t2
add $t8, $t8, 2
sw $t8, 0($sp) # store t1
lw $t8, 0($sp) # load t1
lw $t9, 4($sp) # load t2
mul $t8, $t8, $t9
sw $t8, 0($sp) # store t1
```

Le code n'est pas très bon, mais nous nous débrouillerons, d'abord pour éviter le plus possible d'allouer en pile et ensuite pour bien choisir les temporaires spillés.

Dans notre compilateur un module spécifique `Spill` est chargé de l'allocation en pile et de la réécriture du code, son interface est donnée par la figure 9.1. En spillant tous les temporaires d'un code donné, on obtient donc un code assembleur exécutable et correct. C'est exactement ce que fait l'option `-spill` du compilateur du cours.

FIGURE 9.1 – Interface du module `Spill`, chargé de l'allocation en pile

```
val spill_fun : Ass.temp Smallset.set -> Spim.procedure -> Spim.procedure
(* « spill_fun temps proc »
   renvoie une procédure modifiée : les temporaires de temps
   sont alloués en pile. *)
```

9.2 Coloriage de graphe

Rappelons que les sommets du graphe d'interférence sont les temporaires, et que les arcs expriment que deux temporaires reliés ne peuvent pas résider au final dans le même registre.

Allouer des registres aux temporaires revient à *colorier* le graphe d'interférence, c'est à dire à assigner des couleurs aux sommet de sorte que deux sommets reliés sont de couleurs différentes. Posé dans toute sa généralité le coloriage de graphe est NP-complet. Heureusement il existe un algorithme simple (et linéaire) permettant de colorier un graphe avec K couleurs.

Soit donc un graphe G, et K couleurs. Par définition, les sommets de G se répartissent en sommets de faible et de fort degré, c'est à dire qui possèdent strictement moins de K voisins, ou K voisins ou plus. L'algorithme, connu depuis le $19^{\text{ième}}$ siècle, repose sur l'observation suivante : si le graphe G possède un sommet s de faible degré, et que le graphe $G - \{s\}$ est K-coloriable, alors G est K-coloriable. Mieux, si on a su effectivement colorier les sommets de $G - \{s\}$, alors on sait colorier s : il suffit de choisir pour s une couleur différente de celle de tous ses voisins.

On déduit facilement un algorithme récursif de l'observation. Dans une première phase de descente on retire un sommet de faible degré de graphe, cette opération pouvant changer des voisins de degré K en sommets de faible degré. Puis, dans une phase de remontée (après un appel récursif), on colorie le sommet retiré du graphe à la descente. L'algorithme proposé est incomplet, c'est à dire qu'il peut échouer alors que le graphe est K-coloriable. Ce n'est pas très gênant en pratique :

— Nous cherchons d'abord à produire, pour un coût raisonnable, du code exécutable. Or nous pouvons toujours allouer certains registres en pile.

— En pratique, l'algorithme simple légèrement amélioré se révèle suffisamment puissant. En outre cet algorithme amélioré est capable de détecter les temporaires à allouer en pile (à spiller).

9.2.1 L'algorithme de base

Pour fixer les idées, nous allons immédiatement coder l'algorithme en Caml. Il n'est ni pratique, ni efficace d'effectivement enlever des sommets (et des arcs) au graphe en cours de coloriage. À la place, nous allons plutôt répartir les sommets entre les divers sous-ensembles d'une partition des sommets. Le module `Partition` (cf. figure 9.2) fournit une réalisation impérative des partitions. Nous commençons par nous donner une partition en quatre sous-ensembles :

FIGURE 9.2 – Interface du module `Partition`.

```
(*
  Ce module fournit des opérations efficaces sur un ensemble partitionné en
  un nombre fixe de sous-ensembles.
*)

type 'a t      (* type des sous-ensembles de la partition *)
type 'a elem   (* type des éléments *)
val make : int -> 'a t array
      (* make n créer un vecteur de partitions *)
val clear : 'a t -> unit
       (* clear s efface la partition s e *)

val create : 'a t -> 'a -> 'a elem
    (* « create s e » créer un élément e dans le sous-ensemble s *)
val info : 'a elem -> 'a
    (* « info e » retourne les informations sur l'élément e *)
val belong : 'a elem -> 'a t -> bool
   (* test d'appartenance *)
val move : 'a elem -> 'a t -> unit
   (* changement de sous-ensemble *)
val pick : 'a t -> 'a elem option
   (* « pick s » renvoie un élément de s, None si s est vide *)
val pick_lowest : ('a elem -> float) -> 'a t -> 'a elem option
   (* « pick_lowest cost s » renvoie un élément de s de coût minimal, None si s est vide *)
val list : 'a t -> 'a elem list
   (* renvoie la liste des éléments d'un sous-ensemble *)
```

```
let sets = Partition.make 4

let precolored = sets.(0)
and low = sets.(1)
and high = sets.(2)
and removed = sets.(3)
```

Les sommets de G se répartiront donc entre sommets déjà coloriés (les registres machine), sommets de faible et fort degré, et sommets enlevés du graphe en attente de coloriage.

Rappelons ensuite la définition des informations associées aux sommets du graphe d'interférence.

```
(* Contenu des noeud du graphe d'interférence *)
type interference = {
    temp : temp;                     (* un temporaire *)
(* Les champs suivants sont utiles pour l'allocation de registres *)
    mutable color : temp option ;
    mutable degree : int ;
    mutable elem : ((interference Sgraph.node) Partition.elem) option ;
    mutable occurs : int ;
  }
```

Pour le moment, nous nous intéressons aux champs `temp` (le temporaire à colorier), `color` (une option de temporaire qui sera la couleur), `degree` (le degré courant du sommet), et `elem`. Supposons qu'une phase initiale non décrite ajuste les champs `color` (`None` pour un ≪ vrai ≫ temporaire et `Some` r pour un registre machine r), et `degree` (en comptant tout bêtement les voisins). Nous pouvons alors répartir initialement les sommets dans la partition :

```
(* Mach.registers est la liste des registres machines *)
let colors = Smallset.of_list Mach.registers
and ncolors = List.length Mach.registers

let ig_info ig n = Sgraph.info ig n

let build_partition ig =
  Sgraph.iter ig
   (fun n ->
     let i = ig_info ig n in
     let e = match i.color with
     | Some r -> Partition.create precolored n
     | None   ->
         if i.degree < ncolors then
           Partition.create low n
         else
           Partition.create high n in
     i.elem <- Some e)
```

Notez que pour une raison qui apparaîtra par la suite, le sommet lui-même est enregistré en tant qu'élément de la partition dans le champ `elem`. C'est un détail d'implémentation.

En effet, lorsque l'on retire un sommet du graphe, il faut aussi enlever les arcs correspondants, ce qui revient, dans notre réalisation à diminuer le champ `degree` des voisins. Il convient alors, si le degré est passé sous la barre des $K-1$ de faire passer le voisin de `high` à `low` :

```
let decr_degree ig e =           (* e est un élément de partition *)
  let n = Partition.info e in    (* sommet du graphe *)
  let i = Sgraph.info ig n in    (* informations associés au sommet *)
  if Partition.belong e low || Partition.belong e high then begin
     i.degree <- i.degree-1 ;
     if i.degree = ncolors-1 then
       Partition.move e low
  end
```

La fonction suivante `remove` enlève donc un sommet du graphe `ig` en oubliant pas de décrémenter les degrés de ses voisins.

```
let elem_of_node ig n = match info_ig ig n with
| {elem=Some e} -> e
| _             -> assert false

let remove ig e =
  let n = Partition.info e in
  let adjs = Sgraph.adj ig n Inter in
  List.iter (fun n -> decr_degree ig (elem_of_node ig n)) adjs ;
  Partition.move e removed
```

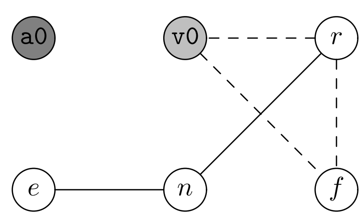
Une fois tous les sommets ≪ *enlevés* ≫, nous les colorions un par un, dans l'ordre inverse (dernier enlevé en premier). Pour ce faire nous avons besoin d'une fonction `adj_color` qui donne les couleurs des voisins d'un sommet, utilisée par une fonction `choose_color` qui choisit une couleur arbitraire parmi les couleurs qui ne sont pas des couleurs de voisins. Cette dernière fonction utilise `Smallset.choose` qui prend un ensemble en argument et renvoie un élément arbitraire de cet ensemble comme une option, ou qui renvoie `None` si l'ensemble est vide.

```
let adj_colors ig e =
  let n = Partitition.info e in
  let adjs = Sgraph.adj ig n Inter in
  Smallset.union_list
    (List.map
      (fun n -> match ig_info ig n with
      | {color=Some r} -> Smallset.singleton r
      | {color=None}   -> Smallset.empty)
      adjs)

let choose_color ig e =
  let forbiden = adj_colors ig e in
   Smallset.choose (Smallset.diff colors forbiden)
```

Notez bien que les voisins qui n'ont pas de couleurs (champ `color` à `None`) sont ceux qui ont été ≪ enlevés ≫ avant le sommet colorié et qui donc seront coloriés ensuite.

Nous sommes maintenant équipés pour colorier le graphe d'interférence. La fonction `colorize` (figure 9.3) colorie le graphe par effet de bord et renvoie un booléen qui dit si elle a pu mener sa tâche à bien. Notons que, lors de la remontée, le coloriage ne peut échouer, c'est à dire que `choose_color` renvoie toujours bien une couleur (ne renvoie jamais `None`). L'échec ne peut survenir que lors de la descente, quand il n'y a plus de sommets de faible degré et encore au moins un sommet de fort degré.

Considérons le fonctionnement de notre algorithme simple sur le graphe d'interférence de la section 8.2. En fait, le graphe colorié est un peu différent. D'une part, nous ajoutons un second sommet précolorié, `a0`, en plus de `v0`, afin de clairement montrer que le coloriage s'effectue à l'aide de deux registres. D'autre part les arcs `move` subissent la fermeture décrite à la fin du chapitre précédent, ce qui revient ici à ajouter un arc pointillé entre r et `v0`.



Allouer des registres aux quatre temporaires, revient donc ici à colorier ces quatre sommets du graphe d'interférence à l'aide des deux couleurs des sommets qui sont déjà des registres (gris foncé et gris clair). Initialement nous avons deux sommets déjà coloriés (`a0` et `v0`), trois sommets de faible

Figure 9.3 – Algorithme élémentaire de coloriage de graphe

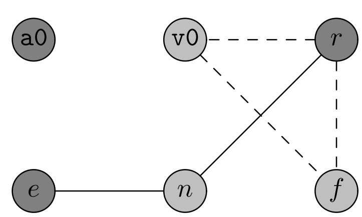
```
let put_color ig e c =
  let n = Partition.info e in
  let i = ig_info ig n in
  i.color <- c

let rec colorize ig = match Partition.pick low with
| Some e -> (* prendre un sommet de faible degré *)
    remove ig e ;                       (* « l'enlever » *)
    if colorize ig then begin           (* colorier le reste du graphe *)
      let c = choose_color ig e in      (* colorier le sommet enlevé *)
      put_color ig e c ;
      true
    end else
      false

| None -> (* low est vide *)
   match Partition.pick high with
   | Some _ -> false
   | None   -> true
```

degré (e, f et r) et un sommet de fort degré (n). La figure 9.4 décrit l'exécution des deux phases de l'algorithme. Lors de la descente la partition des sommets est montrée. Lors de la remontée, la couleur choisie est montrée. On remarquera que lors de la descente, enlever le sommet e fait passer

Figure 9.4 – Coloriage d'un graphe simple

low	high	removed
e, f, r	n	
f, n, r		e
n, r		e, f
r		e, f, n
		e, f, n, r

sommet	interdit	possible	choisie
r		a0, v0	a0
n	a0	v0	v0
f		a0, v0	v0
e	v0	a0	a0



son voisin n dans les sommets de faible degré. Au retour le choix des couleurs est arbitraire, pour r et f et forcé pour n et e. Le choix fait ici ne tient pas compte des arc `move`, une allocation plus pertinente résulterait d'un premier coloriage de r en `v0`, mais nous y reviendrons.

9.3 Choix des temporaires spillés, coloriage optimiste

Observons maintenant la situation qui mène à l'échec de l'algorithme simple : le graphe est non-vide et tous ses sommets sont de fort degré. Selon le schéma simple, nous ne pouvons plus rien faire, il convient alors de simplifier le graphe en spillant quelques temporaires et de tenter un coloriage du graphe ainsi simplifié. Toutefois, rien ne nous dit que le graphe n'est pas K-coloriable, nous pourrions très bien choisir un sommet s (représentant le temporaire t) de fort degré, ≪ l'enlever ≫ du graphe et continuer.

Dès lors, en phase de remontée du coloriage, on aura deux possibilités :

— Les $v \geq K$ voisins de s portent au total strictement moins de K couleurs distinctes, parce que ces voisins ont des couleurs identiques en nombre suffisant.

— Ou bien, il n'est pas possible de colorier s.

Dans le premier cas, s n'empêche pas en fait de colorier le graphe, dans le second, il convient d'allouer t en mémoire, c'est à dire de le spiller. En raison de la seconde possibilité, le temporaire choisi parmi les sommets de fort degré s'appelle un spill *potentiel* (*potential spill*).

On notera que si les auxiliaires de spill sont réservés (cf. section 9.1) alors on peut continuer la remontée, afin de terminer le coloriage du graphe d'interférence produit en enlevant les sommets spillés. Dans tous les cas, il convient de poursuivre la remontée afin d'identifier ceux des spills potentiels qui doivent effectivement être spillés.

Le choix du sommet s est critique et ceci pour deux raisons :

— Il faut, dans l'intérêt du coloriage, sélectionner un sommet qui interfère avec beaucoup d'autres sommets.

— Il faut, dans l'intérêt de l'efficacité finale du code, sélectionner un temporaire qui apparaît peu dans le code.

En pratique, on se donne une fonction de coût `cost` qui décroît avec le degré des sommets et croit avec le nombre d'occurences des temporaires dans le code (champ `occur` des informations du graphe d'interférence). Le moment venu, on choisit un sommet de coût minimum parmi les sommets de fort degré.

L'algorithme de coloriage optimiste est donné par la figure 9.5. Par rapport au coloriage simple de la figure 9.3, on note l'apparition de deux nouveaux sous-ensembles dans la partition des sommets du graphe d'interférence : `colored` pour les sommets finalement coloriés, et `spilled` pour les autres. À la remontée, on répartit les sommets de `removed` dans l'un ou l'autre de ces sous-ensembles, selon que leur coloriage est possible ou pas. On notera que le code comporte une astuce, les spills potentiels ne sont pas directement enlevés, mais bougés du sous-ensemble `high` vers `low`. L'astuce permet de n'insérer le code de coloriage qu'après le premier appel récursif à `colorize`.

Au retour de `colorize` on détectera la réussite ou l'échec en regardant si `spilled` est vide ou pas. Dans le premier cas on se livrera à un dernier passage sur le code (remplacer les temporaires par leur couleur, émettre la définition de la constante symbolique correspondant à la taille du frame). Dans le second cas, il faut réécrire le code à l'aide de `spill_fun` (figure 9.1) et tout recommencer.

En effet, notre compilateur alloue des temporaires frais comme auxiliaires de spill, temporaires qui peuvent interférer avec d'autres, ce qui commande de reconstruire un graphe d'interférence et de recommencer l'allocation de registres. Mais spiller un temporaire revient à transformer le graphe d'interférence en un autre ≪ plus simple ≫. En effet les temporaires créés comme auxiliaires de spill ont une durée de vie très courte, on les appelle des *éphémères.* Par conséquence, le temporaire spillé est remplacé par une multitude de temporaires éphémères et les interférences diminuent. Par exemple, considérons l'exemple de la section 9.1 en utilisant cette fois des temporaires éphémères.

lw e_1, `4($sp)` # *load* t_2
add e_2, e_1, `2`
sw e_2, `0($sp)` # *store* t_1
lw e_3, `0($sp)` # *load* t_1
lw e_4, `4($sp)` # *load* t_2
mul e_5, e_3, e_4
sw e_5, `0($sp)` # *store* t_1

Si les temporaires t_1 et t_2 interféraient avec un troisième t_3 en raison par exemple du code qui suit notre exemple, le temporaire t_3 est maintenant moins contraint, puisqu'il ne peut interférerer

Figure 9.5 – Coloriage optimiste

```
let sets = Partition.make 6
let precolored = sets.(0)
...

and colored = sets.(4)
and spilled = sets.(5)

let cost ig e = ...

let select_spill ig = Partition.pick_lowest (cost ig) high

let rec colorize ig = match Partition.pick low with
| Some e -> (* prendre un sommet de faible degré *)
    remove ig e ;                       (* « l'enlever » *)
    colorize ig ;                       (* colorier le reste du graphe *)
    begin choose_color ig e with         (* choisir une couleur *)
    | Some c ->  (* colorier *)
        put_color ig e (Some c) ; Partition.move e colored
    | None   ->  (* spiller *)
        Partition.move e spilled
    end

| None -> (* low est vide *)
   match select_spill ig with (* selectionner un spill potentiel *)
   | Some e -> Partition.move e low ; colorize ig  (* « l'enlever », continuer *)
   | None   -> ()                                   (* graphe vide, c'est fini  *)
```

avec les éphémères e_1 à e_5.

En pratique, on itère donc l'algorithme de coloriage de graphe sur des graphes d'interférence de plus en plus simples. L'expérience montre que le coloriage s'effectue après éventuellement une, plus rarement deux tentatives infructeuses, *à condition de ne pas spiller les auxiliaires de spill.* En effet :

— Il est clairement inutile de spiller un auxiliaire de spill.

— Il est dangereux de le faire, car alors le coloriage pourrait boucler.

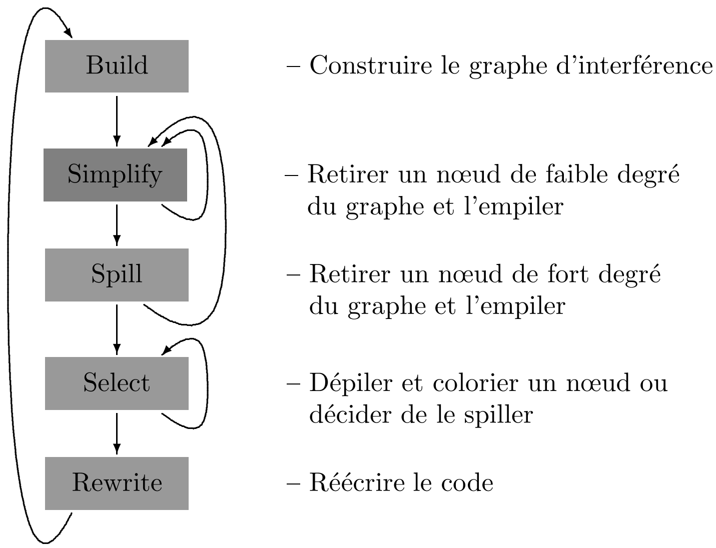
En pratique on évite simplement le spill des auxiliaires de spill en dotant les éphémères alloués par `spill_fun` d'un coût exhorbitant. En notant que les éphémères sont créés à l'aide d'une fonction idoine (`Gen.ephemere` au lieu de `Gen.new_temp`) et que l'on peut savoir si un temporaire est un éphémère (par la fonction `Gen.is_ephemere` de type `Gen.temp -> bool`). Nous pouvons essayer cette fonction de coût pour le choix des spills potentiels parmi les sommets de fort degré :

```
let cost ig e =
  let n = Partition.info e in
  let i = Sgraph.info ig n in
  (float i.occurs) /. (float i.degree) +.
  (if Gen.is_ephemere i.temp then 100.0 else 0.0)
```

La fonction `cost` est bien croissante avec le nombre d'occurences, décroissante avec le degré et de coût exhorbitant pour les éphémères. Cette fonction peut être améliorée par des essais. En outre, si on dispose d'informations précises sur le contrôle du programme, on dotera les temporaires qui apparaissent dans les boucles d'un coût relativement élevé.

L'algorithme complet de coloriage peut être représenté graphiquement (figure 9.6). Dans cette

Figure 9.6 – Représentation graphique de l'algorithme de coloriage optimiste



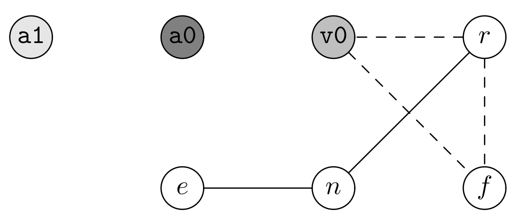
représentation, la pile des sommets est explicitée, alors que dans le code récursif de la figure 9.5 cette pile était implicite. Cette pile est remplie lors de la descente et vidée lors de la remontée.

9.4 Bon choix des couleurs, coloriage biaisé

Jusqu'ici nous avons superbement ignoré les arcs `move`, ce qui fait que nous choisissons les couleurs un peu au hasard. Ainsi, la figure 9.7 donne (à droite) le code final produit pour l'exemple de la figure 8.1 (rappelé à gauche) à partir de l'allocation des couleurs obtenue à la figure 9.4. Le dernier transfert entre registres (de `v0` dans `v0`) est inutile et peut être enlevé, mais il reste encore un transfert (à l'étiquette `L17`).

En revenant sur le déroulement de l'algorithme (figure 9.4) nous voyons que l'attribution arbitraire du registre `a0` au temporaire r est maladroite. En effet, les arcs `move` relient r f et `v0`, de sorte que l'attribution de `v0` est désirable. Plus généralement nous pouvons definir les couleurs *désirables* d'un temporaire t comme celles des temporaires voisins de t selon les arcs `move`, et choisir pour t une couleur désirable quand c'est possible. Si, lors de la remontée nous choisisons des registres désirables, nous obtiendrons ici la meilleure allocation des registres possible, puisque tous les transferts entre registres disparaissent, comme indiqué par la figure 9.8.

Mais ce n'est pas tout, donnons nous maintenant trois registres `v0`, `a0` et `a1`. Le graphe d'interférence devient.



Tous les sommets du graphe d'interférence sont maintenant de faible degré, et le coloriage peut très bien commencer par exemple par le temporaire n. Ce temporaire ne possède pas de couleur désirable, puisqu'il n'a pas de voisins selon les arcs `move`. Pourtant, choisir pour lui le registre `v0` est maladroit puisque nous rendons alors impossible l'attribution de la couleur `v0` à r. Nous pouvons tenir compte de cet effet en définissant les couleurs *indésirables* d'un temporaire comme les couleurs désirables des temporaires qui interfèrent avec lui. Le choix des couleurs s'opère ensuite selon ce schéma :

Figure 9.7 – Choix « arbitraire » des registres

```
    li  e, 1
    ble n, e, L12
L13:
    li  r, 1
    b L16
L15:
    mul r, r, n
    sub n, n, 1
L16:
    bgt n, $zero, L15
L17:
    move f, r
    b fact_end
L12:
    li  f, 1
fact_end:
    move $v0, f
```

```
    li  $a0, 1
    ble $v0, $a0, L12
L13:
    li  $a0, 1
    b L16
L15:
    mul $a0, $a0, $v0
    sub $v0, $v0, 1
L16:
    bgt $v0, $zero, L15
L17:
    move $v0, $a0
    b fact_end
L12:
    li  $v0, 1
fact_end:
    move $v0, $v0
```

Figure 9.8 – Coloriage en suivant les couleurs désirables

sommet	interdit	desirable	choisie
r		v0	v0
n	v0		a0
f		v0	v0
e	a0		v0

```
    li  $v0, 1
    ble $a0, $v0, L12
L13:
    li  $v0, 1
    b L16
L15:
    mul $v0, $v0, $a0
    sub $a0, $a0, 1
L16:
    bgt $a0, $zero, L15
L17:
#   move $v0, $v0
    b fact_end
L12:
    li  $v0, 1
fact_end:
#   move $v0, $v0
```

1. Les couleurs *possibles* sont celles qui ne sont pas des couleurs de voisins selon les arcs d'interférence.
2. Tenter de donner une couleur possible et désirable.
3. Si aucune couleur désirable n'est possible, éviter de donner une couleur indésirable.
4. Si toutes les couleurs possibles sont indésirables, donner une couleur possible arbitraire.

En pratique on transforme facilement le colorieur de la figure 9.5 en un colorieur biaisé en changeant seulement la fonction `choose_color`. Ce petit codage est laissé en exercice.