Principes des langages de programmation
IN213

Michel Mauny

Inria-Paris
2 rue Simone Iff
CS 42112
F-75589 Paris Cedex 12
prénom.nom@inria.fr
http://www.mauny.net/

François Pessaux

ENSTA ParisTech
828 boulevard des Maréchaux
91120 Palaiseau
prenom.nom@ensta.fr
http://perso.ensta-paristech.fr/~pessaux/

Édition 2023-2024 version 2.1.8

Un ingénieur, quelque soit son domaine d'intervention, doit maîtriser les concepts sous-jacents aux langages de programmation, ceux qui marquent les différences profondes entre les langages actuels, passés et futurs, ainsi que les différentes notions relatives au sens et à la correction des programmes. Cette culture informatique lui sera bien sûr nécessaire s'il réalise, dirige, ou contribue à diriger des développements informatiques, mais elle le sera aussi lorsqu'il aura à évaluer des éléments logiciels, ou à participer à des prises de décision stratégiques impliquant des choix en matière de logiciel.

Ce cours décrit les concepts fondamentaux des langages de programmation, en commençant par en donner une perspective historique, puis en donnant les outils permettant de décrire précisément le sens des programmes informatiques, partant de leur syntaxe jusqu'à leur sémantique, en passant par une revue de différentes techniques de mise en œuvre : interprètes ou compilateurs.

Une part importante du cours sera laissée à l'expérimentation : on utilisera en effet le langage OCaml pour mettre en œuvre différents aspects d'un langage de programmation, le tout formant un mini-projet.

Note : ce document peut contenir des erreurs, typographiques ou autres, ou bien des imprécisions. Si, lors de votre lecture attentive, vous avez des commentaires ou remarques, même mineures, qui sont à même d'améliorer ce document, je vous serais très reconnaissant de faire l'effort me les communiquer. Vous pourrez le faire oralement, à l'occasion d'un cours, ou bien par courrier électronique à `prenom.nom@ensta.fr`, en remplaçant `prenom` et `nom` par les valeurs idoines [1].

Remerciements : merci aux différents enseignants auxquels j'ai emprunté quelques éléments, ainsi qu'à Maurice Diamantini pour sa relecture attentive. Je voudrais exprimer ici ma reconnaissance toute particulière envers François Pessaux qui a généreusement et spontanément accepté de dispenser ce cours à ma place en 2011-2012, et qui, à l'occasion, y a apporté de nombreuses améliorations.

Michel, Paris, mars 2013.

Remerciements : je remercie chaleureusement Michel de m'avoir transféré tout le matériel de ce cours, le polycopié, les exercices, les diapositives, afin que je puisse perpétrer l'enseignement du cours « Principes des langages de programmation » à l'ENSTA Paris. Michel a fait un travail remarquable en construisant ce cours et je compte contribuer à l'enrichir.

François, Palaiseau, janvier 2019.

1. Note François PESSAUX : sans cédille

Introduction

Les premiers ordinateurs, dans les années 1940, occupaient des salles entières, chacun d'entre eux consommait une puissance électrique confortable et coûtait des millions de dollars de l'époque, pour fournir une puissance de calcul plus faible que le moindre assistant numérique ou téléphone cellulaire qui équipe la plupart d'entre nous en ce début de 21ème siècle. À cette époque, le temps de calcul avait un coût comparable à celui du temps de travail humain, et les ordinateurs étaient programmés directement en langage machine. Par exemple, le programme de la figure 1 calcule le PGCD de deux entiers en utilisant l'algorithme d'Euclide sur un processeur MIPS R4000.

```
27bdffd0 afbf0014 0c1002a8 0c1002a8 afa200lc 8fa400lc 00401825
10820008 0064082a 10200003 10000002 00832023 00641823 1483fffa
0064082a 0c1002b2 8fbf0014 27bd0020 03e00008 00001025
```

FIGURE 1 – Le code machine MIPS R4000 du calcul du PGCD

Pour écrire des programmes plus importants, on a rapidement eu besoin d'une notation de plus haut niveau et on a inventé pour cela les langages d'assemblage, dont les instructions sont composées d'opérations exprimées par des *mnémoniques* comme `move` ou `beq`[2], et des adresses (opérandes) sur lesquelles ces opérations vont agir. Le code machine de la figure 1 ci-dessus s'exprime en langage d'assemblage MIPS par le programme donné à la figure 2.

Chaque nouveau processeur apportait avec lui un nouveau langage d'assemblage dans lequel les programmes se traduisaient ligne à ligne en code machine, à l'exception de quelques *macros* – abréviations paramétrées – qui permettaient d'écrire de façon concise des séquences d'instructions fréquentes en en abstrayant les opérandes.

Il devenait pénible de traduire chaque programme vers un nouveau langage d'assemblage pour l'exécuter sur un nouveau processeur, d'autant plus que le nombre de programmes et leur taille ne cessaient de croître. Le besoin de langages de programmation un peu moins proches de la machine devenait impératif (si j'ose dire), notamment pour effectuer des calculs arithmétiques qu'il est très tentant de noter par les formules mathématiques habituelles au lieu des instructions des langages d'assemblage. C'est ainsi que vers le milieu des années 1950 est apparue la première version du langage Fortran[3] dédié aux calculs numériques qui formaient à l'époque l'immense majorité des calculs effectués par les ordinateurs. D'autres langages de haut niveau sont ensuite rapidement apparus comme Lisp et Algol, et chacun de ces langages a laissé une empreinte très forte sur les langages de programmation que l'on emploie de nos jours.

Ces langages permettent donc de décrire des calculs, mais sans mentionner directement les ressources particulières de la machine tels le nombre de registres ou les modes d'adressage disponibles. Ils sont traduits en langage machine par un *compilateur*, ou interprétés par un programme appelé *interprète* comme cela a longtemps été le cas des langages Lisp. Les premiers compilateurs ne produisaient pas

2. *Branch if equal.*
3. Pour « FORmula TRANslator ».

```
    addiu   sp,sp,-32
    sw      ra,20(sp)
    jal     getint
    jal     getint
    sw      v0,28(sp)
    lw      a0,28(sp)
    move    v1,v0
    beq     a0,v0,D
    slt     at,v1,a0
A:  beq     at,zero,B
    b       C
    subu    a0,a0,v1
B:  subu    v1,v1,a0
C:  bne     a0,v1,a
    slt     at,v1,a0
D:  jal     putint
    lw      ra,20(sp)
    addiu   sp,sp,32
    jr      ra
    move    v0,zero
```

FIGURE 2 – Le code assembleur MIPS R4000 du calcul du PGCD

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

int pgcd(int i, int j) {
   if (i == j) return i;
   else if (i > j) return pgcd(i-j, j);
   else return pgcd(i, j-i);
}

int main(int argc, char **argv) {
   int n = atoi(argv[1]);
   int m = atoi(argv[2]);
   printf("%d\n", pgcd(n,m));
   exit(0);
}
```

FIGURE 3 – Le programme `pgcd` en langage C

nécessairement un très bon code et les programmeurs un peu expérimentés faisaient très souvent mieux. L'amélioration des techniques de compilation et d'optimisation d'une part et la complexité des architectures de processeurs d'autre part ont rapidement inversé cette tendance, et il est très rare de nos jours d'avoir recours au langage d'assemblage de la machine pour écrire un programme : un bon compilateur fera généralement mieux, et – bien sûr – de façon beaucoup plus fiable. La qualité du code produit est certes un argument majeur en faveur de ces langages de programmation, mais

```
let rec pgcd(i, j) =
  if i = j then i
  else if i > j then pgcd(i-j, j)
  else pgcd(i, j-i)
;;

let main () =
  let (a,b) =
    (int_of_string(Sys.argv.(1)),
     int_of_string(Sys.argv.(2)))
  in
  Printf.printf "%i\n" (pgcd(a,b));
  exit 0
;;

main();;
```

FIGURE 4 – Le programme pgcd en langage OCaml

parmi les éléments quantifiables qui militent en faveur de l'utilisation de langages de programmation de haut niveau, c'est bien sûr le coût de développement et de maintenance, c'est-à-dire le coût du travail humain, qui prédomine. La *portabilité*, c'est-à-dire la possibilité d'exécuter un programme sur des architectures différentes sans avoir à modifier son code source, juste en le re-compilant, a permis un premier gain important en main d'œuvre. La rigueur de développement logiciel apportée par la *programmation structurée*, les différentes techniques algorithmiques, la constitution de bibliothèques de code réutilisable, les analyses effectuées à la compilation (telle la vérification de types), l'expressivité que confèrent les objets ou les techniques de programmation dites *déclaratives*, sont autant d'aspects qui ont été (et sont encore) développés avec un but essentiel quasi-unique : améliorer la productivité et la sécurité de développement des programmes, ainsi que la facilité de leur réutilisation et de leur maintenance.

Chapitre 1

Les langages de programmation

Même s'il ne nous vient à l'esprit que quelques noms de langages de programmation quand on y réfléchit rapidement, des milliers de langages de programmation ont été créés depuis les langages « pionniers » qu'ont été les trois langages mentionnés en introduction, et l'on continue d'en créer très régulièrement. Les raisons de cette prolifération sont multiples. Essayons d'en énumérer quelques-unes.

L'évolution des techniques. L'informatique est une discipline jeune en perpétuelle évolution où de nouvelles façons de procéder apparaissent, faisant suite à des efforts de recherche ou comme fruits de l'expérience ou encore grâce aux évolutions des technologies matérielles. La programmation structurée apparut vers le début des années 1970, remplaçant le **goto** par des structures de contrôle ne violant pas la structure de bloc des programmes comme la boucle **while** ou les sélections de cas **case** ou **switch**. Plus tard, au début des années 1980, des concepts introduits dans le langage Simula à la fin des années 1960 ont refait surface et popularisé une nouvelle façon de voir les programmes : au lieu de voir un programme comme un *calcul* utilisant des structures de données, les programmes sont apparus comme des mises en scène d'*objets* dotés de capacités de calcul partagées par les membres d'une même famille ou *classe*.

Les spécificités des domaines d'application. De nombreux langages ont été conçus pour être utilisés dans un domaine bien précis. Les langages de la famille Lisp ont excellé dans la manipulation de données symboliques complexes. Snobol et Icon ont été conçus pour manipuler des chaînes de caractères. Le langage C est très bien adapté à la programmation système. Le langage Prolog a été conçu pour effectuer des déductions par inférence. Les différents *shells* d'Unix puis les langages Perl, Python, *etc.* ont permis de développer rapidement des outils plus ou moins complexes dans le domaine du traitement des fichiers systèmes, des outils d'installation de logiciel, voire dans celui des applications Internet (web). Le langage Javascript connaît un succès très important grâce aux outils de communication que sont les navigateurs Internet et les clients de messagerie électronique.

Chacun de ces langages peut bien sûr être utilisé pour effectuer des tâches quelconques, il n'en reste pas moins que le langage se révèlera particulièrement bien adapté à certains contextes, dont ceux pour lesquels il a été initialement conçu, mais pas à tous.

Les préférences individuelles. Certains préfèrent penser « récursivement », d'autres « itérativement » ; certains aiment la concision de C, d'autres lui préfèrent la stucture marquée de mots-clés d'Ada ou de Pascal. La diversité des individus contribue elle aussi, dans une certaine mesure, à une certaine variété des langages.

Bien sûr, parmi tous ces langages, certains sont massivement utilisés et d'autres sont probablement condamnés à rester confidentiels à tout jamais. Quelles sont les raisons possibles du succès d'un langage ? Là encore, il y a plusieurs éléments de réponse.

La puissance d'expression. C'est un argument assez courant qui tente de refléter la facilité d'exprimer tel ou tel concept, algorithme ou calcul dans un langage donné. Certes, tous les langages sont – pour la plupart – *théoriquement* équivalents : ils permettent tous d'exprimer n'importe quelle fonction calculable, et donc tout algorithme. Ainsi, il est possible de traduire tout programme écrit dans un langage donné en un programme écrit dans un autre langage : le résultat sera simplement plus ou moins élégant. La puissance d'expression d'un langage donné vise à mesurer la facilité d'écriture de programmes clairs, concis, faciles à relire et à maintenir, et ce, tout spécialement lorsqu'il s'agit de très gros programmes.

La facilité d'usage. Basic et Logo sont deux langages dont le succès est certainement dû à la simplicité de leurs concepts et à la rapidité avec laquelle on se familiarise avec eux. C'est aussi la raison pour laquelle le langage Pascal a longtemps été utilisé comme langage d'enseignement de la programmation. Java et Python sont probablement en train de prendre cette place pour la même raison. Cette facilité d'usage est souvent caractérisée en anglais par la notion de *learning curve*. Cette expression provient d'une constatation effectuée dans les années 1930 dans l'industrie aéronautique aux USA : à chaque fois que la production d'appareils doublait, le temps nécessaire à la fabrication de l'un d'entre eux diminuait [1] de 15%. Cette expression, utilisée dans le contexte de langages de programmation indique la vitesse avec laquelle on devient tellement familier avec le langage qu'il ne reste du coût de la conception des programmes que celui de l'élaboration de la solution proprement dite et le temps de saisie du programme. Pensez par exemple à l'aisance avec laquelle on s'exprime en langage naturel ou celle que l'on atteint lorsqu'on est habitué à utiliser un logiciel comme un traitement de texte, un système de fenêtrage ou un système d'exploitation. Selon les domaines, les progrès (économies de temps, ou de coût en général) varient entre 5 et 30% jusqu'à atteindre ce coût constant incompressible.

On dit que la courbe d'apprentissage est haute (*high learning curve*) lorsque les progrès sont faibles, et basse (*low learning curve*) lorsque les progrès sont importants. Les langages de programmation sont inégaux de ce point de vue : certains sont plus complexes que d'autres, ou induisent des façons de penser plus ou moins naturelles. Cela dit, je n'ai pas connaissance d'études comparatives de cette courbe d'apprentissage dans le domaine des langages de programmation.

La facilité d'implémentation. Il est notoire que le succès de Pascal [2] a été largement dû à sa disponibilité sur une grande variété d'architectures, en particulier sur les micro-ordinateurs pauvres (à l'époque) en ressources comme la mémoire vive et la puissance de calcul. L'implémentation de Pascal réalisée par Niklaus Wirth était simple et portable : elle utilisait une sorte de *bytecode* [3] appelé le *p-code* et il suffisait de disposer d'un programme d'interprétation pour avoir une implémentation complète du langage. Les langages Java et C# utilisent des technologies similaires avec la machine virtuelle Java (JVM) et le CLR (*Common Language Runtime*) de la plateforme .NET de Microsoft.

La disponibilité de compilateurs efficaces. Le succès et la pérennité du langage Fortran sont pour partie dus à la qualité de ses compilateurs. Bien sûr, l'investissement énorme en temps et en argent effectué par les réalisateurs de compilateurs Fortran est une raison importante de la qualité de ces compilateurs, mais il faut dire aussi que les versions de Fortran antérieures aux années 1990 étaient dépourvues de récursion et de pointeurs et fournissaient ainsi un contexte très favorable à la production de très bon code machine pour les programmes correspondants.

Les facteurs non techniques. Le fait qu'IBM ait soutenu les langages Cobol et PL/1 explique grandement leur succès. Ada doit beaucoup au département de la défense américain (DoD), et le langage Java, soutenu par la société Sun Microsystems, est rapidement devenu populaire grâce notamment au battage médiatique organisé par cette même compagnie. Le langage C#, proposé par Microsoft, s'apprête probablement à devenir lui aussi un langage important, puisqu'il est central à la plateforme .NET

1. Avec très certainement un coût asymptotique constant et non nul.
2. Dans une moindre mesure, le succès de Caml-Light dans l'enseignement a eu les mêmes raisons.
3. Code symbolique représenté par une suite d'octets, et exécutés par un programme appelé « interprète de *bytecode* ».

disponible aussi bien sous MS-Windows que sous Linux. Enfin, l'existence d'une base importante de programmes et/ou de bibliothèques déjà écrite dans un langage freine l'adoption de nouveaux langages, même si ces nouveaux langages sont considérés comme « techniquement meilleurs » que les anciens. Il s'agit là de facteurs économiques et sociaux, qui peuvent aller dans le même sens, ou à contresens, des arguments techniques classiques. Effectivement, réécrire une telle base de code est à la fois coûteux et dangereux, le risque d'introduire de nouveaux bugs au cours de la réécriture (du fait de différences sémantique entre les langages ou simplement à cause d'erreurs d'inattention) étant très grand. Ainsi, Cobol est toujours utilisé dans le domaine bancaire car de nombreux millions de lignes de code écrites dans ce langage existent et continuent à tourner depuis des décennies au sein des systèmes bancaires.

Ces différents arguments, même ceux qui sont d'ordre technique, sont très souvent contradictoires, et lorsqu'il est question de concevoir, de mettre en œuvre, ou simplement de choisir un langage de programmation, on est naturellement amené à faire des compromis comme renoncer à telle ou telle caractéristique trop coûteuse ou trop complexe, ou à choisir tel ou tel langage afin d'optimiser des critères tels l'existence d'une base importante de programmeurs ou de programmes (le langage est populaire), ou, au contraire, l'expressivité d'un langage dont la popularité n'est pas encore établie. Le point de vue du programmeur n'est plus nécessairement prédominant comme il l'a été aux premiers temps de la programmation. Le choix des outils de développement est aussi une affaire de décideurs qui vont naturellement chercher à priviligier des notions de coût, là où le programmeur va aussi prendre en compte son goût personnel et probablement privilégier une forme d'efficacité. Le décideur ne va pas complètement ignorer cette efficacité, et ne va pas non plus se limiter à prendre en compte le coût de développement. Il va probablement inclure le coût de la maintenance et de l'évolution du produit, et là vont intervenir des critères techniques comme l'expressivité du langage (la clarté et la lisibilité des programmes) ainsi que des critères économiques comme sa pérennité probable ou les avantages en terme d'image que procurerait l'utilisation de tel ou tel langage.

1.1 Familles de langages

On peut cartographier les langages de programmation de plusieurs façons. Les plus communes consistent à d'une part les ranger par grandes familles composées de langages utilisant les concepts voisins, et d'autre part à élaborer une sorte de généalogie formant une perspective historique purement temporelle avec des influences fortes des ancêtres sur les descendants.

Pour ce qui est de la classification en grandes familles, la division la plus commune consiste à composer deux groupes : les langages dits *impératifs* et les langages *déclaratifs*. La famille des langages dits **déclaratifs** est composée de langages dont les programmes ont tendance à indiquer quel est le problème à résoudre (en un sens, *déclarer* le problème), que *comment* le résoudre. Ces langages sont généralement de haut niveau au sens où les concepts qu'ils proposent sont loin des détails d'implémentation. Dans cette famille, les langages les plus déclaratifs ne demandent même pas au programmeur de donner un algorithme (qui relève du *comment*). D'autres restent des langages algorithmiques, et sont, en ce sens, proches de la frontière – très floue – qui les sépare de l'autre grande famille de langages. Les langages dits **impératifs** proposent quant à eux un modèle de calcul assez proche du fonctionnement effectif de la machine. Un programme écrit dans un langage impératif est généralement constitué d'*instructions* (d'où le terme *impératif*) dont l'exécution va modifier la mémoire (des variables ou des champs de blocs mémoire qui représentent des structures de données). Le modèle de calcul impératif est aussi appelé *modèle de von Neumann*, du nom du mathématicien John von Neumann (1903–1957) qui a en 1945 décrit une architecture d'ordinateurs où le programme était stocké dans la mémoire de l'ordinateur au même titre que les données manipulées par le programme. (Auparavant, les « ordinateurs » étaient à programme fixe, un peu comme l'est, en principe, une calculatrice arithmétique de base.) Pour la petite histoire, on raconte[4] d'ailleurs que cette invention n'est pas due à von Neumann seul, mais aussi à certains de ses collaborateurs, voire même que d'autres auraient eu cette idée avant lui.

4. http://en.wikipedia.org/wiki/Von_Neumann_architecture

Chacune de ces deux grandes familles peut se décliner en sous-familles de langages assez proches les uns des autres, comme décrit à la figure 1.1.

langages déclaratifs	
fonctionnels	Lisp/Scheme, ML/OCaml, Haskell, ...
à flots de données	Id, Val, ...
logiques, à contraintes	Prolog, VisiCalc, ...
langages impératifs	
classiques (von Neumann)	Fortran, Pascal, Basic, C, ...
à objets	Smalltalk, Eiffel, C++, Java, C#, ...

FIGURE 1.1 – Une classification des langages

Les langages fonctionnels sont essentiellement des langages d'expressions (par opposition aux langages d'instructions) où les calculs sont des évaluations d'appels de fonctions. Certains d'entre eux sont dits « purs », au sens où ils ne disposent pas du tout d'opérations de modification de variables ou de modification de la mémoire. C'est le cas du langage Haskell, par exemple. D'autres, comme Scheme ou OCaml, disposent de traits impératifs que l'on peut utiliser pour adopter un style de programmation plus classique. Ces langages privilégient toutefois la notion d'expression, de définition et d'appel de fonction. On notera que, de ce fait, on peut discuter du bien fondé du classement de ces langages fonctionnels « impurs » dans la catégorie des langages déclaratifs : ce sont en effet des langages où l'expression des programmes est plus algorithmique que purement déclarative.

Les langages à flots de données (*dataflow languages*) modélisent les calculs comme la traversée d'un graphe dirigé par des données qui sont transformées à chaque nœud du graphe. Ce modèle de calcul est inhéremment parallèle, mais est aussi lié à la programmation fonctionnelle en ce que l'enchaînement des nœuds peut être vu comme une composition de fonctions, si on voit chaque nœud comme une fonction. Les langages Id, Val, Sisal (langages fonctionnels à affectation unique), SAC (*Single-Assignement C*), Lustre, Lucid Synchrone sont des exemples de langages à flots de données.

Les langages logiques ou à contraintes sont probablement les meilleurs exemples de langages déclaratifs en ce que la rédaction d'un programme consiste réellement en l'exposition du problème : les faits connus, les schémas de déduction possibles et la question posée dans le cas de la programmation logique, les contraintes à satisfaire et le problème à résoudre dans le cas des langages à contraintes. Le mécanisme d'exécution des programmes peut alors se schématiser comme un mécanisme de résolution qui va chercher à répondre à la question posée dans chacun de ces deux cas. Dans ces langages, on n'écrit pas d'algorithme : c'est le langage qui fournit une sorte d'algorithme universel. Il est toutefois bien utile de comprendre finement le mécanisme de recherche de solutions fourni par le langage (ou son implémentation) pour adopter un style de description qui lui soit bien adapté. Le langage logique le plus connu est le langage Prolog, conçu en France, et qui a connu son heure de gloire dans les années 1980 après qu'il ait été mentionné dans un grand projet japonais de conception d'ordinateurs dits « de cinquième génération ». L'heure est plutôt aux langages de résolution de contraintes, et il se trouve que les langages du style Prolog s'enrichissent assez naturellement de mécanismes de résolution de contraintes. Plus proches de nous, les feuilles de calcul du style VisiCalc ou Excel intègrent bien souvent des solveurs de contraintes.

Les langages impératifs classiques de la famille dite « von Neumann » sont ceux qui ont eu le plus grand succès grâce à la proximité des concepts qu'ils offrent avec l'architecture des machines sur lesquels les programmes s'exécutent. Ils donnent accès à la mémoire sous la forme de variables et de structures

de données : la mémoire constitue à chaque point du programme un état global ou local modifiable. Les programmes des langages impératifs sont constitués d'instructions qui vont essentiellement modifier la mémoire. Ces modifications de l'état ou de l'environnement du programme (opérations de lecture/écriture) sont appelés *effets de bord*, c'est-à-dire des modifications d'état effectuées par un appel de fonction ou de procédure et qui ne sont pas visibles dans résultat (valeur de retour) de ces appels. Il s'agit plus d'effets que l'on pourrait de nos jours de qualifier de *collatéraux* [5] de l'exécution d'un morceau de programme. Les effets de bord sont appelés *side effects* en anglais.

Les langages à objets ont bénéficié d'une popularité récente, relativement aux langages impératifs classiques, même si on peut faire remonter leur origine au langage Simula conçu avant 1970. Les langages à objets offrent une organisation des programmes radicalement différente de la vision classique. Alors que cette dernière considère le programme comme un *calcul* opérant sur des structures de données, et l'architecture d'un programme classique visant à obtenir une structuration claire de ce calcul, la vision objet intègre les calculs (fonctions et procédures) dans les structures de données elles-même qui sont alors appelées *objets*. Dans les langages à objets majeurs, les objets sont organisés en familles appelées *classes*. Les calculs associés aux objets sont vus comme des attributs appelés *méthodes*.

Ainsi, un programme à objets reste bien sûr un calcul, mais l'organisation de ce calcul est bien différente de l'organisation classique. Les objets ont chacun leur *état*, sorte de mémoire privée modifiable, et le modèle de calcul reste généralement un modèle impératif. Le langage objet le plus pur est le langage Smalltalk, dans lequel toute donnée est un objet. C++ est probablement le plus utilisé. Les langages majeurs récemment apparus que sont Java et C# sont tous les langages à objets, ainsi que les langages de script que sont Python ou Ruby. Les langages fonctionnels peuvent eux aussi offrir un sous-système à objets comme Common Lisp (CLOS : *Common Lisp Object System*), ou OCaml, mais dans un modèle de calcul qui garde des aspects potentiellement impératifs.

1.2 Perspective historique

La figure 1.2 donne une perspective historique et approximative de l'histoire des langages de programmation. Cette perspective est partielle : elle souffre très probablement d'un certain nombre d'oublis volontaires ou non. Les dates données dans ce schéma sont elles aussi souvent discutables : elles sont en effet difficiles à fixer précisément, car il peut s'écouler plusieurs années entre la présentation d'un langage et sa mise en œuvre effective. Notons aussi que certains langages n'ont jamais existé que sur papier tout en ayant une influence notable sur le déroulement de l'histoire. Les flèches représentent des influences importantes entre langages.

1.3 Objectifs du cours

Sans contester l'importance des autres thèmes ou disciplines de l'informatique, les langages de programmation restent, avec ce qui touche à la conception des machines elles-mêmes, au centre de l'informatique. Un peu comme les conducteurs de voitures peuvent s'intéresser à ce qui se passe sous le capot de leurs engins, et en tirer ainsi le meilleur parti, l'un des intérêts de l'étude des langages de programmation est de comprendre ou de prédire de manière plus ou moins fine ce qui se passe lors de l'exécution des programmes.

Mieux comprendre les langages permet de comprendre leurs forces ou faiblesses, de savoir que tel langage est bien adapté aux calculs numériques, que tel type de construction ou telle caractéristique est précieuse pour construire des programmes de grande taille, voire qu'il est préférable d'éviter l'usage de tel ou tel trait de certains langages parce que son sens est mal défini ou qu'il conduit très souvent à une

5. Chaque épisode guerrier de notre ère sur-médiatisée laisse ses éléments de vocabulaires. La guerre du Golfe a brillé par ses frappes chirurgicales et ses *dommages collatéraux*, d'autres épisodes plus récents nous ont laissé des *feuilles de routes* à ne plus savoir qu'en faire.

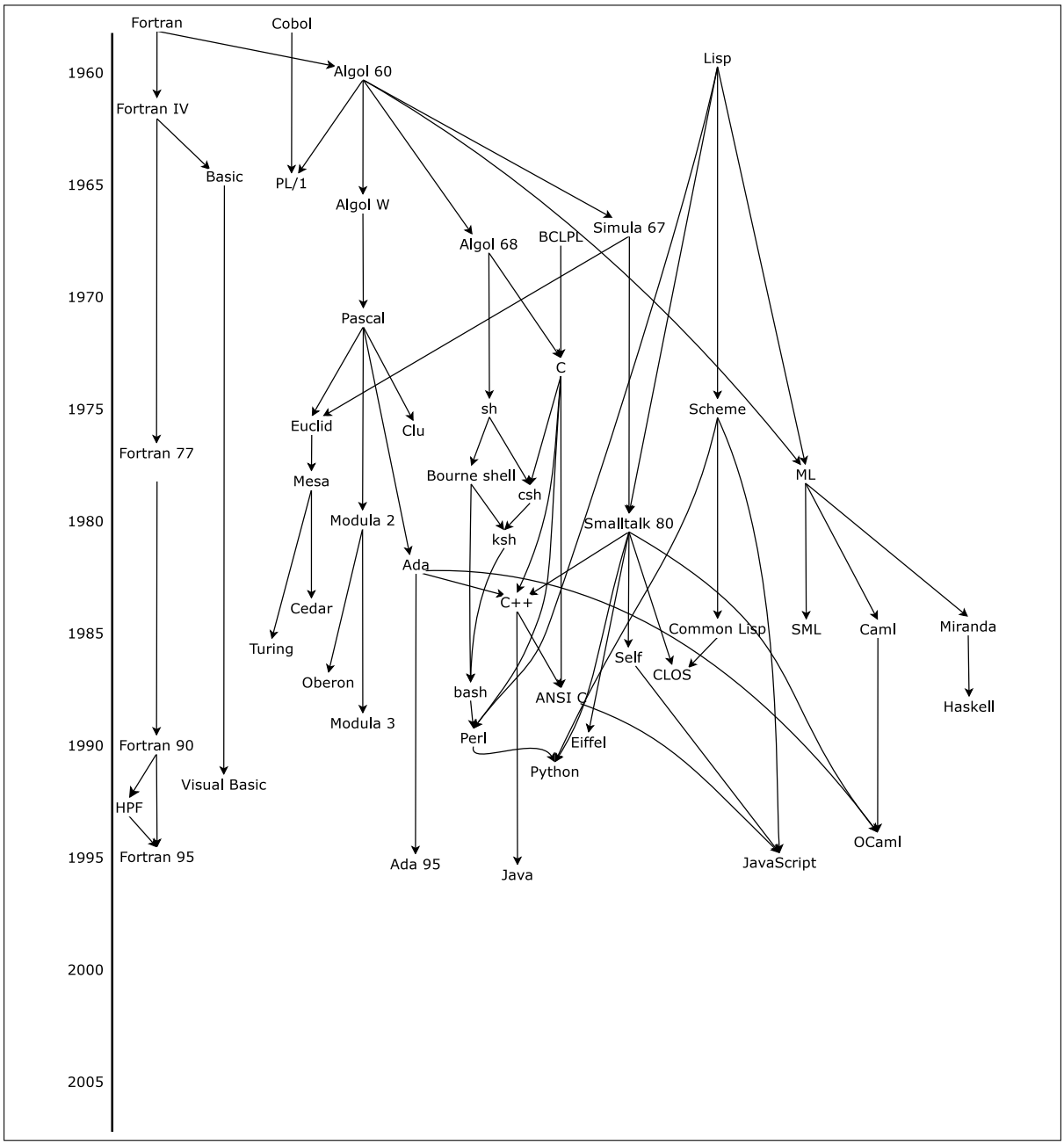


FIGURE 1.2 – Quelques points dans le nuage des langages

classe d'erreurs difficile à détecter. Bref, l'étude des langages apporte une forme de maturité qui permet de formuler des appréciations justes, motivées et exemptes de parti pris.

Connaître quelques langages représentatifs ou bien les principales caractéristiques qu'ils offrent permet aussi d'appréhender facilement tel ou tel nouveau venu dans la grande famille des langages de programmation, ou plus simplement de maîtriser les aspects essentiels d'un langage qu'il faut comprendre ou utiliser dans l'urgence.

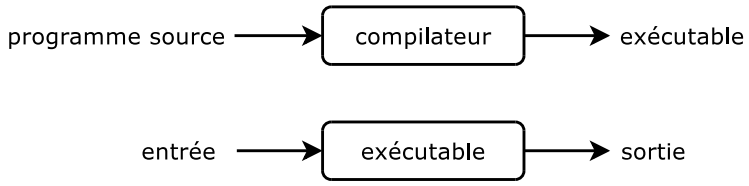
Les objectifs de ce cours sont donc d'apporter quelques éléments permettant d'avoir une vision un peu plus large des langages de programmation, de mieux comprendre les concepts qui sous-tendent

les constructions qu'ils offrent au programmeur. Le cours a à la fois des ambitions théoriques, puisqu'il utilise par exemple des présentations un peu formelles des sémantiques, mais aussi et surtout des ambitions pratiques, puisque les activités hors cours magistraux visent à produire des mises en œuvre effectives de certains aspects d'un langage particulier. En équilibrant ainsi théorie et pratique, on espère garder le recul nécessaire à l'examen de caractéristiques assez générales des langages de programmation sans se noyer dans des aspects particuliers à tel ou tel langage, tout en restant « les pieds sur terre » en prototypant certains aspects. Bien sûr, le risque existe de ne faire que survoler les différents aspects, sans en approfondir les aspects théoriques ni en examinant précisément les implantations efficaces. Par bonheur, la littérature approfondissant la théorie ou les implémentations est abondante, et j'orienterai avec grand plaisir vers les documents correspondants les élèves désireux d'en savoir plus.

1.4 Techniques de mise en œuvre

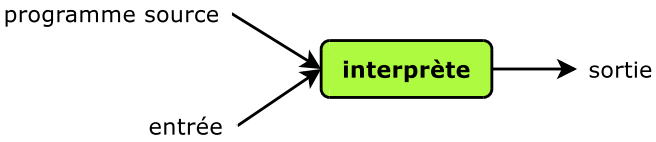
Ce cours n'est pas un cours de compilation, et, comme nous l'avons déjà indiqué ci-dessus, les techniques de génération et d'optimisation de code n'y sont pas décrites. Pourtant, un peu comme nous avons esquissé ci-dessus une classification en grandes familles des langages de programmation, il est intéressant de distinguer les différentes techniques d'implémentation des langages de programmation.

De façon très générale, la compilation d'un programme et son exécution peuvent être vues graphiquement sur le schéma suivant :



Le compilateur traduit le programme source en un programme directement exécutable par la machine. Le programme exécutable est alors indépendant du compilateur, et on peut l'exécuter sur des machines où le compilateur n'est pas présent. Ce programme peut toutefois nécessiter la présence de bibliothèques externes pour que son exécution puisse avoir lieu. De plus, l'exécutable généré par le compilateur ne pourra fontionner que sur des machines ayant un microprocesseur « compatible » avec celui visé lors de la compilation.

Un autre type de mise en œuvre de langages est l'*interprétation*. Un programme, qu'on appelle l'interprète, reçoit le programme source *avec ses entrées* et l'exécute.



Le programme n'est alors pas autonome : la présence de l'interprète est nécessaire pour que le programme puisse être exécuté.

Qu'ils soient interprétés ou compilés, certains langages de programmation sont dotés de pré-processeurs (par exemple C et C++ qui utilisent le préprocesseur `cpp`) qui effectuent un pré-traitement des programmes consistant à en transformer le code source en fonction de définitions ou de tests qui reflètent souvent des conditions liées à la machine sur laquelle la compilation ou l'exécution vont avoir lieu.

Cette dernière technique ne constitue pas à elle seule une technique d'implémentation de langage de programmation, mais elle a sa place dans le paysage un peu plus précis que nous tentons de dresser ci-dessous.

Les différentes régions de notre classification de langages ont dans la réalité des frontières très floues. De la même manière, un langage de programmation n'est généralement pas « ou bien compilé ou alors interprété » (avec un « ou » exclusif). Les langages de programmation utilisent des doses plus ou moins importantes de compilation, d'interprétation et de pré-traitement. Avant de donner un schéma plus général, tentons de définir un peu plus précisément ce que l'on appelle *compilation*, *interprétation* et pré-traitement.

On définit le pré-traitement (*pre-processing*, en Anglais) comme un procédé visant à transformer la syntaxe (le code source) des programmes. Dans la catégorie des pré-processeurs, on connaît bien sûr `cpp` qui est pour C et C++ un processus séparé du compilateur, mais appelé généralement *par* le compilateur. Citons aussi l'expansion des macros du langage Scheme, réalisée par le compilateur ou l'interprète Scheme, ou des préprocesseurs généraux comme `m4`. Le pré-traitement en lui-même connaît tout au plus la syntaxe du langage, mais n'effectue aucune analyse dite *sémantique* des programmes : le pré-traitement ne tente pas de *comprendre* les programmes.

La compilation d'un programme, quant à elle, peut être définie comme une famille de procédés qui traduisent le programme source en un autre programme, qu'on appellera le programme objet, sans disposer des données d'entrées du programme, et donc sans pouvoir l'exécuter, mais en développant une certaine compréhension du programme. Le code objet peut être du code machine, mais cela n'est pas nécessaire : on peut tout-à-fait envisager un compilateur traduisant, par exemple, du Scheme ou du Pascal en C. Et, de fait, de tels compilateurs existent. La compilation analyse donc plus ou moins les programmes pour les traduire en des programmes d'un autre langage. Par contre, la compilation d'un programme se fait sans disposer des entrées que recevra ce programme.

L'interprétation, quant à elle ne traduit pas : elle exécute. Étant donné un programme (de haut niveau comme Python, ou de bas niveau comme du bytecode OCaml ou du code machine) et les entrées de ce programme, un interprète exécute ce programme avec ses entrées pour en calculer un résultat ou en afficher les effets. Interpréter un programme, c'est donc le faire exécuter par une machine logicielle. Exécuter un programme au sens classique du terme, c'est l'interpréter avec la machine réelle.

On voit bien que si on mélange ces trois types d'opérations que sont pré-traitement, compilation et interprétation, on les effectue toujours dans un ordre où l'interprétation, qui est l'exécution du programme, a lieu en dernier. Le pré-traitement, quant à lui, peut avoir lieu de façon itérative (*i.e.* on effectue plusieurs pré-traitements successifs), suivi d'une passe de compilation optionnelle, qui consiste à traduire le programme source en un autre langage, généralement de plus bas niveau. En toute généralité, le programme obtenu doit ensuite être considéré comme un nouveau programme source qui peut subir à son tour pré-traitements et compilation. *In fine*, le dernier programme obtenu par compilation doit être interprété ou exécuté avec les données d'entrée pour obtenir un résultat. Notons qu'il est possible qu'une phase de compilation ait lieu *durant l'exécution* : il s'agit alors de compilation dite *just in time* (*JIT*), et ne concerne généralement qu'une partie du programme.

Le schéma donné à la figure 1.3 donne un résumé des différentes possibilités de traitements subis par un programme source en vue de son exécution. Sur ce schéma, les traits pleins représentent le chemin le plus classique, les traits pointillés sont des chemins alternatifs, qu'il est possible de prendre plusieurs fois (en nombre fini, bien sûr) lorsque plusieurs pré-traitements sont nécessaires ou lorsque la compilation produit un programme dans un langage nécessitant des traitements du même ordre.

1.5 Les différentes étapes de compilation

La compilation a généralement lieu selon un nombre de phases assez bien établies. Les premières phases consistent à découvrir (et vérifier) la structure syntaxique du programme : une **analyse lexicale** décompose le programme, présenté par un flux de caractères en mots ou lexèmes. Le but de cette

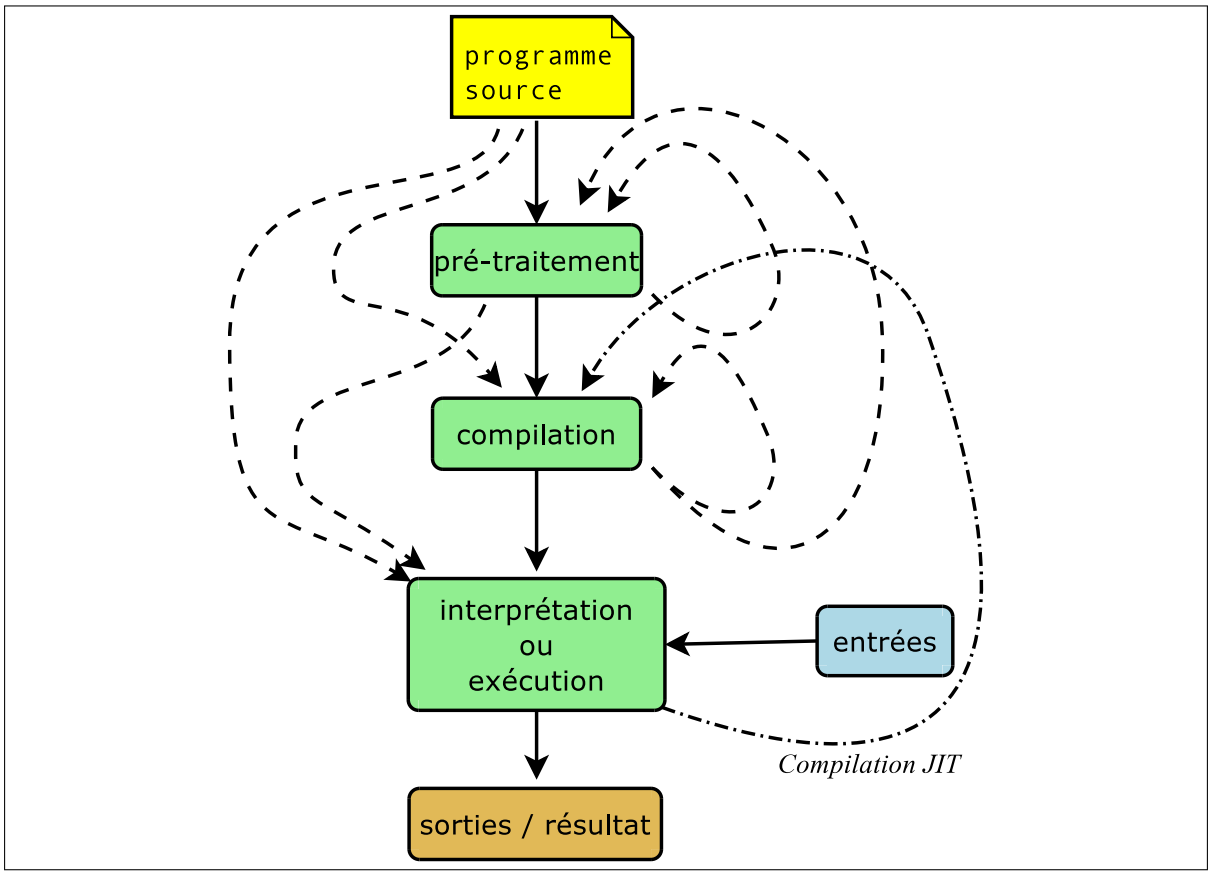


FIGURE 1.3 – Le chemin menant à l'exécution

première phase est, outre d'assembler les caractères en suites de mots, de distinguer les constantes littérales, les mots-clés et les identificateurs. Les lexèmes ont donc des formes différentes selon leur nature. Typiquement, l'analyseur lexical « oublie » les espaces, les retours à la ligne et les commentaires contenus dans le programme. Cette phase permet au passage de refuser les mots qui ne font pas partie du langage : c'est en quelque sorte une vérification d'orthographe.

Le flux de lexèmes ainsi produit est reçu par l'**analyseur syntaxique** qui va assembler ces mots en phrases du langages, exigeant par exemple que le lexème qui suit le mot-clé **while** soit une parenthèse ouvrante, ou qu'un bloc ouvert par « { » soit correctement refermé par « } », s'il s'agit par exemple d'analyser un programme C ou Java. L'analyseur syntaxique va donc composer ces suites de mots en phrases, et va produire en résultat une représentation structurée de ces phrases, à même d'être manipulée efficacement par la phase ultérieure. De ce fait, toute phrase mal formée sera rejetée : c'est en quelque sorte une vérification grammaticale. À ce stade, le programme est une donnée structurée, assez lointaine du texte initial, qu'on appelle généralement *arbre de syntaxe abstraite*, par opposition à la syntaxe dite concrète qui composait le programme initial. Nous parlons ici d'*arbre syntaxique superficiel*, car les phases ultérieures vont généralement transformer cet arbre en des formes encore plus éloignées du texte source initial. Bien sûr, dans de telles données arborescentes, on garde des informations concernant l'origine des éléments qui le composent (nom de fichier et position dans le fichier) de sorte à pouvoir afficher des messages d'erreurs informatifs si besoin est durant la compilation, voire transmettre ces informations jusque dans le code produit afin que des erreurs d'exécution puissent elles aussi afficher la partie fautive du programme dans le programme source lors de certaines erreurs d'exécution.

L'arbre de syntaxe abstraite représente maintenant tout le programme à compiler et il est relativement aisé de le parcourir de sorte à effectuer un certain nombre d'analyses qui dépendent du langage en

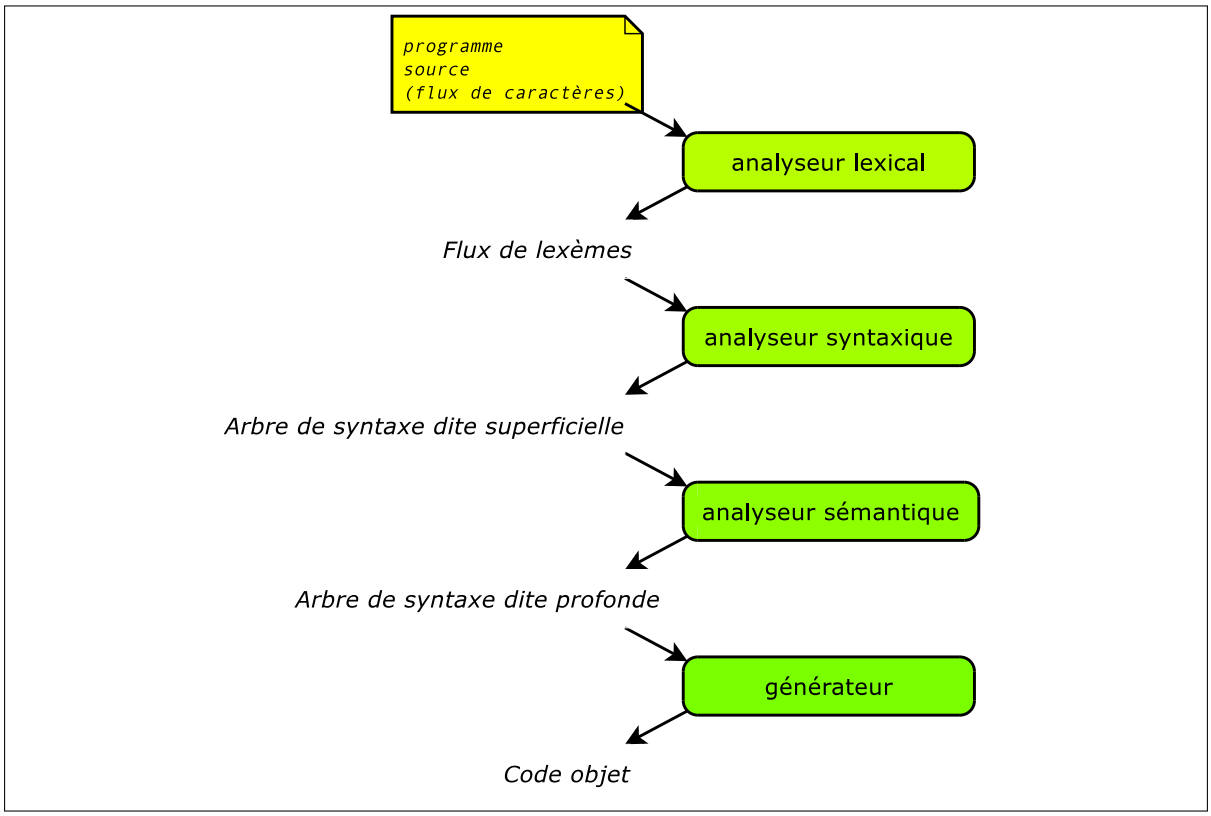


FIGURE 1.4 – Les grandes phases de la compilation

question : c'est le rôle de l'**analyse sémantique**. Par exemple, des analyses ou transformations concernant le typage – détecter certaines erreurs, propager des informations de types de sorte à optimiser la représentation des données –, ou, en OCaml, des transformations de programmes qui changent les analyses de cas complexes en cascades de tests plus simples, sorte de précompilation, sont des exemples typiques d'analyses sémantiques. Leur but peut être de rejeter certains programmes, de détecter des anomalies possibles pour en informer le programmeur ou de transformer et simplifier les programmes afin de les préparer à la phase ultérieure du compilateur. La phase d'analyse sémantique produit elle aussi une structure arborescente qui est, comme nous l'avons déjà remarqué ci-dessus, plus éloignée du programme source que l'arbre de syntaxe superficielle reçu en entrée. C'est pourquoi nous appelons cette structure résultante un **arbre de syntaxe profonde**.

La dernière étape de la compilation consiste en la traduction proprement dite, qui consiste généralement en la production de code assembleur. Cependant, nous adoptons ici une vision plus générale de la compilation et considérons qu'elle consiste en la production d'un programme équivalent au programme à compiler et écrit dans un langage différent (généralement de plus bas niveau que le premier). Lorsqu'il est question de compilation au sens classique du terme, cette phase peut faire intervenir des techniques très sophistiquées, notamment lorsque le compilateur cherche à produire du code machine optimisé, qui évite les séquences d'instructions inutiles, utilise au mieux les ressources offertes par le processeur (mémoire, registres, parallélisme interne, *etc*). Ce cours n'étant en aucune façon un cours de compilation, nous ignorerons ces aspects par la suite, et lorsque l'on parlera de compilation, ce sera de façon assez générale, voire abstraite.

1.6 OCaml

Le langage OCaml[6] sera utilisé dans ce cours comme outil d'implantation du langage que nous étudierons. OCaml est un langage essentiellement fonctionnel, doté de traits impératifs qui permettent une programmation à la Pascal ou Ada, dont la mémoire est gérée automatiquement comme par exemple dans le langage Java. Les attraits esssentiels des langages de la famille ML, et d'OCaml en particulier sont la facilité avec laquelle ils permettent l'écriture de programmes manipulant des structures de données complexes. Les programmes que nous écrirons dans ce cours vont manipuler essentiellement des arbres de syntaxe représentant d'autres programmes : ceux du langage que nous chercherons à implanter. De telles données se manipulent de façon beaucoup plus facile, plus concise, généralement plus sûre, voire même plus élégante[7] que dans les langages plus classiques.

OCaml dispose aussi d'objets et de modules : nous n'utiliserons probablement pas les premiers, par contre les seconds nous permettront de spécifier les interfaces des implémentations que nous développerons, et d'en masquer les détails le cas échéant.

Dans cette version de ces notes de cours, je n'ai pas souhaité rédiger un chapitre introductif à la programmation en OCaml. En effet, il existe de bons ouvrages sur le sujet, et les ressources disponibles sur Internet sont nombreuses.

Le site `https://ocaml.org` est lieu principal à partir duquel trouver des informations à propos d'OCaml. Il dispose de nombreux tutoriels, guides, pointeurs vers d'autres sites de qualité, etc. Il est donc intéressant de le conserver dans vos marque-pages de navigateur. . . au moins le temps du cours.

Une page en particulier vous sera très utile : `https://ocaml.org/docs`, dans laquelle le lien intitulé « Standard Library API » (ce lien change en fonction de la dernière version du langage) pointe vers la documentation en ligne des modules et fonctions disponibles dans la bibliothèque standard. Ainsi, pour savoir quelles fonctions sont disponibles pour manipuler des chaînes de caractères, on consultera la section du module `String`. On y verra alors, par exemple, qu'il existe une fonction `String.length` qui sert, sans surprise, à calculer et retourner la longueur de la chaîne reçue en argument. Pour comprendre le fonctionnement de la fonction `printf`, on ira se documenter dans la section du module `Printf`. Pour les listes, ce sera le module `List`, etc.

1.7 Installation d'OCaml et d'outils auxilliaires

OCaml ne dispose pas d'environnement de développement intégré (IDE) et se satisfait très bien des outils classiques tels que l'excellent Emacs, Visual Studio Code, Atom, GEdit et j'en oublie. Ces derniers permettent en effet d'éditer des fichiers source OCaml en les formattant avec l'indentation adéquate, et d'utiliser des couleurs permettant de distinguer les différentes entités lexicales du langage (mots-clés, identificateurs, constantes littérales, *etc.*).

Les directives d'installation en fonction de votre système d'exploitation sont disponibles à `https://ocaml.org/docs/up-and-running`. Cette installation s'appuie sur le gestionnaire de paquetages OCaml nommé Opam, qui permet de garantir les dépendances correctes entre les différentes bibliothèques que l'on peut choisir d'utiliser. Cela permet donc une installation sûre, pérenne et extensible.

Toutefois, dans le cadre de ce cours, nous n'utiliserons pas de bibliothèques « exotiques » et nous contenterons des outils fournis par la bibliothèque standard. Ainsi, une installation simple du paquetage de base d'OCaml en utilisant le gestionnaire de paquetages de votre système d'exploitation pourra suffire. Si vous êtes sous Linux, en fonction de votre distribution, ce gestionnaire sera vraisemblablement `apt`, `rpm`, `dnf`, `pacman`, `yum` ou d'autres qui ne me viennent pas à l'esprit.

En marge des outils relatifs à OCaml, ce cours nécessitera également l'outil `make` permettant de compiler des programmes comportant plusieurs fichiers source, tout en garantissant la compilation séparée en accord avec les dépendances entre ces différents fichiers. Il convient donc que cet outil soit

6. La dénomination « Objective Caml » a été abandonnée en 2012 : le langage s'appelle maintenant OCaml.
7. Mais c'est là presque une affaire de goût.

installé sur votre machine. Les distributions Linux et MacOS le fournissent par défaut. Sous les versions récentes de Windows (10 et 11), il conviendra d'installer un environnement Linux (WSL2) en suivant les indications disponibles à : `https://learn.microsoft.com/fr-fr/windows/wsl/install`. Par défaut une distribution Ubuntu sera installée (dont le gestionnaire de paquetages est `apt`), contenant *de facto* l'outil `make` tant désiré.

En dernier recours, tous ces outils sont installés sur le serveur `byod.ensta.fr`, que vous avez utilisé en cours d'informatique de première année. Il vous est donc possible de ne rien installer d'autre que Visual Studio Code sur votre machine, et d'utiliser la connexion SSH à distance à ce serveur, au travers de Visual Studio Code (comme vous avez fait en première année).

Chapitre 2

Expressions rationnelles, automates, analyse lexicale

L'analyse lexicale est la première phase d'un compilateur ou d'un interprète : elle consiste à identifier et à catégoriser les différents mots qui constituent un programme. Ces « mots » ou entités lexicales sont appelés *lexèmes* (*tokens* en Anglais). Par exemple, si on considère l'extrait suivant du programme Pascal vu au chapitre précédent :

```
while i ≠j do (* commentaire *)
  if i > j then
    i := i - j
```

et si l'on représente les entités lexicales de Pascal par un type OCaml du style :

```
type token =
  FOR | WHILE | DO | DONE | IF | THEN | ELSE | ...
| EQUAL | COLONEQUAL | PLUS | MINUS | STAR | SLASH
| GREATER | SMALLER | SMALLERGREATER
| IDENT of string | INT of int | FLOAT of float
| ...
```

l'analyseur lexical devra produire la suite suivante :

```
WHILE, IDENT("i"), SMALLERGREATER, IDENT("j"), DO, IF, ...
```

où les mots sont classés en un certain nombre de catégories : symboles, mots-clés, identificateurs, constantes littérales, *etc.*, et où les espaces et commentaires ont disparu.

L'analyseur syntaxique, qui cherchera à donner de la structure à cette suite de mots (voir chapitre suivant), vérifiera qu'il est bien légal d'avoir une séquence `WHILE`, ..., `DO`, ..., `DONE`, pourvu que ce qui correspond aux points de suspension soit bien, respectivement, une expression et une séquence d'instructions. Comme nous l'avons remarqué auparavant, l'analyseur syntaxique questionnera probablement l'analyseur lexical chaque fois qu'il est nécessaire, lui demandant « quel est le prochain

lexème ? » On peut donc considérer que l'interaction entre l'analyseur lexical et l'analyseur syntaxique est de type « maître/esclave », avec l'analyseur syntaxique étant le « maître ».

Se posent alors les deux questions suivantes :

1. Comment spécifier les catégories lexicales d'un langage ?
2. Si l'on dispose d'une telle spécification, comment fabrique-t-on l'analyseur correspondant ?

La spécification des catégories lexicales est habituellement réalisée en utilisant la notion d'*expressions rationnelles*, qui apporte à la fois une notation et un cadre théorique élégants pour régler ce problème. Ce cadre théorique est tellement bien compris qu'il fournit aussi la réponse à la seconde question : à la notion d'expression rationnelle correspond très exactement celle d'automate fini, qui donne le moyen d'implémenter efficacement et mécaniquement des analyseurs lexicaux.

2.1 Expressions rationnelles

Prenons l'exemple des entiers naturels. On sait qu'ils sont habituellement écrits comme des suites de chiffres, en commençant par un chiffre non nul. Pour dire qu'un chiffre non nul est ou bien le caractère '1', ou alors '2', ou alors '3', *etc.*, on peut noter :

`chiffre_non_nul` ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

où le symbole ::= indique une définition, et la barre verticale dénote une alternative (similaire à l'utilisation de ce symbole dans une définition de type OCaml). On peut alors définir un `chiffre` comme :

`chiffre` ::= '0' | `chiffre_non_nul`

et un entier naturel comme :

`naturel` ::= '0' | (`chiffre_non_nul` `chiffre`*)

où la juxtaposition se lit comme « suivi de » et `chiffre`* représente une séquence éventuellement vide d'éléments de `chiffre`.

Cette notation est appelée la notation des *expressions rationnelles*. Les expressions rationnelles servent à désigner de façon concise des ensembles de *mots* sur un alphabet. Si Σ est un alphabet, l'ensemble des *mots* sur Σ, noté Σ^*, est l'ensemble de toutes les séquences possibles d'éléments de Σ. L'opération qui sert à construire les séquences est la concaténation, et est notée par simple juxtaposition. Si m_1 et m_2 sont des mots de Σ^*, alors leur concaténation $m_1 m_2$ est aussi un mot de Σ^*. Le mot vide est noté ϵ, et est neutre pour la concaténation. La concaténation est une opération associative.

Les entités que nous avons nommées ci-dessus `chiffre` ou `naturel` désignent en fait des sous-ensembles de mots sur un alphabet qui contient notamment les dix chiffres. On appelle *langage* un sous-ensemble de Σ^*, pour Σ donné. La notation que nous avons utilisée pour identifier les langages `chiffre_non_nul`, `chiffre` et `naturel` est appelée *expression rationnelle*.

Définition (expressions rationnelles) Étant donné un *alphabet* Σ, on appelle *expression rationnelle* une expression construite par les règles suivantes :

- tout élément de Σ est une expression rationnelle ;
- si e_1 et e_2 sont deux expressions rationnelles, alors la concaténation $e_1 e_2$ est une expression rationnelle ;
- le *mot vide* noté ϵ est une expression rationnelle ;
- si e_1 et e_2 sont deux expressions rationnelles, alors l'*alternative* $e_1|e_2$ est une expression rationnelle ;
- si e est une expression rationnelle, l'*itération* e^* est une expression rationnelle.

L'opérateur « * » est appelée itérateur de Kleene. On utilise quelquefois le terme *expression régulière*, au lieu d'expression rationnelle : cela est dû à la langue anglaise qui les nomme *regular expressions*.

Étant donnés Σ et e une expression rationnelle sur Σ, le langage $\mathcal{L}(e)$ désigné par l'expression rationnelle e peut être défini de la manière suivante.

Définition (langage désigné par une expression rationnelle) Le langage $\mathcal{L}(e)$ désigné par une expression rationnelle e sur l'alphabet Σ est défini par :

- $\mathcal{L}(\epsilon) = \{\epsilon\}$
- $\mathcal{L}(a) = \{a\}$, où $a \in \Sigma$
- $\mathcal{L}(e_1e_2) = \{m_1m_2 \text{ avec } m_1 \in \mathcal{L}(e_1) \text{ et } m_2 \in \mathcal{L}(e_2)\}$
- $\mathcal{L}(e_1|e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$
- $\mathcal{L}(e*) = \{\epsilon\} \cup \mathcal{L}(e) \cup \mathcal{L}(ee) \cup \ldots$
  $= \bigcup_{i=0}^{\infty} \mathcal{L}(e^i)$

On utilise bien sûr un parenthésage pour grouper les constructions, et on adopte la convention que la concaténation est prioritaire vis-à-vis de l'alternative, de sorte que

$e_1e_2|e_3e_4$ sera lu comme $(e_1e_2)|(e_3e_4)$

et l'étoile sera prioritaire sur les autres opérateurs :

$e_1e_2{}^*|e_3$ sera lu comme $(e_1(e_2{}^*))|e_3$

À partir de cette notation de base, on utilise très souvent des formes dérivées qui représentent des combinaisons des constructions de base ci-dessus. Par exemple :

- e? désigne un e optionnel, c'est-à-dire $e|\epsilon$
- e^+ désigne ee^*
- [abc] désigne $(a|b|c)$
- [a_1-a_2] désigne l'un des éléments de Σ compris entre a_1 et a_2, ce qui suppose que les éléments de Σ soient totalement ordonnés
- [^abc] désigne l'un des éléments du complémentaire de $\{a, b, c\}$ dans Σ

Notons aussi que les définitions de langages de programmation utilisent aussi souvent les crochets [et] pour encadrer les éléments optionnels, et les accolades { et } pour encadrer les occurrences multiples. Ainsi, on a, dans ces cas :

- [e] pour $e|\epsilon$
- {e} pour e^*
- {e}$^+$ pour ee^*

Au vu de toutes ces possibilités (non exhaustives), il est prudent, en présence d'un document utilisant de telles notations, de vérifier l'interprétation qu'il faut en faire.

2.2 Interprétation directe des expressions rationnelles

Utiliser une expression rationnelle pour spécifier la forme des lexèmes sert assez aisément de référence pour programmer la reconnaissance des lexèmes correspondants. Par exemple, pour les chiffres non nuls, on pourrait écrire une fonction de reconnaissance qui indique si le début d'une liste de caractères contient ou non un chiffre non nul :

```
let chiffre_non_nul cs =
  match cs with
  |      (('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')      as d) :: rcs →
    Some ((Char.code d − Char.code '0'), rcs)
  | _ → None ;;
val chiffre_non_nul : char list → (int * char list ) option = <fun>
```

La fonction précédente reçoit une liste de caractères en entrée et produit un résultat booléen qui indique si la reconnaissance s'est opérée avec succès ainsi que le reste non consommé de la liste d'entrée.

Nous pouvons maintenant programmer la reconnaissance d'un chiffre en utilisant le type `option` :

```
let chiffre cs =
  match cs with
  | '0' :: rcs → Some (0, rcs)
  | cs → chiffre_non_nul cs
;;
val chiffre : char list → (int * char list ) option = <fun>
```

En fait, les fonctions d'ordre supérieur d'OCaml permettent de coder directement la séquence, l'alternative ainsi que l'étoile :

```
let rec zero_ou_plus e cs =
  match e cs with
  | Some(r, rcs1) →
    (match zero_ou_plus e rcs1 with
     | Some (rs, rcs2) → Some ((r :: rs), rcs2)
     | None → Some([r], rcs1))
 | None → Some ([ ], cs)
;;
val zero_ou_plus : (α → (β * α) option) → α→ (β list * α) option = <fun>

let et_puis e1 e2 cs =
  match e1 cs with
  | Some (r1, rcs1) →
    (match e2 rcs1 with
     | Some (r2, rcs2) → Some ((r1, r2), rcs2)
     | None → None)
  | None → None
;;
val et_puis :
  (α → (β * γ) option) →
  (γ → ('d * 'e) option) → α→ ((β * 'd) * 'e) option = <fun>

let ou_bien e1 e2 cs =
  match e1 cs with
  | Some (r1, rcs1) as res1 → res1
  | None → e2 cs
;;
val ou_bien :
  (α → (β * γ) option) →
  (α → (β * γ) option) → α→ (β * γ) option = <fun>

let zero cs =
  match cs with
  | '0' :: rcs → Some (0, rcs)
  | _ → None
```

```
;;
val zero : char list → (int * char list ) option = <fun>

let naturel cs =
  ou_bien zero (et_puis chiffre_non_nul (zero_ou_plus chiffre)) cs
;;
val naturel : char list → ((int * int list ) * char list ) option = <fun>
```

Nous pouvons maintenant tester l'analyseur `naturel` en définissant au préalable une fonction qui convertit une liste de caractères en en entier :

```
let digits_to_int digits =
  let rec digirec ds =
    match ds with
    | [ ] → 0
    | d :: ds → (Char.code d − Char.code '0') + 10 * (digirec ds) in
  digirec (List.rev digits)
;;
val digits_to_int : int list → int = <fun>

let nat cs =
  match naturel cs with
  | Some ((d, ds), cs) → Some (digits_to_int (d :: ds), cs)
  | None → None
;;
val nat : char list → (int * char list ) option = <fun>

let _ = nat ['1'; '2'; '3'; '4'; 'z'] ;;
− : (int * char list ) option = Some (1234, ['z'])

# nat ['0'; '1'; 'z'] ;;
− : (int * char list) option = None
```

En fait, même s'il s'agit d'un exercice de programmation intéressant, cette interprétation directe a des limites. Par exemple, le codage présenté ici produit un programme qui ne termine pas lorsqu'on interprète naïvement $(\epsilon|a)^*$, qui va conduire à itérer à l'infini l'analyseur associé à ϵ.

2.3 Automates finis

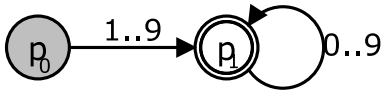
Une phase de *compilation* des expressions rationnelles se révèle donc nécessaire à l'implémentation correcte des analyseurs associés. Il se trouve que les expressions rationnelles sont équivalentes à ces sortes de machines virtuelles que sont les *automates* finis. Un automate fini *déterministe* est une machine dotée d'un ensemble fini d'états, et qui lit des éléments d'un alphabet, faisant passer la machine d'un état à un autre à chaque lecture. Formellement :

Définition (automate) Un automate fini déterministe est la donnée de :
- un alphabet fini Σ ;
- un ensemble fini d'états Q ;
- une fonction de transition $\delta : Q \times \Sigma \to Q$;
- un état initial p_0 ;
- un ensemble d'états finaux F.

Un automate fini *non-déterministe* est un automate qui offre plusieurs changements d'état pour un même symbole, voire des changements d'états spontanés, c'est-à-dire sans lire aucun symbole. La fonction de transition d'un automate fini non-déterministe est donc de la forme $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$.

Le langage accepté par un automate A sur un alphabet Σ est l'ensemble des mots sur Σ qui permettent de faire passer A de l'état initial à un état final.

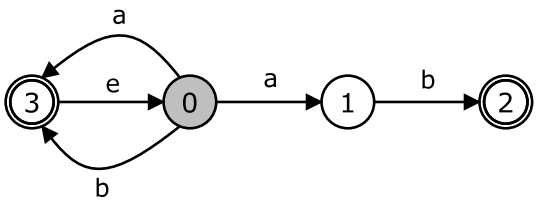
Par exemple, l'automate suivant, composé de deux états p_1 et p_2, représente l'expression rationnelle $(1..9)(0..9)^*$ qui correspond aux entiers naturels donnés ci-dessus. L'état p_1 est final, et nous avons remplacé les 9 transitions sur chacun des caractères de l'ensemble $0..9$ par une seule transition étiquetée $1..9$.



Cet automate se lit de la manière suivante : initialement, l'automate est dans l'état p_0. S'il lit un élément de $1..9$, l'automate passe à l'état p_1, sinon on s'arrête en échec. Puisque l'état p_1 est un état final (que l'on dessine avec un double cercle), l'automate est autorisé à s'y arrêter. On dit alors que la reconnaissance s'est opérée avec succès. Si l'on lit un élément de $0..9$, on retourne dans l'état p_1, et on peut de nouveau choisir de continuer ou d'arrêter.

On voit que cet automate déterministe nous laisse le choix de continuer ou d'arrêter lorsqu'on a reconnu un mot acceptable qui est préfixe d'un autre. Les analyseurs lexicaux sont très souvent confrontés à de tels choix lorsque, par exemple, un préfixe d'un identificateur est un mot-clé du langage (par exemple, en OCaml, le mot-clé **let** est un préfixe de l'identificateur `lettre`). On fait toujours le choix de continuer quand c'est possible : c'est ce qu'on appelle la sémantique du *filtrage le plus long* ou de *longest match* en anglais.

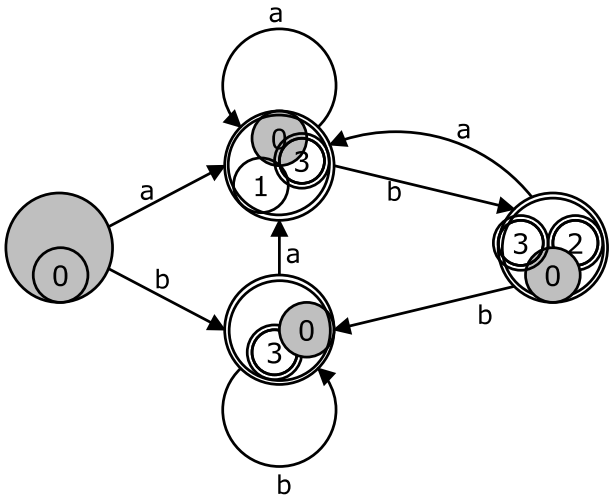
Un automate non déterministe peut, quant à lui, offrir des points de choix aux états qui ont plusieurs transitions possibles sur un même élément de l'alphabet. Par exemple, l'automate suivant, qui reconnaît tous les mots composés d'au moins une lettre sur l'alphabet {a,b}, offre deux possibilités de reconnaissance du mot ab à partir de l'état initial grisé : une possibilité part vers la gauche pour arriver à l'état final 3 et l'autre vers la droite pour arriver à l'état final 2.



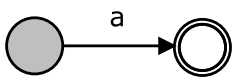
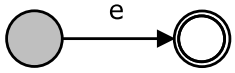
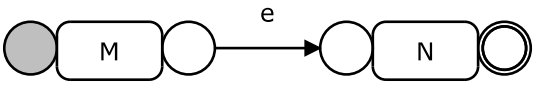
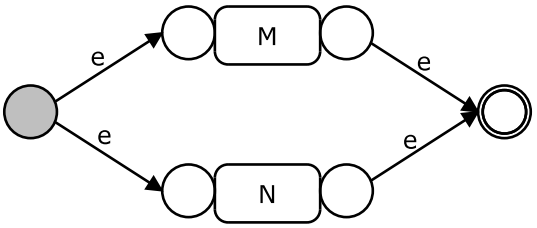
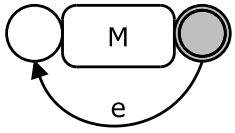
Par bonheur, les automates finis non-déterministes sont équivalents aux automates finis déterministes ! À l'évidence, un automate déterministe est un cas particulier d'automate non déterministe, mais c'est vrai aussi dans l'autre sens : il existe un algorithme qui traduit un automate non-déterministe en automate déterministe. À partir d'un automate non déterministe N, l'algorithme procède à la construction d'un automate déterministe D dont les états seront des ensembles d'états de N. L'état initial d_0 de D est constitué de l'état initial de N et des états de D qui sont accessibles, directement ou indirectement, par des transitions sur ϵ. Ensuite, pour chaque élément s de Σ sur lequel l'un au moins des éléments de d_0 a une transition, on construit une transition sur s partant de d_0 vers un nouvel état constitué de l'ensemble

des états destination des transitions sur s au départ de l'un des éléments de d_0. On complète cet état en y ajoutant les états de N accessibles par des transitions spontanées issues de l'un de ses éléments. On procède ainsi récursivement pour chacun des états ainsi créés. Les états finaux de D seront les états qui contiennent un état final de N. L'alphabet étant fini, ainsi que l'ensemble des ensembles d'états de N, on est certain que le processus termine.

Voici l'automate déterministe obtenu à partir de l'automate non-déterministe ci-dessus, en utilisant cet algorithme :



Il nous reste à voir comment passer systématiquement d'une expression rationnelle à un automate non-déterministe. Il s'agit d'une compilation qui va traduire une expression rationnelle en une représentation d'automate non déterministe. Cette traduction est donnée par la table suivante, qui décrit un processus récursif de traduction, défini par cas selon la structure de l'expression à traduire. Les boîtes correspondent aux automates générés pour les sous-expressions de l'expression considérée. Pour celles-ci, l'état initial est toujours à gauche, et l'état final à droite. Lorsque ces boîtes sont intégrées dans un automate englobant, ces états perdent leur statut particulier, et l'état initial du résultat est grisé, alors que l'état final est doublement cerclé.

Si l'expression rationnelle est	on produit
$a \in \Sigma$	
ϵ	
MN	
$M\|N$	
M^*	

L'inconvénient des automates non déterministes est leur difficulté d'implémentation car il faut maintenir plusieurs états en parallèle. Puisqu'il y a équivalence entre automates non déterministes et déterministes et qu'il existe un algorithme pour passer des premiers aux seconds, il n'y à qu'à appliquer cet algorithme sur l'automate obtenu à partir des règles ci-dessus. L'inconvénient de la procédure de construction que nous avons décrite ici est que la déterminisation peut produire des automates avec un grand nombre d'états. En effet, le nombre d'états de l'automate déterminisé peut être exponentiel par rapport à celui de l'automate non déterministe d'origine.

Dans les faits, les générateurs d'analyseurs lexicaux reçoivent essentiellement une expression rationnelle, et produisent directement un automate déterministe. Nous n'abordons pas ici les techniques utilisées, de plus amples informations pouvant se trouver aisément dans l'abondante littérature consacrée à la théorie des automates.

2.4 Générateurs d'analyseurs lexicaux

Nous décrivons dans cette section le générateur d'analyseurs lexicaux `ocamllex` qui génère du OCaml. Il existe de nombreux équivalents pour produire des analyseurs dans d'autres langages : Lex pour C (l'un des premiers générateurs, suivi de Flex), JFlex pour Java, FsLex pour F#, etc. en étant très loin d'une énumération exhaustive.

La commande `ocamllex` produit un analyseur lexical à partir d'un ensemble d'expressions rationnelles auxquelles sont attachées des actions dites « sémantiques », qui sont exécutées à chaque fois qu'une expression rationnelle a été reconnue. L'exécution de :

```
$ ocamllex monlexer.mll
```

produit le fichier `monlexer.ml` contenant le code OCaml de l'analyseur lexical correspondant. Les fonctions produites prennnent en argument une mémoire tampon (un *buffer*) d'analyse lexicale, reconnaissent

dans ce *buffer* une occurrence d'expression rationnelle et produisent une des valeurs spécifiées dans les actions sémantiques.

Un *buffer* peut être construit à partir d'une des fonctions suivantes :

`Lexing.from_channel` : `in_channel` → `lexbuf`
`Lexing.from_string` : `string` → `lexbuf`

La première permet à l'analyseur de récupérer le flot de caractères au travers du descripteur d'un fichier en lisant le contenu de ce fichier. La seconde obtient ce flot en parcourant le contenu d'une chaîne de caractères.

La fonction :

`Lexing.lexeme` : `lexbuf` → `string`

extrait de la mémoire tampon la chaîne correspondant à l'expression rationnelle reconnue.

La syntaxe des définitions d'analyseurs lexicaux est la suivante :

```
{ header }
let ident = regexp ...
rule entrypoint [arg1... argn] =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint [arg1... argn] =
  parse ...
and ...
{ trailer }
```

L'en-tête « `header` », nécessairement placée à l'intérieur d'accolades est du code OCaml qui sera recopié textuellement au début du fichier produit. Idem, mais à la fin, pour « `trailer` ».

Les règles définissent des « points d'entrée » – des fonctions dotées de paramètres – qui sont les analyseurs lexicaux définis. En général un seul de ces points d'entrée est utilisé par les fonctions appelantes. Par contre, ces points d'entrée sont là pour d'une part structurer les définitions d'analyseurs lexicaux, en nommant des analyseurs auxilliaires, mais aussi pour procéder à des appels récursifs qui sortent du cadre strict des expressions rationnelles (voir chapitre suivant).

Les `regexp` sont des expressions rationnelles, dans une syntaxe plus riche que celle que nous avons vue jusque là, et les « `action` » entre accolades sont des expressions OCaml qui seront renvoyées en résultat lorsqu'un « `entrypoint` » aura reconnu une « `regexp` ».

L'exemple de la reconnaissance des entiers naturels que nous avons vu au début de ce chapitre peut s'écrire :

```
{
  exception Erreur
}

let chiffre_non_nul = ['1'-'9']

rule naturel = parse
  chiffre_non_nul ('0' | chiffre_non_nul)*
```

```
    { int_of_string (Lexing.lexeme lexbuf) }
| [' ' '\n' '\t']
    { naturel lexbuf }
| _
    { raise Erreur }

{
  (* Rien. *)
}
```

La compilation se fait par :

```
$ ocamllex naturel.mll
4 states, 267 transitions, table size 1092 bytes
```

et donne quelques informations au sujet de l'automate produit.

Le code OCaml de l'analyseur produit se trouve dans le fichier `naturel.ml` qui a été engendré par la commande ci-dessus :

```
$ cat naturel.ml
# 1 "naturel.mll"

  exception Erreur

# 6 "naturel.ml"
let __ocaml_lex_tables = {
  Lexing.lex_base =
   "\000\000\253\255\254\255\010\000";
  Lexing.lex_backtrk =
   "\255\255\255\255\255\255\000\000";
  Lexing.lex_default =
   "\001\000\000\000\000\000\255\255";
  Lexing.lex_trans =
   "\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\
    ...
    \000\000\000\000\000\000";
  Lexing.lex_check =
   "\255\255\255\255\255\255\255\255\255\255\255\255\255\255\255\255\
    ...
    \255\255\255\255\255\255";
  Lexing.lex_base_code = "";
  Lexing.lex_backtrk_code = "";
  Lexing.lex_default_code = "";
  Lexing.lex_trans_code = "";
  Lexing.lex_check_code = "";
  Lexing.lex_code =  "";
}
let rec naturel lexbuf =
    __ocaml_lex_naturel_rec lexbuf 0
and __ocaml_lex_naturel_rec lexbuf __ocaml_lex_state =
```

```
  match Lexing.engine __ocaml_lex_tables __ocaml_lex_state lexbuf with
      | 0 →
# 9 "naturel.mll"
      ( int_of_string (Lexing.lexeme lexbuf) )
# 105 "naturel.ml"

  | 1 →
# 11 "naturel.mll"
      ( naturel lexbuf )
# 110 "naturel.ml"

  | 2 →
# 12 "naturel.mll"
      ( raise Erreur )
# 115 "naturel.ml"

  | __ocaml_lex_state →
        lexbuf.Lexing.refill_buff lexbuf;
        __ocaml_lex_naturel_rec lexbuf __ocaml_lex_state
;;
# 14 "naturel.mll"
# 125 "naturel.ml"
```

Pour utiliser cet analyseur, nous devons le compiler, le charger et l'appeler :

```
$ ocamlc naturel.ml

$ ocaml
          OCaml version ...

# #load "naturel.cmo" ;;
# Naturel.naturel ;;     (* Analyseur lexical . *)
- : Lexing.lexbuf → int = <fun>
# Naturel.naturel (Lexing.from_string "1234abcd") ;;
- : int = 1234
# let buf = Lexing.from_string "1234abcd" ;;   (* On nomme le tampon de lecture . *)
val buf : Lexing.lexbuf = { ... }
# Naturel.naturel buf ;;   (* Quel est le prochain lexème ? *)
- : int = 1234
# Naturel.naturel buf ;;   (* Reste-t-il un lexème ? *)
Exception: Naturel.Erreur.
```

Chapitre 3

Grammaires algébriques, analyse syntaxique

Tous les langages de programmation sont dotés d'une définition de la structure syntaxique de leur programmes, et c'est à cette définition que l'on a recours, explicitement ou non, lorsque l'on écrit un programme, ou lorsque l'on cherche à comprendre pourquoi le compilateur refuse tel ou tel morceau de programme avec un message du style « `erreur de syntaxe` » ou « `syntax error` ».

La définition de la structure syntaxique d'un langage (informatique ou non) est appelée sa *grammaire*. La grammaire d'un langage naturel est la plupart du temps très complexe, mais, par bonheur, son non-respect n'est généralement pas fatal à la compréhension. *A contrario*, la grammaire d'un langage informatique est plutôt simple, mais, par contre, son non-respect est la plupart du temps jugé fatal par l'analyseur syntaxique qui tente de reconnaître les phrases du langage en question.

La simplicité syntaxique des langages informatiques est telle que leur grammaire sert non seulement de référence utilisable par un humain, mais aussi de spécification que l'on peut donner en entrée à un générateur d'analyseurs syntaxiques pour qu'il produise automatiquement un analyseur du langage. Il ne faut pas toutefois trop se fier à cette simplicité qui n'est que superficielle et n'est bien sûr pas suffisante pour *comprendre* un programme. On a beau connaître la grammaire d'un langage, la lecture de tous ses programmes n'en est pas facile pour autant. Des éléments comme l'indentation, les choix des identificateurs (noms de variables, de fonctions), voire la structuration même des programmes peuvent en compliquer significativement la lecture, et donc la réutilisation et la maintenance. À titre d'exemple (extrême), le code suivant est un code C parfaitement valide, qui imprime les nombres premiers inférieurs à 100 :

```
#include <stdio.h>
_(__,___,____){___/__<=1?_(__,___+1,____):!(___%__)?_(__,___+1,0):___%__==___/
__&&!____?(printf("%d\t",___/__),_(__,___+1,0)):___%__>1&&___%__<___/__?_(__,1+
___,____+!(___/__%(___%__))):___<__*__?_(__,___+1,____):0;}main(){_(100,0,0);}
```

Cet exemple est tiré de http://en.wikipedia.org/wiki/Obfuscated_code.

En d'autres termes, tout comme en littérature, la lisibilité n'est pas qu'une affaire de syntaxe : c'est aussi une affaire de style et de discipline !

Plus sérieusement, l'analyse syntaxique a pour but d'*organiser les lexèmes* des programmes en une représentation compréhensible, de sorte à pouvoir la traiter systématiquement par le reste de la chaîne de compilation. Elle a aussi pour rôle de *refuser* les programmes syntaxiquement mal construits, auxquels le reste de la chaîne ne saurait donner de sens. La figure 3.1 rappelle la place de l'analyse syntaxique dans la chaîne de compilation.

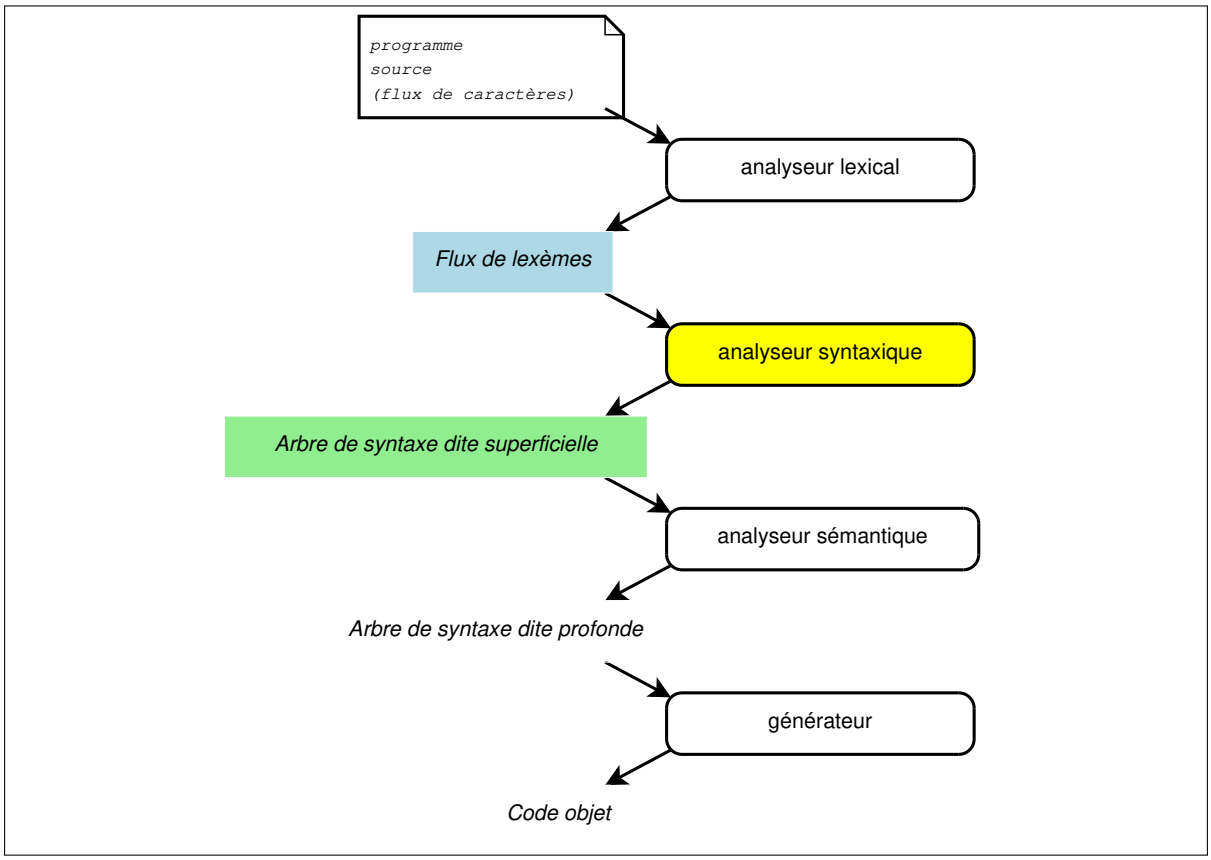


FIGURE 3.1 – La place de l'analyse syntaxique dans la chaîne de compilation

3.1 Grammaires

Une grammaire pour un langage est la donnée des éléments constituant le langage (les mots ou lexèmes), et les différentes règles d'assemblage de ces mots. Ces règles d'assemblage définissent la plupart du temps des catégories syntaxiques (comme *expression* ou *instruction*). Dans la suite, on appellera *terminaux* les lexèmes du langage, et *non terminaux* les catégories syntaxiques. Nous justifierons plus tard cette appellation.

Les règles sont notées $\alpha \to \beta$ où α et β sont des séquences de terminaux et de non-terminaux. Dans les cas des grammaires qui nous intéressent (les grammaires dites *algébriques*), elles ont la forme particulière plus simple $A \to \beta$ où A est un non terminal.

La définition de la notion générale de grammaire est donnée ci-dessous. Dans cette définition, étant donnés des ensembles A et B, on note A^* l'ensemble des séquences éventuellement vides d'éléments de A, et AB l'ensemble des concaténations d'éléments de A et d'éléments de B.

Définition (Grammaire) Une grammaire G est un quadruplet (N, T, P, S) où :
- N est un ensemble fini de symboles dits *non terminaux*
- T est un ensemble de symboles dits *terminaux*
- P est l'ensemble des *productions* : c'est un sous-ensemble de $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$; on note $\alpha \to \beta$ un élément (α, β) de P
- S est un symbole particulier de N et est appelé le *symbole de départ*.

On remarquera que le membre gauche d'une production doit comporter au moins une occurrence de non terminal. Intuitivement, c'est la catégorie syntaxique associée à ce non terminal que cette production

contribue à définir.

Notation : on regroupe souvent dans une même règle $\alpha \to \beta_1|\beta_2|\ldots|\beta_n$ l'ensemble des règles $\alpha \to \beta_i$ ayant le même membre gauche α.

Le langage $\mathcal{L}(G)$ engendré par une grammaire G est l'ensemble des suites de mots produits par toutes les réécritures possibles issues du symbole de départ S (on suppose avoir au moins une production $S \to \alpha$).

Cette forme de grammaire est beaucoup trop générale pour notre besoin. De fait, le linguiste Chomsky a classé dans les années 1950 les langages par les grammaires qui les engendraient.

Définition (Hiérarchie de Chomsky) Une grammaire est dite :

- de type 3 ou **rationnelle** si chaque production est ou bien de la forme $A \to wB$ ou $A \to w$ où A et B sont des non terminaux et w est un terminal, ou alors de la forme $A \to \epsilon$;
- de type 2 ou **algébrique** ou **non contextuelle** si chaque production est de la forme $A \to \alpha$ où A est un non terminal et α est une séquence de terminaux ou non terminaux ;
- de type 1 ou **contextuelle** si chaque production est de la forme $\alpha A\beta \to \alpha\gamma\beta$ où α, β et γ sont des séquences de terminaux ou non terminaux et $\gamma \neq \epsilon$;
- de type 0 ou **générale** si chaque production est de la forme $\alpha \to \beta$, sans autre contrainte.

On reconnaît dans cette définition les grammaires rationnelles (type 3) qui engendrent les langages rationnels que nous avons rencontrés au chapitre précédent. Par exemple, la grammaire ci-dessous engendre le langage noté a^*bc^* sous la forme d'expression rationnelle.

$$S \to aS$$
$$S \to bC$$
$$C \to cC$$
$$C \to \epsilon$$

Les grammaires des langages de programmation sont le plus souvent des grammaires de type 2 (dites algébriques, ou non contextuelles), dont les catégories syntaxiques sont constituées indépendamment du contexte où elles apparaissent. Le langage a^ib^i pour $i > 0$ peut être défini par la grammaire algébrique suivante :

$$S \to aSb$$
$$S \to ab$$

Le langage a^ib^i (c'est-à-dire l'ensemble des mots composés d'autant d'occurrences successives de a que de b) n'est pas rationnel, et ne peut donc pas être défini par une grammaire rationnelle.

Enfin, le langage $a^ib^ic^i$ pour $i > 0$ n'est, quant à lui, pas algébrique. Pour le définir sous la forme d'une grammaire, on doit avoir recours à une grammaire contextuelle, telle la grammaire suivante :

$$S \to aSBC$$
$$S \to aBC$$
$$Ca \to aC$$
$$Ba \to aB$$
$$CB \to BC$$
$$aB \to ab$$
$$bB \to bb$$
$$bC \to bc$$
$$cC \to cc$$

Si l'on considère une grammaire algébrique $G = (N, T, P, S)$, ses productions sont nécessairement de la forme $A \to \alpha$. Précisons la définition du *langage engendré* par G que nous avons donnée ci-dessus. Le langage engendré par G est l'ensemble des suites finies w de terminaux que l'on peut dériver de S par la méthode suivante, où α est ce qui a été dérivé jusqu'alors (initialement, $\alpha = S$) :

1. si α est une suite w composée uniquement de terminaux, alors $w \in \mathcal{L}(G)$;
2. sinon $\alpha = \beta A \gamma$ où A est un non terminal : considérons une production $A \to \delta$ et remplaçons α par $\beta\delta\gamma$, puis recommencer en 1.

Une étape de cette dérivation qui change α en β est notée $\alpha \Rightarrow \beta$. La fermeture transitive de cette opération (vue comme une relation entre α et β) est notée $\_ \Rightarrow^* \_$.

Exemple La grammaire des expressions arithmétiques peut s'écrire :

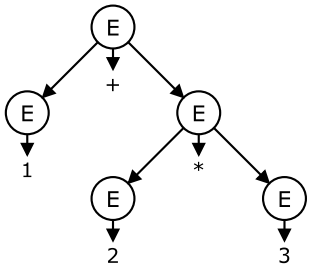
$$\begin{array}{lcl} E & \to & E + E \\ E & \to & E - E \\ E & \to & E * E \\ E & \to & E / E \\ E & \to & (E) \\ E & \to & \texttt{INT} \end{array}$$

où les terminaux sont les constantes entières littérales (symbolisées par le lexème `INT`), les parenthèses et les quatre opérateurs habituels. Le seul non terminal est E et c'est aussi le symbole de départ.

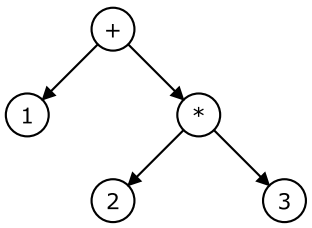
Une expression arithmétique comme $1 + 2 * 3$, représentée par comme $\texttt{INT}_1 + \texttt{INT}_2 * \texttt{INT}_3$, peut être dérivée de plusieurs manières selon que l'on utilise d'abord la production concernant l'addition ou bien celle concernant la multiplication :

$$\begin{array}{l} E \Rightarrow \underline{E} + E \Rightarrow \texttt{INT}_1 + \underline{E} \Rightarrow \texttt{INT}_1 + \underline{E} * E \Rightarrow \texttt{INT}_1 + \texttt{INT}_2 * \underline{E} \Rightarrow \texttt{INT}_1 + \texttt{INT}_2 * \texttt{INT}_3 \\ E \Rightarrow \underline{E} * E \Rightarrow \underline{E} + E * E \Rightarrow \texttt{INT}_1 + \underline{E} * E \Rightarrow \texttt{INT}_1 + \texttt{INT}_2 * \underline{E} \Rightarrow \texttt{INT}_1 + \texttt{INT}_2 * \texttt{INT}_3 \\ E \Rightarrow E + \underline{E} \Rightarrow E + E * \underline{E} \Rightarrow E + \underline{E} * \texttt{INT}_3 \Rightarrow \underline{E} + \texttt{INT}_2 * \texttt{INT}_3 \Rightarrow \texttt{INT}_1 + \texttt{INT}_2 * \texttt{INT}_3 \end{array}$$

Ces trois façons de reconnaître cette expression arithmétique donnent deux résultats différents selon les priorités relatives que l'on attribue à l'addition et la multiplication : la première dérivation a considéré qu'une expression était une somme de produits, et que la chaîne d'entrée devait être lue comme $1+(2*3)$, alors que la seconde dérivation l'a lue comme $(1 + 2) * 3$. Graphiquement, la première dérivation peut être représentée par l'arbre suivant :



dont les nœuds sont des instances de productions de la grammaire. Un tel arbre est appelé *parse tree* en anglais. Cet arbre amène naturellement à la production d'une représentation arborescente – appelée *arbre de syntaxe abstraite* – de l'expression originale qui sera de la forme :



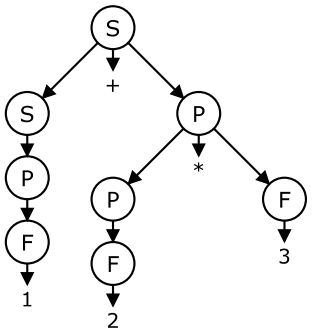
Les deux premières dérivations sont radicalement différentes puisqu'elles donnaient des expressions représentant des valeurs numériques différentes. Toutefois, elles avaient en commun le fait de toujours dériver à chaque étape le non terminal le plus à gauche : on les appelle des *dérivations gauches*. La dernière, quant à elle, utilise les mêmes priorités d'opérateurs que la première, mais a choisi de dériver à chaque étape le non terminal le plus à droite. En fait, l'ordre dans lequel on opère les dérivations (à droite ou à gauche) n'a pas beaucoup d'importance : cela ne change que l'ordre dans lequel on construit le *parse tree*. Et quelque soit cet ordre, on obtient l'un ou l'autre des arbres correspondants à $1 + (2 * 3)$ et à $(1 + 2) * 3$ pour la reconnaissance de $1 + 2 * 3$ avec la grammaire ci-dessus. On dit, pour cette raison, que cette grammaire est *ambiguë*.

Désambiguation de la grammaire des expressions arithmétiques Nous souhaitons donner une grammaire non ambiguë au langage d'expressions arithmétiques sans pour autant changer le langage. Il nous faut transformer notre grammaire en une autre, qui intègre d'une part les priorités des opérateurs multiplicatifs sur les opérateurs additifs, mais aussi l'association à gauche des opérateurs, de sorte que $1 - 2 - 3$ soit correctement reconnu comme $(1 - 2) - 3$.

Les priorités relatives des opérateurs peuvent se représenter en hiérarchisant la grammaire, et en l'exprimant comme ceci : une expression arithmétique est une somme (ou différence) de produits, dont les facteurs sont des entiers ou des expressions entre parenthèses. L'association à gauche des opérateurs se représente par une récursion gauche des non terminaux correspondants :

$$
\begin{array}{lcl}
S & \rightarrow & S \; + \; P \\
S & \rightarrow & S \; - \; P \\
S & \rightarrow & P \\
P & \rightarrow & P \; * \; F \\
P & \rightarrow & P \; / \; F \\
P & \rightarrow & F \\
F & \rightarrow & (\; S \;) \\
F & \rightarrow & \texttt{INT}
\end{array}
$$

Avec cette nouvelle grammaire, le *parse tree* correspondant à $1 + 2 * 3$ est unique, de la forme :



conformément aux priorités habituelles des opérateurs arithmétiques.

3.2 Analyse descendante

Il est tentant de chercher à interpréter un non terminal comme une fonction recevant en entrée un flux de lexèmes et construisant la représentation de l'expression arithmétique en résultat. On voit aisément que cela revient à chercher à construire le *parse tree* en commençant par la racine. La consommation effective des lexèmes du flux d'entrée a lieu lorsque l'on exécute une fonction associée à un non terminal. Ainsi, l'interprétation du non terminal F consomme (si c'est possible) une parenthèse ouvrante, appelle ensuite la fonction correspondant à S, puis ensuite cherche à consommer une parenthèse fermante.

On voit rapidement que cette interprétation ne peut fonctionner correctement en présence de non terminaux récursifs à gauche : l'interprétation de S commence par s'appeler elle-même sans rien consommer, conduisant irrémédiablement à une boucle infinie.

Une solution consiste à transformer de nouveau la grammaire pour en produire une version qui ne soit pas récursive à gauche, et qui ne va donc pas considérer naturellement les opérations comme associant à gauche.

Pour ce faire, on utilise la méthode suivante. On commence par regrouper les productions récursives à gauche de la manière suivante (pour toute production S récursive à gauche) :

$$S \rightarrow S\,\alpha_1 \mid S\,\alpha_2 \mid ... \mid S\alpha_n \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_m$$

où aucun des β_i ne contient S. Ensuite, on remplace chacune des productions S par :

$$\begin{array}{lcl} S & \rightarrow & \beta_1\,S' \mid \beta_2\,S' \mid ... \mid \beta_m\,S' \\ S' & \rightarrow & \alpha_1\,S' \mid \alpha_2\,S' \mid ... \mid \alpha_n\,S' \mid \epsilon \end{array}$$

Cette méthode appliquée à notre grammaire précédente nous permet de dériver une nouvelle version de celle-ci n'étant plus récursive à gauche, et donc maintenant adaptée à une analyseur descendant.

$$\begin{array}{lcl} S & \rightarrow & P\ +\ S \\ S & \rightarrow & P\ -\ S \\ S & \rightarrow & P \\ P & \rightarrow & F\ *\ P \\ P & \rightarrow & F\ /\ P \\ P & \rightarrow & F \\ F & \rightarrow & (\ S\ ) \\ F & \rightarrow & \texttt{INT} \end{array}$$

La lecture de ces non terminaux comme des fonctions d'analyse, s'appelant les unes les autres et consommant des lexèmes dans le flux d'entrée devient naturelle. Illustrons ce principe en développant un analyseur complet, lexical et syntaxique pour la grammaire. Il convient de commencer par ce qui a trait à l'analse lexicale. Nous définissons donc le type des lexèmes dans lequel `EOF` dénote la fin de fichier (ou du flux d'entrée).

```
(* Lèxemes. *)
type lexem =
  | EOF             (* Fin de source. *)
  | INT of int      (* Constante entière (sur 1 chiffre ). *)
  | PLUS | MINUS    (* Addition, soustraction . *)
  | STAR | SLASH    (* Multiplication , division . *)
  | LPAREN | RPAREN (* Parenthèses, ouvrante et fermante. *)
;;
```

L'analyseur lexical est simplifié à l'extrême et ne reconnaît que des nombres composés d'un seul chiffre (pour reconnaître des constantes entière arbitraires, il suffit de l'étendre, ce qui est un travail du ressort du chapitre précédent). Le flux d'entrée est également simplifié et représenté par une chaîne de caractères. L'état interne de l'analyseur permet de mémoriser le dernier lexème reconnu (le lexème courant) et la position, dans la chaîne, à laquelle il faudra continuer l'analyse lors d'une prochaine demande de reconnaissance de lexème. Notons que la taille de la chaîne est également mémorisée uniquement éviter de la recalculer à chaque fois (optimisation).

```
(* État interne d'un analyseur lexical. *)
type lexer = {
  data : string;          (* La chaîne du source à analyser. *)
  datalen : int;          (* Longueur de la chaîne. *)
  mutable index : int;    (* Indice de la prochaine consommation. *)
  mutable curr_lexem : lexem (* Dernier lexème reconnu. *)
} ;;
```

Viennent ensuite deux exceptions pour signaler des erreurs lexicales ou syntaxiques. Leur contenu est minimaliste : dans un véritable compilateur, il faudrait au moins conserver trace de la position où l'erreur s'est produite dans le code source du programme.

```
exception LexError of char ;;  (* Erreur lexicale. *)
exception ParseErr ;;          (* Erreur syntaxique. *)
```

L'analyse ayant pour objectif de construire un arbre de syntaxe superficielle, il est nécessaire de définir le type représentant cet arbre. Sans surprise, il suit la grammaire initiale de nos expressions, c'est à dire la grammaire avant désambiguation et élimination de la récursion à gauche.

```
(* Arbre de syntaxe des expressions arithmétiques. *)
type expr =
  | E_INT of int
  | E_PLUS of (expr * expr) | E_MINUS of (expr * expr)
  | E_STAR of (expr * expr) | E_SLASH of (expr * expr)
  | E_PAREN of expr (* On décide de conserver les () dans l'AST. *)
;;
```

Nous pouvons désormais écrire l'analyseur lexical. Son travail consiste à lire le caractère se trouvant dans la chaîne à l'indice courant, convertir ce caractère en valeur de lexème, mémoriser cette dernière et incrémenter l'indice courant.

```
(* Reconnaît le prochain lexème dans le code source, ou échoue. *)
let next_lexem lexer =
  (* Convertir la suite de caractères en lexème. *)
  let tok =
    if lexer.index ≥lexer.datalen then EOF
    else
      let c = lexer.data.[lexer.index] in
      (* Avance le curseur pour un prochain appel. *)
      lexer.index ←lexer.index + 1 ;
      match c with
      | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' →
          INT ((Char.code c) − (Char.code '0'))
      | '+' → PLUS | '−' → MINUS
```

```
      | '*' → STAR | '/' → SLASH
      | '(' → LPAREN | ')' → RPAREN
      | _ → raise (LexError c) in
  (* La mémoriser dans l'état interne du lexer. *)
  lexer.curr_lexem ←tok
;;
```

L'analyseur syntaxique est composé d'une fonction par non terminal, donc de 3 fonctions mutuellement récursives : `parse_S`, `parse_P` et `parse_F`. Lorsqu'une production contient un non terminal en partie droite, un appel à la fonction correspondant à ce non terminal est effectué. Lorsqu'un terminal figure en partie droite, il faut s'assurer que le lexème courant est égal à ce terminal et, le cas échéant, demander un nouveau lexème à l'analyseur lexical. Chaque fonction a donc la structure d'un filtrage avec un cas par production du non terminal associé à cette fonction.

```
(* Productions de S. *)
let rec parse_S lexer =
  let left = parse_P lexer in
  match lexer.curr_lexem with
  | PLUS → next_lexem lexer ; E_PLUS (left, parse_S lexer)
  | MINUS → next_lexem lexer ; E_MINUS (left, parse_S lexer)
  | _ → left        (* Production S → P. *)

(* Productions de P. *)
and parse_P lexer =
  let left = parse_F lexer in
  match lexer.curr_lexem with
  | STAR → next_lexem lexer ; E_STAR (left, parse_P lexer)
  | SLASH → next_lexem lexer ; E_SLASH (left, parse_P lexer)
  | _ → left        (* Production P → F. *)

(* Productions de F. *)
and parse_F lexer =
  match lexer.curr_lexem with
  | LPAREN → (
      next_lexem lexer ;
      let mid = parse_S lexer in
      match lexer.curr_lexem with
      | RPAREN →next_lexem lexer ; E_PAREN mid
      | _ → raise ParseErr
    )
  | INT i → next_lexem lexer ; E_INT i
  | _ → raise ParseErr
;;
```

Il ne reste finalement plus qu'à écrire une fonction principale qui fabrique un état interne initial d'analyseur lexical, appelle ce dernier une première fois, puis appelle la fonction correspondant au symbole de départ de la grammaire.

```
(* Point d'entrée. *)
let main str =
  (* Initialisation de l'analyseur lexical. *)
  let lexer = {
    data = str; datalen = String.length str; index = 0; curr_lexem = EOF } in
  (* Récupération du premier lexème. *)
  next_lexem lexer;
  (* Appel de la règle correspondnt au symbole de départ. *)
  parse_S lexer
;;
```

Il est possible de tester notre analyse en fournissant une chaîne à la fonction `main`. On notera que l'analyse respecte effectivement la priorité des opérateurs, comme encodé dans la grammaire après désambiguation.

```
# main "1+2" ;;
- : expr = E_PLUS (E_INT 1, E_INT 2)
# main "1+2*3" ;;
- : expr = E_PLUS (E_INT 1, E_STAR (E_INT 2, E_INT 3))
# main "1*2+3" ;;
- : expr = E_PLUS (E_STAR (E_INT 1, E_INT 2), E_INT 3)
# main "1*(2" ;;
Exception: ParseErr.
Raised at file "aexpr_ll.ml", line 77, characters 19-27
```

La programmation directe d'analyseurs descendants n'est utilisable que pour de petits langages. On utilise plutôt des générateurs d'analyseurs descendants pour des grammaires appartenant à une classe nommée LL (*Left-to-right scan*, *Leftmost derivation*) et qui produisent un automate (à pile) représenté par des tables. Toutefois, les générateurs LL sont rares, car leur utilisation nécessite de profondes modifications des grammaires de sorte à éliminer la récursion gauche.

3.3 Analyse ascendante

Les techniques ascendantes, et plus précisément les techniques LR (*Left-to-right scan*, *Rightmost derivation*) sont quant à elles utilisées avec beaucoup de succès pour la génération automatique d'analyseurs.

L'analyse ascendante va lire des lexèmes en les empilant (*shift*). Lorsqu'une règle peut être réduite, les éléments qui constituent sa partie droite sont dépilés, et remplacés par le non terminal en partie gauche de la règle (*reduce*).

Ces générateurs analysent la grammaire qui leur est présentée, et peuvent se plaindre de conflits *shift/reduce* : cela signifie qu'il existera des cas d'entrée où le moteur d'analyse aura le choix entre deux actions possibles : empiler ou réduire. Les générateurs du style Yacc (OCamlyacc, par exemple) choisiront toujours d'empiler (*shift*).

Je ne décris pas ici plus avant les techniques de génération ou d'analyse LR. Pour en savoir plus sur le sujet, se référer aux ouvrages classiques sur la compilation.

3.4 Ocamlyacc

OCamlyacc est un générateur d'analyseurs syntaxiques auquel on donne essentiellement une grammaire dans un fichier de suffixe `.mly`, et qui produit un analyseur dans un fichier de même nom avec pour suffixe `.ml`. Comme pour les analyseurs lexicaux, il existe des outils équivalents pour produire des analyseurs dans d'autres langages : Yacc ou Bison pour C, Jacc ou Beaver pour Java, PLY pour Python, etc. pour n'en citer qu'une minorité.

```
%{
  header (* avec commentaires OCaml *)
%}
  declarations /* commentaires à la C */
%%
  rules /* commentaires à la C */
%%
  trailer (* avec commentaires OCaml *)
```

Les prélude `header` et postlude `trailer` sont du code OCaml qui sera recopié tel quel dans le fichier résultat.

La partie « `déclarations` » sert d'une part à déclarer les lexèmes, qui vont appartenir à un type appelé `token`, qui sera défini automatiquement dans le fichier `.mli` de l'analyseur syntaxique. Les valeurs de ce type seront celles retournée par l'analyseur lexical. D'autre part, cette partie sert aussi à donner les précédences et règles d'associativité des différents lexèmes. La partie « `déclarations` » se termine par « %% ».

La précédence d'un lexème (représentant majoritairement un opérateur) représente sa priorité par rapport aux autres. Les lexèmes sont déclarés en ordre de précédence croissant. Ainsi, `PLUS` et `MINUS` ayant des précédences plus faibles que `DIV`, l'expression $1 + 2/3 - 4$ donnera la priorité à / par rapport à + et sera lue comme $1 + (2/3) - 4$. Des lexèmes déclarés sur la même ligne ont la même précédence.

L'associativité représente quant à elle la manière de parenthéser une expression dans laquelle un opérateur binaire apparaît plusieurs fois consécutives. Ainsi, `PLUS` étant déclaré associatif à gauche, l'expression 1 + 2 + 3 sera lue comme ((1 + 2) + 3). Déclarer un opérateur $\bullet$ comme non associatif (%`nonassoc`) revient à interdire son utilisation en multiples occurrences successives. Ainsi, $x \bullet y \bullet z$ serait considéré comme syntaxiquement incorrect. Le pseudo lexème `UMINUS` est un peu particulier et sert à conférer une précédence très forte au moins unaire. En effet, chaque lexème ne peut avoir qu'une seule précédence déclarée. Le symbole − sert à deux expressions : la soustraction et l'opposé. Sa précédence déclarée se rapporte à l'utilisation dans le contexte de la soustraction. Pour gérer le contexte de l'opposé, on déclare un pseudo lexème (jamais retourné par l'analyseur lexical) qui sert à modifier localement la précédence de `MINUS` avec la directive %`prec`.

```
/* File parser.mly */
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token EOL
%left PLUS MINUS              /* lowest precedence */
%left TIMES DIV               /* medium precedence */
%nonassoc UMINUS              /* highest precedence */
%start main                   /* the entry point */
%type <int> main
```

```
%%
main:
    expr EOL                { $1 }
;
expr:
    INT                     { $1 }
  | LPAREN expr RPAREN      { $2 }
  | expr PLUS expr          { $1 + $3 }
  | expr MINUS expr         { $1 - $3 }
  | expr TIMES expr         { $1 * $3 }
  | expr DIV expr           { $1 / $3 }
  | MINUS expr %prec UMINUS { - $2 }
;
```

La partie suivante (`rules`) contient la grammaire proprement dite, où chaque règle est accompagnée d'une expression OCaml écrite entre accolades, dont la valeur sera produite à chaque fois que cette règle sera *réduite*, c'est-à-dire que le non-terminal correspondant aura été effectivement reconnu.

L'analyseur ci-dessus a pour effet d'évaluer au vol les expressions arithmétiques. En particulier, il ne construit pas d'arbre de syntaxe contrairement à celui écrit « à la main » par descente récursive. Dans la majorité des cas, les traitements de compilation sont trop complexes pour être effectués à la volée et la construction explicite d'un arbre de syntaxe est indispensable.

Chapitre 4

Sémantique dénotationnelle

Dans les trois chapitres précédents, nous nous sommes essentiellement intéressés aux aspects syntaxiques des langages de programmation. Dans ce chapitre et dans les suivants, nous nous intéresserons plutôt au *sens* des programmes, et tiendrons pour résolues leurs difficultés d'ordre syntaxique. En particulier, lorsque nous parlerons des constructions syntaxiques des programmes, nous aurons en tête les *arbres syntaxiques* correspondants plutôt que les suites de caractères avec lesquels ils ont été écrits.

Le sens d'un programme est avant tout son comportement, ce qu'il calcule et comment il le calcule. Lorsque nous écrivons des programmes, nous avons une idée (plus ou moins) précise de ce qu'il va faire, du sens qu'il a. Toute description de l'action d'un programme repose nécessairement sur le sens précis des différentes constructions syntaxiques du langage dans lequel est écrit ce programme.

Écrire un interprète pour un langage de programmation, c'est exprimer le sens de ce langage par le biais d'un programme particulier (écrit éventuellement dans un autre langage). Écrire un compilateur pour un langage, c'est exprimer le sens du langage source en fonction de celui du langage cible (assembleur, langage intermédiaire ou langage de haut niveau). La fidélité de la traduction repose aussi bien sûr sur la qualité du compilateur, et donc sur le langage dans lequel celui-ci est écrit.

Ces considérations n'ont pas d'autre but que de montrer que la précision de la description d'un langage repose bien sûr sur ce qu'on décrit, mais aussi sur le langage que l'on utilise pour exprimer cette description. Nous utiliserons dans la suite des descriptions formelles du sens des langages de programmation. Lorsque nous voudrons effectuer des implémentations de ces descriptions, nous passerons alors d'un langage mathématique à un langage informatique.

4.1 Le sens du sens

Une sémantique cohérente, précise et non-ambiguë, a de nombreuses utilités. Elle peut être utilisée comme une définition du langage dans un but de standardisation, par exemple. Elle peut aussi bien sûr servir de référence aux programmeurs, qui ne souhaitent pas avoir à exécuter un programme pour le comprendre. La plupart des actions qui impliquent un raisonnement mathématique à propos des programmes nécessitent une sémantique formelle du langage. Les implémenteurs d'interprètes ou de compilateurs utilisent la sémantique du langage comme référence dans le but d'éviter la construction d'implémentations divergentes du même langage.

Certains langages peuvent être implantés directement à partir d'une sémantique formelle, qui est alors dite *exécutable*. Enfin, la possibilité d'une sémantique formelle pour un langage est probablement le gage de qualités telles la simplicité ou l'orthogonalité des concepts.

4.2 Sémantiques

Nous allons, dans ce chapitre et le suivant, distinguer deux façons assez différentes d'exprimer le sens d'un langage. Dans ce chapitre, nous verrons comment exprimer *ce que représente un programme* : c'est ce qu'on appelle sa *dénotation*, et il s'agit d'un objet mathématique. Au chapitre suivant, nous nous attacherons à décrire le *comportement des programmes*, plutôt que leur essence.

Nous n'aborderons pas ici une façon particulière de décrire le sens des programmes qui est connue sous le nom de *sémantique axiomatique*. En sémantique axiomatique, le sens d'un programme est donné en fonction de son impact sur certaines propriétés. Typiquement, dans la sémantique axiomatique d'un programme S, on écrirait $\{p\}S\{q\}$ pour indiquer que si la propriété p était vraie avant l'exécution de S, alors la propriété q est vraie ensuite. La propriété p est appelée *pré-condition*, alors que q est appelée *post-condition*. Ce type de description est utile lorsque l'on veut prouver des propriétés de correction partielle ou totale (c'est-à-dire incluant la terminaison) de programmes. Elle semble moins utilisable pour produire une définition de langage de programmation, et nous n'en parlerons pas plus dans ce cours.

4.3 Le langage PCF

Afin que nos descriptions sémantiques ne restent pas abstraites, nous allons utiliser un mini langage de programmation, qui n'est composé que de quelques constructions syntaxiques, mais qui a assez de puissance d'expression pour exprimer des algorithmes arbitraires. Nous appelons PCF [1] ce langage. On le retrouve quelquefois dans la littérature sous le nom de « Mini-ML ».

La présentation que nous en donnons ci-dessous reste assez abstraite, au sens où nous ne précisons pas quelles sont les constantes et fonctions primitives du langage, et nous restons assez vagues quant à sa syntaxe concrète.

Syntaxe de PCF La syntaxe du langage est donnée sous la forme d'une grammaire. Les catégories lexicales Const et Id correspondent aux constantes (entières, booléennes, *etc.*), mais incluent aussi les primitives telles les opérations arithmétiques ou la concaténation de chaînes de caractères. Le langage PCF est un langage d'expressions (tout comme le langage OCaml) et on notera Exp la catégorie syntaxique (le non terminal) correspondante. On notera par la suite les éléments de ces différentes catégories par des méta-variables particulières (e, e_1, e_2, *etc.* pour les expressions, par exemple).

$c \in$ Const	Constantes, incluant les fonctions primitives
$x \in$ Id	Identificateurs
$e \in$ Exp	Expressions

Les expressions de PCF sont composées de constantes, primitives, de liaisons locales et de fonctions.

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid e_1 e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
& \mid & e_1 + e_2 \mid e_1 = e_2 \\
& \mid & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
& \mid & \mathsf{let\ rec}\ x = e_1\ \mathsf{in}\ e_2 \\
& \mid & \mathsf{fun}\, x \rightarrow e
\end{array}
$$

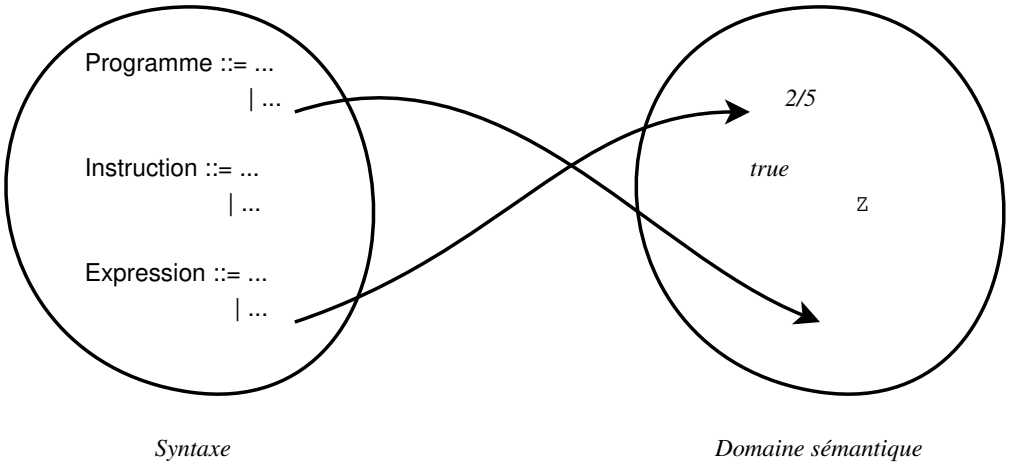
On reconnaît là le noyau des langages fonctionnels comme OCaml. Un exemple de programme dans ce langage pourrait être :

$$\mathsf{let\ rec}\ \mathrm{fact} = \mathsf{fun}\, n \rightarrow \mathsf{if}\ n \leq 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * \mathrm{fact}(n-1)\ \mathsf{in}$$
$$\mathrm{fact}(5)$$

1. *Programming language for Computable Functions* : un langage introduit par Gordon Plotkin en 1977 pour étudier les relations entre les sémantiques que nous présentons dans ce chapitre et le suivant.

4.4 Sémantique dénotationnelle de PCF

La sémantique dénotationnelle d'un langage de programmation consiste à définir ce qu'est la *dénotation* des programmes de ce langage. La dénotation d'un programme est un objet mathématique, donc abstrait, mais qui est précisément défini. La sémantique dénotationnelle voit généralement un programme comme une fonction (au sens mathématique du terme) des entrées vers les sorties, par exemple, ou, plus généralement, d'un *domaine* dans un autre. Les définitions de sémantique des langages sont en général *compositionnelles*, c'est-à-dire qu'elles définissent la sémantique d'un programme en fonction des sémantiques des éléments qui le constituent. En général, la sémantique est définie inductivement, en suivant la structure du langage (ses arbres syntaxiques).



4.4.1 Domaines

Nous serons dans ce chapitre assez approximatifs dans la construction des domaines mentionnés ci-dessus. En particulier, nous construisons ci-dessous un domaine qui contient l'espace des fonctions du domaine vers lui-même. Si nous voulions être précis, nous indiquerions que ces domaines sont composés d'éléments dont certains représentent l'indéfini (pour représenter par exemple le « résultat » d'un calcul qui ne termine pas). Les éléments de nos domaines sont donc partiellement indéfinis, ce qui permet de doter chacun de ces domaines d'un ordre partiel (« moins défini que »), et les domaines sont construits de telle sorte que chacun contienne un élément indéfini (noté $\perp$, inférieur à tous les autres) et que toute chaîne strictement croissante ait une borne supérieure. Cela dote les domaines considérés d'une structure dite d'*ordre partiel complet*, ou CPO (pour *Complete partial Order*. Les fonctions que l'on considérera ici sont des fonctions monotones f telles que pour toute chaîne C, on ait $\mathsf{sup}(f(C)) = f(\mathsf{sup}(C))$. Par souci de simplicité, nous ignorerons par la suite cet aspect des choses.

Pour le langage PCF, nous construisons un domaine D, récursivement, à partir des domaines pour les constantes, et contenant les fonctions du domaine vers lui-même.

- INT : le domaine des entiers
- BOOL = { *true*, *false* }
- D = INT + BOOL + FUN : le domaine des valeurs
- FUN = D $\rightarrow$ D : les fonctions sur D

Nous aurons besoin, pour la construction des fonctions sémantiques, d'un domaine noté ENV, qui contiendra des fonctions associant des valeurs de D à des identificateurs de Id :

- ENV = Id $\rightarrow$ D : le domaine des environnements

Si ρ est un environnement, $\rho' = \rho \oplus [x \mapsto d]$ est l'environnement défini par :

$$\rho'[\![x]\!] = d$$

$\rho'[\![y]\!] = \rho[\![y]\!]$, pour $y \neq x$

Ci-dessus, et par la suite, lorsque qu'une fonction reçoit un argument de nature syntaxique (un identificateur ou une expression, par exemple), on utilise des parenthèses spéciales $[\![\_]\!]$ pour bien marquer ce fait.

4.4.2 Fonctions sémantiques

Le langage PCF tel qu'il est décrit ici nécessite essentiellement une seule fonction sémantique, qui donne le sens dans D d'une expression e. Une expression e contenant des identificateurs *libres*, c'est-à-dire sans le let ou le fun qui les introduit, la sémantique d'une expression est, en toute généralité, paramétrée par celle de ses variables libres. La fonction sémantique des expressions, notée $\mathcal{E}$, a la forme suivante :

$$\mathcal{E} : \text{Exp} \to \text{ENV} \to \text{D}$$

La définition de $\mathcal{E}$ s'écrit par cas selon la structure de l'expression considérée :

$\mathcal{E}[\![c]\!]\rho = d_c$ pour chaque $c \in$ Const, où d_c est la valeur de D dénotée par c
$\mathcal{E}[\![x]\!]\rho = \rho[\![x]\!]$, pour $x \in$ Id
$\mathcal{E}[\![e_1 + e_2]\!]\rho = \mathcal{E}[\![e_1]\!]\rho + \mathcal{E}[\![e_2]\!]\rho$
$\mathcal{E}[\![e_1\ e_2]\!]\rho = (\mathcal{E}[\![e_1]\!]\rho)(\mathcal{E}[\![e_2]\!]\rho)$
$\mathcal{E}[\![e_1 = e_2]\!]\rho = \mathit{true}$ si $\mathcal{E}[\![e_1]\!]\rho = \mathcal{E}[\![e_2]\!]\rho$
$= \mathit{false}$ dans le cas contraire
$\mathcal{E}[\![\textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3]\!]\rho =$
si $\mathcal{E}[\![e_1]\!]\rho = \mathit{true}$ alors $\mathcal{E}[\![e_2]\!]\rho$ sinon $\mathcal{E}[\![e_3]\!]\rho$
$\mathcal{E}[\![\textsf{let } x = e_1 \textsf{ in } e_2]\!]\rho = \mathcal{E}[\![e_2]\!](\rho \oplus [x \mapsto \mathcal{E}[\![e_1]\!]\rho])$
$\mathcal{E}[\![\textsf{let rec } f = e_1 \textsf{ in } e_2]\!]\rho = \mathcal{E}[\![e_2]\!]\rho'$
où $\rho' = \rho \oplus [f \mapsto \mathcal{E}[\![e_1]\!]\rho']$ *(Noter que ρ' est ici défini récursivement)*
$\mathcal{E}[\![\textsf{fun } x \to e]\!]\rho = f$ telle que $f(v) = \mathcal{E}[\![e]\!](\rho \oplus [x \mapsto v])$

Le seul cas un peu complexe de cette définition est le cas des définitions récursives, que l'on fait correspondre à des valeurs qui utilisent un environnement défini récursivement. Sans entrer dans le détail, c'est la structure de CPO des domaines que nous considérons qui permet d'écrire une telle définition.

4.5 Sémantique dénotationnelle des affectations

Si maintenant nous enrichissons notre langage avec une mémoire et des instructions modifiant cette mémoire, nous obtenons de nouveaux domaines et devons définir les fonctions sémantiques correspondantes. Nous considérerons ici que toutes les variables introduites par **let** sont modifiables (importante différence avec le langage précédent).

Syntaxe

$e \in$ Exp, défini par $e ::= \ldots \mid s; e$
$s \in$ Stm, défini par $s ::= x := e \mid s_1; s_2$

La catégorie Stm est celle des instructions : affectation et séquence d'instructions. Les expressions de Exp peuvent maintenant être précédées d'une instruction (ou d'une séquence d'instructions).

Domaines sémantiques Les domaines sémantiques doivent, dès lors, représenter un peu plus finement les variables. En effet, puisque celles-ci sont devenues modifiables, il est nécessaire de distinguer leur adresse (à laquelle il est fait référence dans la partie gauche d'une affectation) de leur valeur (qui est la valeur stockée à cette adresse). À un nom de variable, on associe donc désormais une référence

ou adresse (*location* en anglais). La « valeur » d'une variable pourra ensuite être obtenue depuis cette référence.

Deux nouveaux domaines apparaissent donc : LOC, pour les référence, et STORE qui associe une valeur à une référence. Le domaine D, quant à lui, ne change pas en apparence, mais le sous-domaine FUN change de nature : puisque l'application d'une fonction produit un résultat et une modification de la mémoire, les fonction reçoivent en argument supplémentaire une mémoire, et produisent un couple composé d'un résultat et d'une nouvelle mémoire.

- D = INT + BOOL + FUN
- FUN = (D × STORE) → (D × STORE)
- LOC : le domaine des références (*locations*)
- STORE = LOC → D : le domaine des références initialisées (on parlera de « nouvelles » références pour mentionner des références inutilisées)
  ENV = ID → LOC

On voit clairement que l'accès à la valeur d'une variable par ENV procédera en deux temps : d'abord trouver la référence associée à la variable (son adresse), puis accéder à la valeur stockée à cet endroit dans la mémoire (STORE).

Fonctions sémantiques À la fonction sémantique $\mathcal{E}$ des expressions, s'ajoute maintenant une fonction sémantique $\mathcal{S}$ pour les instructions. Les expressions pouvant contenir des instructions, et les instructions contenant des expressions (les affectations), nous allons donc utiliser ces deux fonctions de façon mutuellement récursive :

$\mathcal{E}$: Exp → ENV → STORE → (D × STORE)
$\mathcal{S}$: Stm → ENV → STORE → STORE

Les instructions, et donc aussi les expressions, peuvent modifier la mémoire : c'est pourquoi les deux fonctions sémantiques $\mathcal{E}$ et $\mathcal{S}$ reçoivent un argument de type STORE. Puisque les instructions ne produisent pas de valeur, seule une mémoire est produite en résultat. Les expressions, quant à elles, produisent à la fois une valeur (de D) et une mémoire : c'est pourquoi la fonction sémantique $\mathcal{E}$ produit un couple de (D × STORE) en résultat. Les catégories syntaxiques Exp et Stm étant définies de façon mutuellement récursive, il est naturel que les fonctions sémantiques $\mathcal{E}$ et $\mathcal{S}$ le soient aussi.

Définitions de $\mathcal{E}$ et $\mathcal{S}$

$\mathcal{E}[\![c]\!]\rho\sigma = (d_c, \sigma)$ pour chaque $c \in$ Const, où d_c est la valeur représentant c dans D

$\mathcal{E}[\![x]\!]\rho\sigma = (\sigma(\rho[\![x]\!]), \sigma)$, pour $x \in$ Id

$\mathcal{E}[\![e_1 + e_2]\!]\rho\sigma =$
  soit $(n_1, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$
  soit $(n_2, \sigma_2) = \mathcal{E}[\![e_2]\!]\rho\sigma_1$
  $(n_1 + n_2, \sigma_2)$

$\mathcal{E}[\![e_1\ e_2]\!]\rho\sigma =$
  soit $(f, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$
  soit $(v, \sigma_2) = \mathcal{E}[\![e_2]\!]\rho\sigma_1$
  $(f(v), \sigma_2)$

$\mathcal{E}[\![e_1 = e_2]\!]\rho\sigma =$
  soit $(v_1, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$
  soit $(v_2, \sigma_2) = \mathcal{E}[\![e_2]\!]\rho\sigma_1$
  si $v_1 = v_2$ alors ($true$, σ_2) sinon ($false$,σ_2)

$\mathcal{E}[\![$if e_1 then e_2 else $e_3]\!]\rho\sigma =$
  soit $(v_1, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$

if $v_1 = \mathit{true}$ then $\mathcal{E}[\![e_2]\!]\rho\sigma_1$ else $\mathcal{E}[\![e_3]\!]\rho\sigma_1$

$\mathcal{E}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma =$
soit $(v_1, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$
soit l_1 une nouvelle référence
$\mathcal{E}[\![e_2]\!](\rho \oplus [x \mapsto l_1])(\sigma_1 \oplus [l_1 \mapsto v_1])$

$\mathcal{E}[\![\mathsf{let\ rec}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma =$
soit l_1 une nouvelle référence
soit $(v_1, \sigma_1) = \mathcal{E}[\![e_1]\!](\rho \oplus [x \mapsto l_1])(\sigma)$
$\mathcal{E}[\![e_2]\!](\rho \oplus [x \mapsto l_1])(\sigma_1 \oplus [l_1 \mapsto v_1])$
Notons que $\sigma(l_1)$ est indéfinie : la raison en est que le calcul de e_1 ne doit pas utiliser x.

$\mathcal{E}[\![\mathsf{fun}\ x\ \rightarrow\ e]\!]\rho\sigma = f$
telle que $f(v, \sigma') =$
soit l_x une nouvelle référence
$\mathcal{E}[\![e]\!](\rho \oplus [x \mapsto l_x])(\sigma' \oplus [l_x \mapsto v])$

$\mathcal{E}[\![s; e]\!]\rho\sigma =$
soit $\sigma' = \mathcal{S}[\![s]\!]\rho\sigma$
$\mathcal{E}[\![e]\!]\rho\sigma'$

$\mathcal{S}[\![x := e]\!]\rho\sigma =$
soit $l_x = \rho[\![x]\!]$
soit $(v, \sigma') = \mathcal{E}[\![e]\!]\rho\sigma$
$\sigma' \oplus [l_x \mapsto v]$

$\mathcal{S}[\![s_1; s_2]\!]\rho\sigma =$
soit $\sigma_1 = \mathcal{S}[\![s_1]\!]\rho\sigma$
$\mathcal{S}[\![s_2]\!]\rho\sigma_1$

Les seuls cas intéressants des définitions ci-dessus sont ceux où on utilise des variables, ou bien pour les déclarer, ou alors pour les modifier. Rappelons qu'ici toutes les variables sont traitées de la même manière : elles représentent toutes des références, et sont, à ce titre, modifiables. Pour avoir des variables qui soient similaires à celles d'OCaml, c'est-à-dire des constantes dont la valeur peut être une structure de données modifiable, il faut d'abord avoir des données structurées dans le langage, représentées par des blocs dans la mémoire (par exemple, en faisant de STORE une collection d'enregistrements) et – à défaut d'un système de types – de marquer certains champs de ces blocs comme modifiables. Cette modification ne pose aucun problème technique.

Il est intéressant de noter que dans notre langage, l'accès à la valeur d'une variable est toujours réalisé dans la mémoire (STORE) et non dans l'environnement (qui sert juste d'indirection pour arriver à la mémoire). De ce fait, les adresses ne sont pas des valeurs du langage, contrairement à C (par exemple) où il est possible de les manipuler par des variables de type pointeur. Pour bénéficier de cette possibilité, il faut rendre l'accès à la mémoire explicite (*déréférencement*) par l'ajout d'une nouvelle construction syntaxique similaire au ! d'OCaml.

4.6 Sémantique dénotationnelle de goto

Tentons maintenant de donner un sens à la fameuse instruction **goto**, qui permet d'interrompre le déroulement d'un programme pour reprendre le calcul dans un autre contexte. La modélisation de ces sauts nécessite d'identifier et de représenter les contextes où on peut ainsi « sauter ». Les destinations de sauts sont identifiées par des étiquettes (*labels*, en anglais).

Une destination de saut est un calcul à reprendre. Nous considérerons ici une forme limitée de **goto**, qui ne permet que de sauter vers un label que depuis l'intérieur de la « portée » de ce label, c'est-à-dire dans la séquence d'instructions qui suit ce label. Un contexte de reprise est donc nécessairement situé entre deux instructions, et nous représenterons un tel contexte par une fonction de type STORE → STORE : on arrive dans ce contexte avec un certain état mémoire, et on reprend une exécution qui produira *in fine* un nouvel état.

Expliciter ainsi *un* contexte de calcul et autoriser à remplacer le calcul courant par un tel nouveau contexte oblige à expliciter *tous* les contextes de calculs : en effet, pour pouvoir remplacer le reste du calcul par un autre que le reste « naturel », il faut que chaque calcul soit paramétré par ce qui reste à faire – la suite, ou la *continuation* –, pour pouvoir changer cette continuation par une autre.

Nos fonctions sémantiques vont donc recevoir un paramètre supplémentaire : la continuation du calcul, ce qui nous amènera de façon naturelle à expliciter l'ordre d'évaluation des différentes sous-expressions de notre langage.

Les continuations vont prendre deux formes, selon qu'il s'agisse de continuations de calculs d'expressions ou de continuations de calculs d'instructions. Les continuations d'instructions appartiendront à Cc (pour *Command continuation*) :

$$\text{Cc} = \text{STORE} \to \text{STORE}$$

et représenteront le reste du calcul après l'exécution d'une instruction. Les continuations d'expressions appartiendront à Ec :

$$\text{Ec} = (\text{D} \times \text{STORE}) \to (\text{D} \times \text{STORE})$$

et représenteront le reste du calcul après celui d'une expression.

Fonctions sémantiques

$\mathcal{E} : \text{Exp} \to \text{ENV} \to \text{STORE} \to \text{Ec} \to (\text{D} \times \text{STORE})$

$\mathcal{S} : \text{Stm} \to \text{ENV} \to \text{STORE} \to \text{Cc} \to \text{STORE}$

Les valeurs fonctionnelles reçoivent elles aussi un paramètre supplémentaire : une continuation. Elles l'utilisent en effet pour la passer à la sémantique de leur corps, lequel l'utilisera comme continuation de la valeur de retour de la fonction.

$$\text{FUN} = (\text{D} \times \text{STORE}) \to \text{Ec} \to (\text{D} \times \text{STORE})$$

Les labels sont de simples identificateurs dont les valeurs (dans la mémoire) sont des continuations d'instructions. On aura donc :

$$\text{D} = \text{INT} + \text{BOOL} + \text{FUN} + \text{Cc}$$

Syntaxe On étend la syntaxe des instructions en :

$s \in \text{Stm}$, et $s ::= x := e \mid s_1; s_2 \mid lab : s \mid \textsf{goto}\ lab$

où

- $lab \in \text{Id}$
- dans "$lab : s$", s est dans la portée de lab, et peut donc contenir "goto lab".

Par souci de simplicité, nous utiliserons ici une version restreinte de **goto**. Des formes plus générales de **goto** nécessiteraient un traitement plus complexe, sans pour autant présenter de réelle nouveauté par rapport à ce que nous présentons ici.

Fonctions sémantiques

$\mathcal{E}[\![c]\!]\rho\sigma k = k\ (d_c, \sigma)$ pour $c \in \text{Const}$, où d_c est la valeur de D correspondant à c

$\mathcal{E}[\![x]\!]\rho\sigma k = k\ (\sigma(\rho[\![x]\!]), \sigma)$, pour $x \in \text{Id}$

$\mathcal{E}[\![e_1 + e_2]\!]\rho\sigma k =$

$\mathcal{E}[\![e_1]\!]\rho\sigma k_1$
où k_1 est la continuation définie par $k_1(n_1, \sigma_1) = \mathcal{E}[\![e_2]\!]\rho\sigma_1(k_2\ n_1)$
où $k_2\ n_1\ (n_2, \sigma_2) = k(n_1 + n_2, \sigma_2)$

$\mathcal{E}[\![e_1\ e_2]\!]\rho\sigma k =$
$\mathcal{E}[\![e_1]\!]\rho\sigma k_1$
où $k_1(v_1, \sigma_1) = \mathcal{E}[\![e_2]\!]\rho\sigma_1(k_2\ v_1)$
où $k_2\ v_1\ (v_2, \sigma_2) = v_1\ (v_2, \sigma_2)\ k$

$\mathcal{E}[\![e_1 = e_2]\!]\rho\sigma k =$ ... laissé en exercice

$\mathcal{E}[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]\rho\sigma k =$ ... laissé en exercice

$\mathcal{E}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma k =$ ... laissé en exercice

$\mathcal{E}[\![\mathsf{let\ rec}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma k =$ ... laissé en exercice

$\mathcal{E}[\![\mathsf{fun}\ x\ \rightarrow\ e]\!]\rho\sigma k =$
$k(f, \sigma)$
où f est la fonction définie par $f\ (v, \sigma')\ k' = \mathcal{E}[\![e]\!](\rho \oplus [x \mapsto l_x])(\sigma' \oplus [l_x \mapsto v])k'$
où l_x est une nouvelle référence

$\mathcal{E}[\![s; e]\!]\rho\sigma k =$
soit k' définie par $k'(\sigma') = \mathcal{E}[\![e]\!]\rho\sigma' k$
$\mathcal{S}[\![s]\!]\rho\sigma k'$

$\mathcal{S}[\![lab : s]\!]\rho\sigma k =$
soit l une nouvelle référence
soit k' définie récursivement par $k'(\sigma') = \mathcal{S}[\![s]\!](\rho \oplus [lab \mapsto l])(\sigma' \oplus [l \mapsto k'])k$
$k'\sigma$

$\mathcal{S}[\![x := e]\!]\rho\sigma k =$
soit k' définie par $k'(v', \sigma') = k(\sigma' \oplus [\rho[\![x]\!] \mapsto v'])$
$\mathcal{E}[\![e]\!]\rho\sigma k'$

$\mathcal{S}[\![s_1; s_2]\!]\rho\sigma k =$
$\mathcal{S}[\![s_1]\!]\rho\sigma k'$
où k' est la continuation définie par $k'(\sigma') = \mathcal{S}[\![s_2]\!]\rho\sigma' k$

$\mathcal{S}[\![\mathsf{goto}\ lab]\!]\rho\sigma k =$
$k'\sigma$
où $k' = \sigma(\rho[\![lab]\!])$

Cela conclut la description de la sémantique dénotationnelle de PCF avec affectations et **goto**. Les sémantiques que nous avons données précédemment, sans continuations, sont appelées *sémantiques directes*.

Chapitre 5

Sémantique opérationnelle

Nous avons vu au chapitre 4 que la sémantique dénotationnelle permettait d'associer à un programme un objet mathématique construit dans un domaine spécifique. Outre l'utilisation d'objets mathématiques pas toujours très simples, la sémantique dénotationnelle a pour inconvénient essentiel de fournir directement un *sens* aux programmes et de ne pas expliciter le *calcul* qui mène à ce sens. Les calculs sont ceux qui sont nécessaires à l'identification de l'image d'un programme par une fonction sémantique, mais n'apparaissent pas comme des *objets à part entière* de la sémantique dénotationnelle, qui considère essentiellement des domaines, valeurs et fonctions sémantiques.

Or, on considère à juste titre que *l'évaluation* ou *l'exécution* d'un programme présente plus d'intérêt que la simple valeur éventuellement produite par ce programme. Il existe en effet bon nombre de programmes intéressants qui ne terminent pas, et qui donc ne produisent par *une* valeur finale : ils ont par contre des *effets* intéressants, comme des opérations d'entrée-sortie ou tout autre effet sur leur environnement. Par ailleurs, en sémantique dénotationnelle, les programmes ont un sens (une valeur de D différente de $\perp$), ou n'en n'ont pas ($\perp$) : on ne peut pas caractériser un programme qui se comporterait correctement jusqu'à un certain point de son exécution, et qui produirait une erreur d'exécution en ce point.

Nous abordons dans ce chapitre un formalisme sémantique qui explicite l'exécution des programmes : la *sémantique opérationnelle*. Présentée initialement par Gordon Plotkin [1] en 1981, la sémantique opérationnelle structurée (SOS) est une sémantique qui indique comment partir d'un programme et le transformer progressivement jusqu'à l'obtention éventuelle d'un état qui soit irréductible, et qui représente donc le résultat du calcul.

En fait, nous opérerons de cette manière sur les termes du λ-calcul au chapitre 10, pour spécifier quelques stratégies de réduction des λ-termes. La forme des règles que nous utilisons ici est légèrement différente de celles que qui seront décrites au chapitre 10 : au lieu de spécifier de petits pas d'évaluation (si e fait un pas d'évaluation vers e', alors ...), nous spécifions ici de *grands pas*, en disant par exemple « si e se réduit en v (irréductible), alors ... ». Cette dernière forme, que nous utilisons dans ce chapitre, est aussi appelée *sémantique naturelle* [2] et a été popularisée par Gilles Kahn au début des années 1980.

Nous reprenons ici le langage PCF tel que nous l'avons défini au chapitre 4, pour lui attribuer des sémantiques opérationnelles. Dans un premier temps, nous ne considérerons que le noyau fonctionnel de PCF pour le doter d'un mécanisme d'évaluation en appel par valeur ainsi que d'une évaluation paresseuse. Nous l'étendons ensuite avec des traits impératifs comme nous l'avons fait au chapitre 4.

5.1 Évaluation stricte du noyau fonctionnel de PCF

Rappelons d'abord la structure des programmes PCF :

1. http://citeseer.ist.psu.edu/plotkin81structural.html
2. Par analogie avec les règles logiques de la déduction naturelle.

$c \in \text{Const}$	Constantes, incluant les fonctions primitives
$x \in \text{Id}$	Identificateurs
$e \in \text{Exp}$	Expressions

Les expressions de PCF sont composées de constantes, primitives, de liaisons locales et de fonctions. Nous limitons ici la construction de valeurs récursives aux fonctions.

$$\begin{aligned} e \quad ::= \quad & c \mid x \mid e_1 e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ \mid \quad & e_1 + e_2 \mid e_1 = e_2 \\ \mid \quad & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ \mid \quad & \mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2 \\ \mid \quad & \mathsf{fun}\, x \ \rightarrow \ e \end{aligned}$$

Tout comme dans le λ-calcul, l'opération essentielle de l'exécution des programmes PCF est l'application de fonction, et, plus précisément, le moment de l'exécution qui correspond à la β-réduction, où on exécute le corps de la fonction en prenant (la valeur de) l'argument comme valeur du paramètre formel de la fonction. Au lieu de transformer les programmes source directement à l'aide d'opérations de substitutions qui sont lourdes, sources d'erreurs, et très éloignées des opérations effectivement réalisées par le code produit par les compilateurs, nous allons utiliser une structure d'environnement pour stocker les valeurs des variables libres des programmes PCF, incluant bien sûr les arguments des fonctions. Ainsi, la valeur d'une fonction sera essentiellement un couple constitué du code du corps de la fonction et d'un environnement qui contiendra les valeurs de ses variables libres. Lorsque cette fonction sera appliquée sur un argument a, on exécutera le code du corps de la fonction dans son environnement auquel on a ajouté a comme valeur du paramètre formel de la fonction.

On définit donc l'ensemble des valeurs résultant d'évaluations correctes comme :

$$v \quad ::= \quad c \mid \mathsf{Valf}(x, e, \rho) \mid \mathsf{Valfr}(f, x, e, \rho)$$

où ρ est un environnement qui associe des valeurs à des identificateurs. Nous noterons $\rho \oplus [x \mapsto v]$ l'environnement ρ étendu par une liaison entre x et v, masquant d'éventuelles autres liaisons pour x dans ρ. Les marques Valf et Valfr indiquent toutes deux des valeurs fonctionnelles, composées d'un nom de paramètre formel x, d'un corps de fonction e et d'un environnement ρ dans le premier cas, avec en plus le nom f de la fonction dans le second. Nous verrons plus tard, que le nom de la fonction nous permettra de retrouver cette même valeur de fonction dans l'environnement lors de l'exécution d'appels récursifs. Ces objets, enfermant dans une même « boîte » une expression (ou plus généralement du code) et un environnement contenant les valeurs des variables libres de l'expression (code) sont appelées *fermetures* [3].

Il se trouve que toutes les évaluations ne produisent pas nécessairement de telles valeurs : certaines peuvent en effet conduire à des erreurs. Nous ajoutons donc un nouvel élément Erreur aux résultats possibles des évaluations, définissant ainsi la catégorie suivante :

$$r \quad ::= \quad v \mid \mathsf{Erreur}$$

Nous pouvons maintenant donner les règles d'évaluation du noyau fonctionnel de PCF. Ce sont des règles d'inférence définissant un jugement de la forme $\rho \vdash e \Rightarrow r$ que l'on lit de la manière suivante : « *dans l'environnement ρ, l'évaluation de l'expression e produit le résultat r* ». Un tel jugement est en fait une relation à 3 places, reliant un environnement ρ, un programme e et un résultat r. Cette relation est définie inductivement par les règles données ci-après. Les règles qui ne sont pas encadrées sont des règles de propagation d'erreurs : elles indiquent généralement que lorsque l'évaluation d'une sous-expression produit une erreur, l'évaluation entière se termine en erreur.

$$\boxed{\rho \vdash c \Rightarrow c\ (\textsc{Const})}$$

$$\boxed{\frac{x \in \mathsf{dom}(\rho)}{\rho \vdash x \Rightarrow \rho(x)}\ (\textsc{Ident})} \qquad \frac{x \notin \mathsf{dom}(\rho)}{\rho \vdash x \Rightarrow \mathsf{Erreur}}\ (\textsc{IdentErr})$$

3. *Closures* en anglais.

$$\boxed{\rho \vdash (\mathsf{fun}\, x \;\to\; e) \Rightarrow \mathsf{Valf}(x, e, \rho)}\ (\textsc{Fun})$$

$$\boxed{\frac{\rho \vdash e_1 \Rightarrow T \qquad \rho \vdash e_2 \Rightarrow r}{\rho \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow r}}\ (\textsc{IfTrue}) \qquad \boxed{\frac{\rho \vdash e_1 \Rightarrow F \qquad \rho \vdash e_3 \Rightarrow r}{\rho \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow r}}\ (\textsc{IfFalse})$$

$$\frac{\rho \vdash e_1 \Rightarrow r \notin \{T, F\}}{\rho \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow \mathsf{Erreur}}\ (\textsc{IfErr})$$

$$\boxed{\frac{\rho \vdash e_1 \Rightarrow n_1 \qquad \rho \vdash e_2 \Rightarrow n_2 \qquad n = n_1 + n_2}{\rho \vdash (e_1 + e_2) \Rightarrow n}}\ (\textsc{Plus})$$

$$\frac{\rho \vdash e_1 \Rightarrow r \notin \mathbb{N}}{\rho \vdash (e_1 + e_2) \Rightarrow \mathsf{Erreur}}\ (\textsc{PlusErrL}) \qquad \frac{\rho \vdash e_1 \Rightarrow n_1 \qquad \rho \vdash e_2 \Rightarrow r \notin \mathbb{N}}{\rho \vdash (e_1 + e_2) \Rightarrow \mathsf{Erreur}}\ (\textsc{PlusErrR})$$

$$\boxed{\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad v_1 = v_2}{\rho \vdash (e_1 = e_2) \Rightarrow T}}\ (\textsc{EgTrue})$$

$$\boxed{\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad v_1 \neq v_2}{\rho \vdash (e_1 = e_2) \Rightarrow F}}\ (\textsc{EgFalse})$$

$$\frac{\rho \vdash e_1 \Rightarrow \mathsf{Erreur}}{\rho \vdash (e_1 = e_2) \Rightarrow \mathsf{Erreur}}\ (\textsc{EgErrL}) \qquad \frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow \mathsf{Erreur}}{\rho \vdash (e_1 = e_2) \Rightarrow F}\ (\textsc{EgErrR})$$

$$\boxed{\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \oplus [x \mapsto v_1] \vdash e_2 \Rightarrow r}{\rho \vdash (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \Rightarrow r}}\ (\textsc{Let}) \qquad \frac{\rho \vdash e_1 \Rightarrow \mathsf{Erreur}}{\rho \vdash (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \Rightarrow \mathsf{Erreur}}\ (\textsc{LetErr})$$

$$\boxed{\frac{\rho \oplus [f \mapsto \mathsf{Valfr}(f, x, e_1, \rho)] \vdash e_2 \Rightarrow r}{\rho \vdash (\mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2) \Rightarrow r}}\ (\textsc{Letrec})$$

$$\boxed{\frac{\rho \vdash e_1 \Rightarrow \mathsf{Valf}(x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho_0 \oplus [x \mapsto v_2] \vdash e_0 \Rightarrow r}{\rho \vdash (e_1 e_2) \Rightarrow r}}\ (\textsc{AppF})$$

$$\boxed{\frac{\begin{array}{c}\rho \vdash e_1 \Rightarrow \mathsf{Valfr}(f, x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow v_2 \\ \rho_0 \oplus [f \mapsto \mathsf{Valfr}(f, x, e_0, \rho_0)] \oplus [x \mapsto v_2] \vdash e_0 \Rightarrow r\end{array}}{\rho \vdash (e_1 e_2) \Rightarrow r}}\ (\textsc{AppFR})$$

$$\frac{\rho \vdash e_1 \Rightarrow r \neq \mathsf{Valf}(\_, \_, \_) \wedge r \neq \mathsf{Valfr}(\_, \_, \_, \_)}{\rho \vdash (e_1 e_2) \Rightarrow \mathsf{Erreur}}\ (\textsc{AppErrL})$$

$$\frac{\rho \vdash e_1 \Rightarrow \mathsf{Valf}(x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow \mathsf{Erreur}}{\rho \vdash (e_1 e_2) \Rightarrow \mathsf{Erreur}}\ (\textsc{AppFErrR})$$

$$\frac{\rho \vdash e_1 \Rightarrow \mathsf{Valfr}(f, x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow \mathsf{Erreur}}{\rho \vdash (e_1 e_2) \Rightarrow \mathsf{Erreur}}\ (\textsc{AppFRErrR})$$

Les règles d'erreurs (dont beaucoup se contentent de propager une erreur d'une sous-expression à l'expression englobante) « polluent » très nettement la présentation, c'est pourquoi on les omet généralement.

Ces axiomes (règles sans prémisses) et règles d'inférence permettent de fabriquer des *arbres d'évaluation* : les feuilles sont des instances d'axiomes, et les nœuds internes sont des instances de règles d'inférence.

Par exemple, en notant $\overline{f} = \mathsf{Valf}(x, x, \emptyset)$ et $\rho_f = [f \mapsto \overline{f}]$:

$$\dfrac{\dfrac{}{\emptyset \vdash (\mathsf{fun}\, x \to x) \Rightarrow \overline{f}}\,(\textsc{Fun}) \qquad \dfrac{\dfrac{f \in \mathsf{dom}(\rho_f)}{\rho_f \vdash f \Rightarrow \overline{f}} \qquad \rho_f \vdash 1 \Rightarrow 1 \qquad \dfrac{x \in \mathsf{dom}([x \mapsto 1])}{[x \mapsto 1] \vdash x \Rightarrow 1}}{\rho_f \vdash f(1) \Rightarrow 1}\,(\textsc{App})}{\emptyset \vdash (\mathsf{let}\, f = \mathsf{fun}\, x \to x\, \mathsf{in}\, f(1)) \Rightarrow 1}\,(\textsc{Let})$$

5.2 Ajout de structures de données

Enrichissons maintenant notre langage avec des couples (e_1, e_2) et deux primitives fst et snd qui vont agir comme la première et seconde projection, respectivement.

$$e \quad ::= \quad \ldots \mid (e_1, e_2) \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e$$

L'enrichissement correspondant de la sémantique opérationnelle consiste essentiellement en l'ajout d'une nouvelle forme de valeurs ainsi que l'écriture des règles d'évaluations qui auront à traiter les nouvelles constructions. Les couples de valeurs formant eux-mêmes des valeurs, l'ensemble des valeurs est étendu par :

$$v \quad ::= \quad \ldots \mid (v_1, v_2)$$

et les règles sont les suivantes (on omet les règles d'erreurs) :

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash (e_1, e_2) \Rightarrow (v_1, v_2)} \qquad \frac{\rho \vdash e \Rightarrow (v_1, v_2)}{\rho \vdash \mathsf{fst}\, e \Rightarrow v_1} \qquad \frac{\rho \vdash e \Rightarrow (v_1, v_2)}{\rho \vdash \mathsf{snd}\, e \Rightarrow v_2}$$

5.3 Évaluation non stricte du noyau fonctionnel de PCF

La sémantique stricte donnée dans la section précédente laisse entrevoir un choix très fort de comportement de programmes dont certaines expressions mènent à des erreurs. Considérons en particulier la sémantique des couples. Pour créer la valeur de couple correspondant à l'expression (e_1, e_2), nous évaluons e_1 puis e_2, ce qui nous donne les valeurs v_1 puis v_2 que nous assemblons pour former la valeur (v_1, v_2).

Que se passe-t-il si e_2 s'évalue correctement, disons en 42, et e_1 lève une erreur (ou boucle indéfiniment) ? Il sera impossible de créer la valeur de couple résultat. Pourtant, l'évaluation du programme $\mathsf{snd}\ (e_1, e_2)$ pourrait très bien retourner 42. La valeur de la première composante du couple n'étant pas utile, pourquoi l'évaluer ? Autant n'évaluer que « ce qui sert vraiment ».

Ce mode d'évaluation existe dans certains (peu de) langages, dont le plus célèbre est Haskell, et correspond à la mise en œuvre d'une sémantique dite *non stricte*. Nous verrons dans le chapitre 10 à quelle stratégie de réduction cela correspond exactement. Dans un le langage, une expression peut s'évaluer en une valeur alors que certaines de ses sous-expressions peuvent ne pas avoir de valeur.

Donnons maintenant, plus brièvement, la sémantique opérationnelle du noyau fonctionnel que nous avons traité ci-dessus mais avec un mode d'évaluation non strict. L'ensemble des valeurs du langage est modifié de sorte à contenir des données partiellement évaluées, selon les nécessités qui se sont faites sentir. Nous représentons ces données non évaluées par nouvelle catégorie d'éléments que nous nommerons *suspensions* :

$$s \quad ::= \quad \langle e, \rho \rangle$$

Dans l'état actuel de notre langage, les couples de suspensions feront des résultats de calculs tout à fait honorables :

$$v \quad ::= \quad c \mid \mathsf{Valf}(x, e, \rho) \mid \mathsf{Valfr}(f, x, e, \rho) \mid (\langle e_1, \rho_1 \rangle, \langle e_2, \rho_2 \rangle)$$

La règle d'évaluation des couples devient un axiome :

$$\rho \vdash (e_1, e_2) \Rightarrow (\langle e_1, \rho \rangle, \langle e_2, \rho \rangle)$$

et les règles concernant les projections se changent en :

$$\frac{\rho \vdash e \Rightarrow (\langle e_1, \rho_1 \rangle, \langle e_2, \rho_2 \rangle) \qquad \rho_1 \vdash e_1 \Rightarrow v_1}{\rho \vdash \mathsf{fst}\ e \Rightarrow v_1} \qquad \frac{\rho \vdash e \Rightarrow (\langle e_1, \rho_1 \rangle, \langle e_2, \rho_2 \rangle) \qquad \rho_2 \vdash e_2 \Rightarrow v_2}{\rho \vdash \mathsf{snd}\ e \Rightarrow v_2}$$

Notons aussi que les environnements ne contiennent plus que des suspensions, suite aux modifications des règles d'application et de déclarations locales données ci-dessous. La règle d'application s'écrit comme :

$$\frac{\rho \vdash e_1 \Rightarrow \mathsf{Valf}(x, e_0, \rho_0) \qquad \rho_0 \oplus [x \mapsto \langle e_2, \rho \rangle)] \vdash e_0 \Rightarrow r}{\rho \vdash (e_1 e_2) \Rightarrow r}$$

et celle concernant les déclarations locales devient :

$$\frac{\rho \oplus [x \mapsto \langle e_1, \rho \rangle] \vdash e_2 \Rightarrow r}{\rho \vdash (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \Rightarrow r}$$

Puisque les environnements ne contiennent plus que des suspensions, la règle d'accès aux valeurs des variables devient, quant à elle :

$$\frac{x \in \mathsf{dom}(\rho) \qquad \rho(x) = \langle e_0, \rho_0 \rangle \qquad \rho_0 \vdash e_0 \Rightarrow v}{\rho \vdash x \Rightarrow v}$$

Si nous avons omis – délibérément – les déclarations récursives, c'est que le mode d'évaluation que nous étudions permet d'élargir le champ des définitions récursives. En appel par valeur, nous avons bien pris garde à nous limiter aux déclarations récursives de fonctions, et nous avons garanti ce fait en limitant la construction syntaxique aux définitions de fonctions : $\mathsf{let\ rec}\ f(x) = e\ \mathsf{in}\ \ldots$. De la sorte, nous étions sûr de ne pas tenter d'évaluer un appel à f avant que la valeur fonctionnelle ne soit construite et stockée dans l'environnement.

Maintenant, l'évaluation se faisant à la demande, nous pouvons mettre directement la suspension récursive dans l'environnement, et son évaluation se fera en fonction du besoin. Nous étendons donc la syntaxe des déclarations récursives en $\mathsf{let\ rec}\ x = e_1\ \mathsf{in}\ e_2$, et les valeurs récursives ont elles aussi une forme plus générale : $\mathsf{Valr}(x, e, \rho)$. L'évaluation d'une définition récursive devient :

$$\frac{\rho \oplus [x \mapsto \mathsf{Valr}(x, e_1, \rho) \vdash e_2 \Rightarrow r}{\rho \vdash (\mathsf{let\ rec}\ x = e_1\ \mathsf{in}\ e_2) \Rightarrow r}$$

et l'activation des suspensions récursives se fait par l'intermédiaire de la règle d'accès aux environnements :

$$\frac{x \in \mathsf{dom}(\rho) \qquad \rho(x) = \mathsf{Valr}(x_0, e_0, \rho_0) \qquad \rho_0 \oplus [x_0 \mapsto \mathsf{Valr}(x_0, e_0, \rho_0)] \vdash e_0 \Rightarrow r}{\rho \vdash x \Rightarrow r}$$

Bien entendu, il est déraisonnable de réévaluer la même suspension autant de fois que sa valeur est nécessaire. Les langages correspondants – dits *paresseux* – mettent à jour les environnements et les structures de données comme les couples lorsqu'un de leurs éléments est évalué pour la première fois. Ainsi, tout accès subséquent trouvera à l'endroit correspondant une valeur prête à l'emploi. Il y a donc partage de l'évaluation pour gagner en performance d'exécution.

5.4 De l'évaluation à l'interprétation

Des règles données ci-dessus à l'écriture d'un interprète dans votre langage de programmation favori, il n'y a qu'un pas. En effet, si l'on prend une règle d'évaluation comme celle de l'application en appel par valeur :

$$\frac{\rho \vdash e_1 \Rightarrow \mathsf{Valf}(x, e_0, \rho_0) \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho_0 \oplus [x \mapsto v_2] \vdash e_0 \Rightarrow r}{\rho \vdash (e_1 e_2) \Rightarrow r}$$

on la traduit aisément dans un langage comme OCaml par :

```
let rec eval t env = match t with
  ...
| App (e1, e2) →
   (match (eval e1 env, eval e2 env) with
      (Funval (x, e0, env0), v2) → eval e0 (extend env0 (x, v2))
      (* On suppose que les erreurs sont traitées par la levée d'exceptions. *)
    | _ → erreur "application de non-fonction")
| ...
```

La traduction ne pose pas de difficulté majeure. Nous commençons par définir le type de l'arbre de syntaxe, donc des expressions.

```
type ident = string ;;

type expr =
  | Exp_bool of bool
  | Exp_int of int
  | Exp_var of string
  | Exp_fun of (string * expr)
  | Exp_app of (expr * expr)
  | Exp_let of (ident * expr * expr)
  | Exp_letr of (ident * ident * expr * expr)
  | Exp_if of (expr * expr * expr)
;;
```

Nous définissons ensuite un type somme qui encode les valeurs, avec un constructeur par cas de valeurs possibles. Ayant 4 cas pour notre langage (constantes entières et booléennes, fermetures récursives ou non), nous obtenons la définition suivante :

```
type pcf_val =
  | Val_int of int
  | Val_bool of bool
  | Val_fun of (ident * expr * env)
  | Val_fun of (ident * ident * expr * env)

and env = (ident * pcf_val) list ;;
```

dans laquelle on remarque une récursivité mutuelle avec le type des environnement puisque les fermetures, qui sont des valeurs, embarquent des environnements, qui contiennent des valeurs.

Vient ensuite la fonction de recherche dans un environnement. Il s'agit d'une simple recherche dans une liste d'association *clef-valeur* qui aurait très bien pu être remplacée par la fonction `List.assoc` de la bibliothèque standard d'OCaml. Le choix est fait ici de l'écrire nous-même afin de lever une exception dédiée plutôt que le `Not_found` levé par `List.assoc`. Une autre solution aurait été de rattraper `Not_found` et de re-lever notre propre exception.

```
let rec find_var id = function
  | [] → failwith ("unbound variable " ^ id)
  | h :: q → if fst h = id then snd h else find_var id q
;;
```

Il ne reste plus qu'à traduire les règles « normales » en cas de filtrage définissant une (ou plusieurs) fonction(s), en levant des exceptions dans les cas correspondant à des erreurs d'évaluation, et le tour est joué.

```
let rec eval env = function
  | Exp_bool b →Val_bool b
  | Exp_int n →Val_int n
  | Exp_var v →find_var v env
  | Exp_fun (p, c) → Val_fun (p, c, env)
  | Exp_app (e1, e2) → (
      let res_e1 = eval env e1 in
      match res_e1 with
      | Val_fun (p_name, e, f_env) →
          let res_e2 = eval env e2 in
          eval ((p_name, res_e2) :: f_env) e
      | Val_funr (f_name, p_name, e, f_env) →
          let res_e2 = eval env e2 in
          eval ((f_name, res_e1) :: (p_name, res_e2) :: f_env) e
      | _ → failwith "eval : not a function"
    )
  | Exp_if (e1, e2, e3) → (
      let resc = eval env e1 in
      match resc with
      | Val_bool true →eval env e2
      | Val_bool false →eval env e3
      | _ → failwith "eval : not a boolean"
    )
  | Exp_let (id, e1, e2) →
      let res_e1 = eval env e1 in
      eval ((id, res_e1) :: env) e2
  | Exp_letr (id, p_name, e1, e2) →
      let clos = Val_funr (id, p_name, e1, env) in
      eval ((id, clos) :: env) e2
;;
```

Bien sûr, cette implémentation très simple dans le cas d'un langage comme le noyau fonctionnel de

PCF. Lorsque des constructions plus complexes, comme des aspects impératifs (références, tableaux, entrées-sorties) ou des ruptures de contrôle (*goto*, continuations, exceptions) sont présentes, l'écrivain d'interprète a une tâche certes un peu plus délicate, mais qui ne pose pas de problèmes insurmontables. Pour des références, on ajoute une mémoire en argument à l'interprète. Pour les entrées-sorties, on met à contribution celles du langage hôte. Le mécanisme d'exceptions et la pleine fonctionnalité peuvent être utilisés pour encoder les ruptures de contrôle et les continuations.

5.5 De l'interprétation à la compilation

Passer de l'interprétation à la compilation est un peu plus délicat : nous tentons tout de même d'en donner l'intuition. Notons au préalable qu'il n'existe pas, à ma connaissance, de moyen efficace de passer d'un interprète à un compilateur [4]. En fait, on ne veut pas passer d'un interprète naïf comme celui qui est esquissé ci-dessus à un compilateur efficace comme celui d'OCaml ou du langage C, mais plutôt d'un *modèle* d'interprète, à un *modèle* de compilateur. L'interprète utilise déjà un environnement ρ et des valeurs, qu'un compilateur peut très bien adopter directement. Ce qui fait passer de l'interprétation à l'exécution d'un code, c'est la *gestion du contrôle* : là où notre morceau d'interprète ci-dessus calculait `eval` $\mathtt{e}_1$ `env` et `eval` $\mathtt{e}_2$ `env`, sans même préciser l'ordre relatif de ces calculs, l'exécution d'un code indiquera clairement cet ordre de calcul, et gèrera explicitement la sauvegarde de $\mathtt{e}_2$ et de `env` dans l'attente du retour de l'évaluation de $\mathtt{e}_1$. C'est donc le *contrôle* qu'un (modèle de) compilateur va expliciter.

Je donne ci-après un code pour le sous-ensemble fonctionnel de PCF : les motivations peuvent, je crois, en être améliorées. Ce le sera peut-être dans une version ultérieure de ces notes de cours.

Rappelons d'abord la syntaxe du langage que nous cherchons à compiler :

$$\begin{array}{rcl} e & ::= & c \mid x \mid e_1 e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ & \mid & e_1 + e_2 \mid e_1 = e_2 \\ & \mid & \mathsf{fun}\, x \ \rightarrow\ e \end{array}$$

Nous allons définir une *machine abstraite* dotée d'une pile, d'un registre et capable d'exécuter un code. Un environnement sera une simple liste de valeurs de la machine abstraite, et les accès aux valeurs des variables vont être réalisés au moyen d'entiers, dits *indices de de Bruijn*. Ce codage des variables par des entiers consiste à remplacer chaque occurrence de variable par un entier représentant le nombre de lieurs qu'il faut traverser pour passer, dans l'arbre de syntaxe abstraite, de l'occurrence considérée au lieur qui la lie. Considérons la définition suivante en OCaml :

```
let f x y =
  let z = x + y in
  z + x y
```

Les lieurs d'OCaml sont les fonctions (comme en λ-calcul où c'est justement le λ qui joue ce rôle) et les `let`. Effectivement, nous verrons plus tard (c.f 10.3.2) que cette construction peut être, en premier abord, encodée comme l'application d'une fonction. D'ailleurs, si nous réécrivons notre définition ci-dessus en λ-calcul nous obtenons le terme :

$$f = \lambda x.\lambda y.(\lambda z.\ +\ z\ (x\ y))\ (+\ x\ y)$$

Les nombres de lieurs à traverser sont :

- pour x, 3

4. Mentionnons tout de même les travaux de John Hannan, qui a tenté de systématiser le passage d'une sémantique opérationnelle à une machine abstraite, ainsi que la tradition de la technique appelée *évaluation partielle*, qui veut qu'un évaluateur appliqué à lui-même devienne un compilateur. . . ou quelque chose comme ça.

- pour y, 2
- et pour z, 1.

La représentation sous forme d'indices de de Bruijn de notre définition est ainsi :

$$f = \lambda.\lambda.(\lambda.\ +\ 1\ (3\ 2))\ (+\ 2\ 1)$$

L'algorithme de transformation d'un λ-terme en indices de de Bruijn est très simple et se réduit à un parcours du terme en profondeur au cours duquel on enregistre les variables au moment de leur liaison et où l'on remplace leurs occurrences dans le terme par la position à laquelle elles se trouvent dans cette liste. Le listing 5.2 illustre ce principe. Notons que dans cette version de l'algorithme, les identificateurs non trouvés dans l'environnement sont laissés inchangés. Les autres reçoivent comme nom la chaîne de caractères qui représente leur indice. Remarquons au passage que l'on fait disparaître le nom des variables dans les constructions λ qui les lient.

```
(** Trouve la position d'un élément dans une liste, avec la position 0 étant
    la tête de la liste. *)
let rec find_pos x = function
  | [] → raise Not_found
  | h :: q → if h = x then 0 else 1 + (find_pos x q) ;;

type lterm =
  | L_ident of string
  | L_app of (lterm * lterm)
  | L_lambda of (string * lterm) ;;

let to_de_bruijn term =
  let rec rec_convert bound t =
    match t with
    | L_ident x →
        (try L_ident (string_of_int (1 + find_pos x bound)) with
        | Not_found →t)
    | L_lambda (id, body) →
        L_lambda ("", (rec_convert (id :: bound) body))
    | L_app (t1, t2) →
        L_app ((rec_convert bound t1), (rec_convert bound t2)) in
  rec_convert [ ]term ;;
```

FIGURE 5.1 – Transformation de λ-terme en notation de de Bruijn

Les entiers de de Bruijn correspondent précisément à (1 + si l'on compte à partir de 0) la profondeur à laquelle il faut « descendre » dans un environnement pour y trouver la valeur associée à une occurrence libre de variable durant l'évaluation.

Nous allons donc considérer une machine comme un tuple (r, c, p) où r est le *registre* (parfois nommé *accumulateur*, c est le *code*, et p est la pile. Au départ de l'exécution, la machine est dans un *état* où le registre r contient l'environnement initial d'exécution, la pile est vide et le code est celui du programme à exécuter. Le code est une suite d'instructions, et l'exécution de chaque instruction fait passer la machine d'un état à un autre, mettant à jour une nouvelle instruction à exécuter, qui va amener la machine dans un autre état, et ainsi de suite jusqu'à ce que le code soit vide, de même que la pile (qui pourrait contenir des adresses de retour). À ce moment-là, le registre devrait contenir le résultat du calcul. Si la machine se

bloque, suite à l'impossibilité d'exécuter une instruction, on dira qu'elle est entrée dans un état erroné, et cet état correspondra à une erreur d'exécution.

Les instructions de la machine sont les suivantes :

$$
\begin{array}{rcl}
i & ::= & \texttt{Loadi(n)} \mid \texttt{Loadb(b)} \\
& \mid & \texttt{Plus} \mid \texttt{Sub} \mid \texttt{Equal} \\
& \mid & \texttt{Access(n)} \\
& \mid & \texttt{Branch}(\texttt{c}_1, \texttt{c}_2) \\
& \mid & \texttt{Push} \mid \texttt{Swap} \\
& \mid & \texttt{Mkclos(c)} \mid \texttt{Apply}
\end{array}
$$

et les objets manipulés par la machine sont des constantes entières (n), des constantes booléennes (b), des valeurs fonctionnelles ($\langle c, \rho \rangle$), des environnements (ρ), et des adresses de code (listes d'instructions, c). Nous pouvons simplement représenter ces objets par des définitions OCaml de la forme :

```
type vm_code = vm_instr list

and vm_instr =
  | VMI_Loadi of int
  | VMI_Loadb of bool
  | VMI_Plus | VMI_Sub | VMI_Equal
  | VMI_Access of int
  | VMI_Branch of (vm_code * vm_code)
  | VMI_Push
  | VMI_Swap
  | VMI_Mkclos of vm_code
  | VMI_Apply

type vm_val =
  | VMV_int of int
  | VMV_bool of bool
  | VMV_closure of (vm_code * vm_val)
  | VMV_env of vm_val list
  | VMV_code_addr of vm_code
```

FIGURE 5.2 – Définition des instructions et des valeurs de la machine abstraite

dans lesquelles nous voyons explicitement toutes les formes possibles (5) des valeurs que la machine abstraite peut traiter.

L'action des instructions est donnée par le tableau 5.3. Il est important de noter la subtile différence entre l'addition et la soustraction. En effet, la soustraction n'étant pas commutative, il conviendra que ses arguments soient opérés dans le bon ordre dans le code que génèrera le compilateur. On tentera bien sûr d'avoir une forme de code identique pour tous les opérateurs afin de ne pas rallonger notre compilateur.

En terme de notation, dans le tableau ci-dessus, nous effectuons des raccourcis de notation lorsque nous disons $c :: p$: ceci se lit "on rajoute sur la pile la valeur c qui est une adresse de code" (donc moralement, l'adresse d'une liste d'instruction, ce qui se représente dans notre implémentation de la figure 5.2 par une valeur `VMV_code_addr of vm_code`) ou se lit "la pile étant de la forme 1 élément c suivi d'autre chose", dans le cas où $c :: p$ apparaît dans la partie gauche du tableau (puisque la partie gauche représente les prémisses de chaque cas d'exécution).

état		état suivant
$(r, \texttt{Loadi}(n) :: c, p)$	$\Rightarrow$	(n, c, p)
$(r, \texttt{Loadb}(b) :: c, p)$	$\Rightarrow$	(b, c, p)
$(n, \texttt{Plus} :: c, m :: p)$	$\Rightarrow$	$(\overline{n + m}, c, p)$
$(n, \texttt{Sub} :: c, m :: p)$	$\Rightarrow$	$(\overline{n - m}, c, p)$
$(n, \texttt{Equal} :: c, m :: p)$	$\Rightarrow$	$(\overline{n = m}, c, p)$
$(\mathsf{true}, \texttt{Branch}(c_1, c_2) :: c, r :: p)$	$\Rightarrow$	$(r, c_1, c :: p)$
$(\mathsf{false}, \texttt{Branch}(c_1, c_2) :: c, r :: p)$	$\Rightarrow$	$(r, c_2, c :: p)$
$(r, \texttt{Push} :: c, p)$	$\Rightarrow$	$(r, c, r :: p)$
$(r_1, \texttt{Swap} :: c, r_2 :: p)$	$\Rightarrow$	$(r_2, c, r_1 :: p)$
$(r, \texttt{Mkclos}(c_1) :: c, p)$	$\Rightarrow$	$(\langle c_1, r \rangle, c, p)$
$(v, \texttt{Apply} :: c, \langle c_0, r_0 \rangle :: p)$	$\Rightarrow$	$((v \oplus r_0), c_0, c :: p)$
$(\rho, \texttt{Access}(n) :: c, p)$	$\Rightarrow$	$(\overline{\rho(n)}, c, p)$
$(r, [\,], c :: p)$	$\Rightarrow$	(r, c, p)

FIGURE 5.3 – Comportement des instructions de la machine virtuelle

En ce qui concerne le cas `Apply`, nous notons par $(v \oplus r_0)$ le fait d'étendre l'environnement (la valeur d'environnement) contenue dans r_0 (et donc on s'attend effectivement à ce que r_0 contienne une valeur d'environnement) en y rajoutant v. En d'autres termes, l'état résultat du cas `Apply` signifie que le registre r contient un environnement où v a été ajouté, que le prochain code à exécuter est celui du corps de la fonction (c_0) et la pile a été augmentée avec l'adresse du code à exécuter après l'appel de fonction.

Lorsque nous utilisons une barre horizontale au-dessus d'une expression, ceci signifie que l'opération invoquée est l'interprétation interne de celle-ci dans le code de l'interpréteur de la machine abstraite, donc l'opération correspondante dans le langage qui a servi à implémenter l'interpréteur. Autrement dit, $\overline{... + ...}$ dénote l'addition du langage d'implémentation, OCaml, C, ou autre. De la même manière, $\overline{\rho(n)}$ utilisé dans le cas `Access` représente l'accès effectué par le code de l'interpréteur dans les structures de celui-ci qui lui permettent de représenter les environnements.

Les trois points les plus importants qu'il faut avoir en tête quant à cette machine abstraite est que :

- **Lorsque l'instruction à exécuter est une application `Apply`, la fermeture à appliquer est attendue au sommet de la pile**.
- **Lorsque l'on entame le début du code d'une fonction (donc juste après avoir exécuté un `Apply`, cet environnement est attendu dans le registre r.**
- **Lorsque l'on doit faire un accès à l'environnement, celui-ci doit être accessible dans le registre r.**

Par la suite, on fera toujours en sorte d'être capable de récupérer cet environnement, donc le mémoriser dans la pile si on est à un point où l'on risque de le perdre et le récupérer lorsque l'on en a besoin. C'est là la plus grande gymnastique qu'il faut arriver à faire.

Il nous reste maintenant à donner la compilation du langage considéré : nous le faisons sous la forme d'une fonction notée $[\![e]\!]_\rho$ qui reçoit une expression e à compiler et un argument ρ qui est une liste de noms de variables et qui va être utilisée pour calculer l'indice de de Bruijn des noms de variables. Intuitivement, ρ est la liste des variables liées rencontrées et il nous rechercherons dedans chaque identificateur afin d'en déduire sa position dans cette liste. Cette position sera alors naturellement son indice de de Bruijn.

Il est important de noter que la fonction de compilation, donc le modèle de génération de code et la machine virtuelle sont intimement liés : le compilateur génère des suites d'instructions en accord avec la sémantique de celles-ci dans la machine abstraite afin de créer un programme en accord avec la sémantique du langage compilé. En d'autre termes, le jeu d'instruction de la machine abstraite a fixé des règles : c'est au compilateur de s'y tenir. Pour autant, il serait possible de choisir un modèle de machine

différent (plus de registres, des instructions différentes, un registre d'environnement, etc.), peut-être plus souple pour le compilateur, mais ce dernier devrait alors en retour être adapté à ces changements.

La fonction de compilation est donnée par :

$$
\begin{array}{lcl}
[\![n]\!]_\rho & = & \texttt{Loadi}(n) \\
[\![b]\!]_\rho & = & \texttt{Loadb}(b) \\
[\![x]\!]_\rho & = & \texttt{Access}(n) \quad \text{où } n \text{ est la profondeur de } x \text{ dans } \rho \\
[\![e_1 e_2]\!]_\rho & = & \texttt{Push}; [\![e_1]\!]_\rho; \texttt{Swap}; [\![e_2]\!]_\rho; \texttt{Apply} \\
[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]_\rho & = & \texttt{Push}; [\![e_1]\!]_\rho; \texttt{Branch}([\![e_2]\!]_\rho, [\![e_3]\!]_\rho) \\
[\![\mathsf{fun}\ x\ \rightarrow\ e]\!]_\rho & = & \texttt{Mkclos}([\![e]\!]_{x \oplus \rho}) \\
[\![e_1 + e_2]\!]_\rho & = & \texttt{Push}; [\![e_2]\!]_\rho; \texttt{Swap}; [\![e_1]\!]_\rho; \texttt{Plus} \\
[\![e_1 - e_2]\!]_\rho & = & \texttt{Push}; [\![e_2]\!]_\rho; \texttt{Swap}; [\![e_1]\!]_\rho; \texttt{Sub} \\
[\![e_1 = e_2]\!]_\rho & = & \texttt{Push}; [\![e_1]\!]_\rho; \texttt{Swap}; [\![e_2]\!]_\rho; \texttt{Equal} \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]_\rho & = & [\![(\mathsf{fun}\ x\ \rightarrow\ e_2) e_1]\!]_\rho
\end{array}
$$

Nous avons noté ici $x \oplus \rho$ l'extension de l'environnement ρ **de compilation** et non d'exécution, qui y rajoute **l'identificateur** x et non la valeur.

Nous avons choisi par simplicité de compiler les définitions locales $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ comme $(\mathsf{fun}\ x\ \rightarrow\ e_2) e_1$: au lieu d'écrire un cas particulier dans le compilateur pour cette construction, nous la transformons syntaxiquement puis nous rappelons la fonction de compilation sur cette nouvelle expression. Nous pourrions éventuellement optimiser un peu, notamment éviter la création de la fermeture, qui est ici inutile. Les définitions récursives sont, quant à elles, plus complexes à traiter : elles nécessitent la création d'une valeur factice spéciale, qui sera stockée dans l'environnement en lieu et place de la fonction récursive, en tant que valeur initiale. Lorsque la valeur de cette fonction récursive sera créée, on met à jour – physiquement – l'environnement en remplaçant la valeur factice par la valeur calculée.

5.5.1 Explications pas-à-pas

Il est maintenant utile de comprendre le fonctionnement pas à pas de la machine abstraite sur le code généré en fonction du programme initial. Nous allons commencer par "tracer" un exemple puis nous verrons comment l'on peut "deviner" le code à générer pour le cas des opérateurs binaires (+, -, = que nous avons d'ailleurs traités).

Trace d'exécution

Commençons par une expression simple : (`fun x` $\rightarrow$ `fun y` $\rightarrow$ `x`) 14 42. Le code généré par la fonction de compilation est le suivant :

```
1  Push
2  Push
3  Mkclos
4    Mkclos
5      Access 1
6  Swap
7  Loadi 14
8  Apply
9  Swap
10 Loadi 42
11 Apply
```

Rappelons-nous que l'exécution débute avec le registre r contenant l'environnement. Nous allons suivre pas à pas la forme, le contenu du registre r, de la pile et de l'environnement.

État initial de la machine : Le registre *r* contient donc l'environnement, le pointeur de code pointe sur le début du programme et la pile est initialement vide.

Registrer :< *env* >= []
Code : [`Push`; `Push`; `Mkclos`[`Mkclos`[`Access1`]]; `Swap`; `Loadi14`; `Apply`; `Swap`; `Loadi42`; `Apply`]
Stack : []

État après étape 1 : Nous avons exécuté l'instruction (1), `Push` ce qui nous permet de sauvegarder l'environnement sur la pile. On remarquera au passage que fort logiquement, l'environnement est vide, puisqu'il n'y a eu jusqu'alors aucune définition `let` et que nous ne sommes pas à l'intérieur d'une fonction (auquel cas, nous aurions vu l'argument de cette fonction dans notre environnement). Notons également que l'environnement reste présent dans *r*. Ce `Push` est initié par la présence d'une application qui "se sauvegarde" l'environnement actuel au cas où elle en aurait besoin pour calculer sa valeur fonctionnelle.

Registrer :< *env* >= []
Code : [`Push`; `Mkclos`[`Mkclos`[`Access1`]]; `Swap`; `Loadi14`; `Apply`; `Swap`; `Loadi42`; `Apply`]
Stack : [< *env* >= []]

État après étape 2 : Nous avons exécuté l'instruction (2), `Push` ce qui nous permet de sauvegarder l'environnement sur la pile une fois de plus. La pile contient donc deux éléments représentant cet environnement.

Registrer :< *env* >= []
Code : [`Mkclos`[`Mkclos`[`Access1`]]; `Swap`; `Loadi14`; `Apply`; `Swap`; `Loadi42`; `Apply`]
Stack : [< *env* >= []; < *env* >= []]

État après étape 3 : Nous avons exécuté l'instruction (3) qui crée une fermeture dans le registre *r*. On voit que le corps de cette fermeture (`Mlclos ; Mkclos`) est le code permettant de créer la seconde fermeture à venir. Ce code ne va donc pas être exécuté à la prochaine étape mais seulement lors de l'application de la fermeture fraîchement créée, donc aux étapes 6-7.

Registrer :< fun([`Mkclos`; `Access1`], < *env* >= []) >
Code : [`Swap`; `Loadi14`; `Apply`; `Swap`; `Loadi42`; `Apply`]
Stack : [< *env* >= []; < *env* >= []]

État après étape 4 : L'instruction exécutée est le `Swap` (6) qui intervertit les contenus de la pile et du registre *r*. Nous avons en effet besoin de conserver quelque part la fermeture pour une éventuelle application ultérieure qui va d'ailleurs se produire en étapes 5-6). Et ce "quelque part", il n'y a que la pile qui nous le propose de manière durable. Donc on met le contenu de *r* qui contenait la fermeture sur la pile. Pour autant, nous aurions pu simplement le `Push`-er, le rajoutant simplement sur la pile par-dessus ce qui existait déjà. Nous choisissons de ne pas augmenter la taille de la pile en faisant un échange avec ce qu'il y avait déjà sur la pile. Pourquoi peut-on faire ceci, en quoi la valeur qui était avant au sommet de la pile peut être remplacée et donc perdue (oui, l'instruction suivante va écraser la valeur du registre *r*)? La réponse à cette question est que c'est l'application la plus interne qui avait provoqué la sauvegarde de l'environnement pour ses éventuels besoin intermédiaires pendant le calcul de la valeur fonctionnelle à appliquer. Or, maintenant cette application est sur le point d'être effectuée, sa valeur fonctionnelle a été calculée et donc l'application n'a plus besoin de cet environment. Il "lui" restera seulement à calculer la valeur de son argument et à β-réduire.

Registrer :< *env* >= []
Code : [`Loadi14`; `Apply`; `Swap`; `Loadi42`; `Apply`]
Stack : [< fun([`Mkclos`[`Access1`]], < *env* >= []) >; < *env* >= []]

État après étape 5 : À la fin de cette étape, nous avons exécuté l'instruction (7) qui charge la valeur immédiate 14 dans le registre *r*, le préparant donc à contenir la valeur de l'argument effectif à passer à la fonction lors de la prochaine étape : l'application.

Registrer : 14
Code : [`Apply`; `Swap`; `Loadi42`; `Apply`]
Stack : [< fun([`Mkclos`[`Access1`]], < *env* >= []) >; < *env* >= []]

État après étape 6 : L'instruction d'application (8) a été exécutée et a récupéré la valeur de l'argument effectif à passer à la fonction dans *r*, a récupéré la valeur fonctionnelle comme attendue sur la pile et à mis dans *r* l'environnement de la fermeture, étendu avec l'argument effectif passé dans le registre *r*. Effectivement, rappelons-nous que nous avions insisté sur le fait qu'au début de l'exécution d'une fonction, on s'attendait à ce que l'environnement se trouve dans le registre *r*. Et là, il y est bien, et comme attendu, étendu par la liaison entre le paramètre formel de la fonction et sa valeur effective. Pour terminer, on remarquera que l'instruction `Apply` a eu effet de "dérouter" le pointeur de code vers la suite des instructions que représente le corps de la fonction appliquée et que "l'adresse" du code à exécuter au retour de l'application a été sauvegardé dans la pile (Stack contient à son sommet < *code* >).

Registrer :< *env* >= [14]
Code : [`Mkclos`[`Access1`]]
Stack : [< *code* >; < *env* >= []]

État après étape 7 : Nous avons ici exécuté la première (et seule) instruction (4) de la fermeture, c'est-à-dire ... la création de la seconde fermeture. Ceci se déroule comme vu précédemment. On note que la liste d'instruction présente dans `Code` est désormais vide, c'est normal puisque nous avosn exécuté toutes les instruction de la fonction appelée (qui n'en comportait qu'une : `Mkclos [ Access 1 ]`.

Registrer :< fun([`Access1`], < *env* >= [14]) >
Code : []
Stack : [< *code* >; < *env* >= []]

État après étape 8 : Puisque `Code` était la liste vide, c'est que nous avions atteint la fin d'une fonction. Il faut donc retourner au calcul laissé en suspens dont l'adresse a été sauvegardé sur la pile (dernière ligne du tableau 5.3). Ainsi, à la fin de cette étape, nous retrouvons dans le code à (continuer à) exécuter celui laissé en suspens : `Swap ; Loadi 42 ; Apply`. Celui-ci a bien été extrait de la pile qui ne contient plus que l'environnement `Push`-é après l'étape 1. Nous nous retrouvons désormais dans une situation similaire à l'étape 3.

Registrer :< fun([`Access1`], < *env* >= [14]) >
Code : [`Swap`; `Loadi42`; `Apply`]
Stack : [< *env* >= []]

État après étape 9 : Le même processus qu'à l'étape 4 se reproduit pour intervertir la fermeture et l'environnement entre la pile et le registre. Ainsi, la fermeture qu'il faudra appliquer "un jour" est bien conservée. Quant à l'environnement qui se retrouve désormais dans le registre, comme nous l'avons dit à l'étape 4, pas grave s'il se retrouve écrasé, on n'en a plus besoin.

Registrer :< *env* >= []
Code : [`Loadi42`; `Apply`]
Stack : [< fun([`Access1`], < *env* >= [14]) >]

État après étape 10 : On met la valeur immédiate 42 dans le registre. Cette valeur est l'argument effectif de l'appel de fonction à venir. Il se retrouve donc comme attendu dans *r* et nous sommes prêts pour effectuer une application.

Registrer : 42
Code : [`Apply`]
Stack : [< fun([`Access1`], < *env* >= [14]) >]

État après étape 11 : Nous nous retrouvons dans une situation identique à l'étape 6, avec le pointeur de code dérouté vers le corps de la fonction appelée et le code mis en suspens ("adresse de retour") empilé. Le code à exécuter est la seule instruction de la fermeture : `Access 1`.

Registrer :< *env* >= [42, 14]
Code : [`Access1`]
Stack : [< *code* >]

État après étape 12 : Comme attendu en début de fonction, l'environnement est dans r. Nous accédons à la cellule 1 (seconde cellule) de l'environnement. C'est la seule instruction de la fonction, donc nous sommes désormais prêt à retourner au calcul en suspens qui est sur la pile.

Registrer : 14
Code : []
Stack : [< *code* >]

État (final) après étape 13 : Puisque c'était la fin du programme, nous sommes retournés à une suite d'instructions vide. L'exécution s'arrête donc.

Registrer : 14
Code : []
Stack : []

Deviner le code à générer pour la soustraction

Après cet échauffement, venons-en à comprendre le code à générer pour les opérateurs binaires. Considérons le cas de la soustraction, les autres opérateurs étant similaires.

Le comportent de l'instruction de soustraction est : $(n, \texttt{Sub} :: c, m :: p) \rightarrow (\overline{n-m}, c, p)$. Cela nous indique que le résultat du calcul de la première opérande doit être dans le registre r et celui de la seconde au sommet de la pile. Il faut donc que la compilation d'une expression $e_1 - e_2$ satisfasse ces deux critères. À un point quelconque d'exécution, r peut contenir quelque chose d'important à conserver. Il faut donc le sauvegarder avant d'écraser sa valeur par le résultat de l'une des deux expressions e_1 ou e_2. Donc il faut générer un `Push`.

Ensuite, se pose la question de générer le code de e_1 ou de e_2 : qu'est-ce qui est le plus pertinent ? Il va falloir que le résultat de e_1 se trouve *in fine* dans r et celui de e_2 dans la pile. Lorsqu'à l'exécution un calcul est effectué, où se trouve le résultat ? En regardant le comportement des instructions, on voit que dans les cas de base (constantes) c'est dans r. Donc nous allons maintenir cet invariant inductivement lors de l'évaluation des expressions arithmétiques. Donc le résultat de e_1 ou e_2 sera dans r. Il apparaît donc qu'il vaut mieux exécuter le code de e_2 en premier, sauvegarder la valeur obtenue (v_2) sur la pile puis exécuter le code de e_1 qui arrivera naturellement en dernier dans r. Faire dans le sens inverse nous aurait obligé à aller mettre v_2 sur la pile et à récupérer v_1 de la pile pour le remettre explicitement dans r. Donc il faut générer le code de e_2.

Il va nous falloir maintenant compiler e_1. La situation est la même que pour e_2 : il faut sauvegarder r pour ne pas perdre la valeur de e_2. De plus, lorsque l'on avait calculé e_2, on avait pris soin de sauvegarder r tout en lui laissant sa valeur initiale qui pouvait être importante pour calculer e_2. Cette valeur est peut-être tout aussi importante pour calculer e_1 donc il faut la remettre dans r. Ça tombe bien, elle est sur la pile et en plus on doit sauvegarder r. On n'a qu'à échanger les deux : on génère un `Swap`.

On peut enfin vraiment générer le code de e_1 et on s'attend à ce que son résultat (v_1) soit dans r. Il nous reste à réellement faire la soustraction. C'est parfait, nous avons v_1 dans r et v_2 sur la pile : c'est exactement ce qu'il nous faut pour l'instruction `Sub`. Nous pouvons émettre un `Sub`. On remarque que l'on a bien pris soin de s'assurer que le code produit faisait la soustraction dans le bon sens : $e_1 - e_2$. En effet, pour l'addition qui est commutative, inverser les opérandes n'aurait pas été faux, mais ce n'est clairement pas le cas pour la soustraction.

Remarquons que la valeur de r au tout début du calcul de l'expression, valeur que nous avions sauvegardée par le premier `Push` est désormais perdue. Ce n'est pas grave puisque le calcul de notre expression est terminé, nous n'en aurons plus besoin. En résumé, nous avons généré la suite d'instructions `Push`; $[\![e_2]\!]_\rho$; `Swap`; $[\![e_1]\!]_\rho$; `Sub`.

Conclusion

Nous voyons donc que la manière de compiler dépend du comportement des instructions et des choix que l'on fait ... quand on compile (dépendance circulaire). En effet, le choix que r contienne le résultat d'une expression arithmétique est propre au schéma de compilation. Mais une fois ce choix arrêté, il faut s'y tenir. Ce choix n'est pas totalement arbitraire puisque les instructions arithmétiques de la machine attendent l'une des valeurs d'opérandes dans r. Il y a donc de subtiles dépendances entre instructions et compilation. Lorsque l'on conçoit une machine virtuelle, on a en tête le processus de compilation qui va y être associé : les deux doivent se simplifier la vie mutuellement autant que possible.

Chapitre 6

Termes du premier ordre

Les termes du premier ordre forment une façon générale de représenter de nombreuses données manipulées en logique ou en programmation. On s'en sert pour représenter des expressions logiques, des expressions de type, ou des données structurées manipulées par des programmes. On y pense en permanence lorsqu'on écrit des expressions ou des programmes de façon linéaire en mathématiques ou dans des programmes informatiques. On peut les voir et les dessiner comme des arbres afin de mieux visualiser leur structure. Ces termes sont construits à partir de symboles et de noms de variables. Un terme avec variables peut être vu comme représentant une infinité de termes, et c'est sur ces termes avec variables que prennent tout leur sens les notions de filtrage et d'unification.

Ces termes sont dits *du premier ordre* car ils sont dénués de *lieur*[1], c'est-à-dire de constructions introduisant des noms de variables dans une certaine portée : ce qu'on appelle des variables muettes en mathématiques, ou des variables liées en informatique.

6.1 Termes

Soit un ensemble Σ de symboles. On suppose qu'à chaque symbole est associé un entier appelé *arité*. On appelle généralement *signature* un tel ensemble et *symboles de fonction* les symboles de Σ. Si on connaît les langages fonctionnels de la famille ML ou Haskell, les symboles de fonctions peuvent être vus comme des constructeurs de données. Pour cette raison, nous les noterons ici par des noms qui commencent par une lettre majuscule, comme `F` ou `Zero`.

Exemple

- $\Sigma_1 = \{(\texttt{Zero}, 0), (\texttt{Succ}, 1), (\texttt{Plus}, 2)\}$
- $\Sigma_2 = \{(\texttt{A}, 0), (\texttt{F}, 1), (\texttt{G}, 3)\}$

Définition (Termes) Étant donné un tel ensemble de symboles, on peut définir l'ensemble $\mathcal{T}_\Sigma$ des termes sur Σ, que l'on appelle Σ-termes, ou simplement *termes* lorsque Σ est implicite, par :

- tout symbole de Σ d'arité 0 est un Σ-terme
- si $t_1, \ldots, t_n$ sont des Σ-termes et si $F \in \Sigma$ est un symbole d'arité n, alors $F(t_1, \ldots, t_n)$ est un Σ-terme.

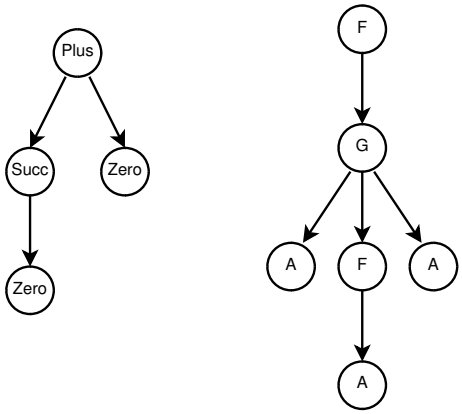
Exemples

- $\texttt{Plus(Succ(Zero), Zero)} \in \mathcal{T}_{\Sigma_1}$
- $\texttt{F(G(A, F(A), A))} \in \mathcal{T}_{\Sigma_2}$
- $\texttt{Succ(F(A))} \in \mathcal{T}_{\Sigma_1 \cup \Sigma_2}$

1. Par exemple, le symbole de quantification universelle $\forall$ introduit généralement une ou plusieurs variables dont la *portée* est son champ de validité dans l'expression. Hors de ce champ, c'est-à-dire hors de la portée du quantificateur, cette variable n'est plus valide.

- `Zero(Succ)` n'est pas un terme valide

On représente souvent graphiquement de tels termes :



Les termes de $\mathcal{T}_\Sigma$ ne contiennent que des symboles, mais pas de variable. On dit qu'ils sont *clos*. Si on dispose, en plus de Σ, d'un ensemble $\mathcal{V}$ de noms de variables (qu'on choisit généralement infini dénombrable), on peut construire des termes sur $\Sigma \cup \mathcal{V}$ de la même manière que précédemment :

Définition (Termes avec variables)
- toute variable de $\mathcal{V}$ est un $\Sigma \cup \mathcal{V}$-terme
- tout symbole de Σ d'arité 0 est un $\Sigma \cup \mathcal{V}$-terme
- si $t_1, \ldots, t_n$ sont des $\Sigma \cup \mathcal{V}$-termes et si $F \in \Sigma$ est un symbole d'arité n, alors $F(t_1, \ldots, t_n)$ est un $\Sigma \cup \mathcal{V}$-terme.

On note $\mathsf{var}(t)$ l'ensemble des variables apparaissant dans le terme t. Dans la suite, on laissera généralement implicite l'ensemble de variables $\mathcal{V}$.

6.2 Substitutions

Une substitution est une fonction totale de l'ensemble des variables $\mathcal{V}$ vers un ensemble de termes $\mathcal{T}$. On appelle *domaine* d'une substitution θ, et on le note $\mathsf{dom}(\theta)$, l'ensemble des variables x telles que $\theta(x) \neq x$. On ne considèrera dans la suite que des substitutions dont le domaine est fini.

On notera quelquefois $[x_1 \mapsto t_1; \ldots; x_n \mapsto t_n]$ la substitution de domaine $\{x_1, \ldots, x_n\}$ qui associe le terme t_i à la variable x_i, pour $i = 1, \ldots n$. On note $[\,]$ ou $\emptyset$ la substitution dont le domaine est vide. On appelle *renommage* une substitution bijective θ dont le codomaine (c'est-à-dire l'ensemble $\{\theta(x)$ pour $x \in \mathsf{dom}(\theta)\}$) ne contient que des variables.

On étend naturellement une substitution θ en un *morphisme* de termes $\overline{\theta}$:
- $\overline{\theta}(\mathtt{F}(t_1, \ldots, t_n)) = \mathtt{F}(\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n))$
- $\overline{\theta}(x) = \theta(x)$

Par abus de langage, on confondra dans la suite une substitution θ et le morphisme qui lui est associé $\overline{\theta}$.

On définit la composition $\theta_1 \circ \theta_2$ de deux substitutions θ_1 et θ_2 comme la substitution qui associe à toute variable x le terme $\overline{\theta_1}(\theta_2(x))$.

On définit l'extension d'une substitution θ par une autre substitution dont le domaine est un singleton de la façon suivante :

Définition (Extension) L'extension de la substitution θ par $[x \mapsto t]$ est la substitution θ' notée $\theta \oplus [x \mapsto t]$ définie par :
- $\mathsf{dom}(\theta') = \mathsf{dom}(\theta) \cup \{x\}$
- $\theta'(x) = t$
- $\theta'(y) = \theta(y)$ pour $y \neq x$

On remarquera que si $\theta(x)$ est définie (et $\neq x$), alors l'extension de θ par $[x \mapsto t]$ masquera l'action de θ sur x.

Les substitutions induisent un pré-ordre sur les termes (c'est-à-dire une relation binaire réflexive et transitive).

Définition (Plus ou moins général) On dira que t_1 est *moins général* que t_2, et on notera $t_1 \leq t_2$ si il existe une substitution θ telle que $t_1 = \theta(t_2)$.

On dira quelquefois dans ce cas que t_1 est une *instance* de t_2.

6.3 Le filtrage

Le problème dit du filtrage, c'est de trouver une réponse à la question suivante :

Étant donnés deux termes t et m, a-t-on $t \leq m$?

que l'on peut aussi formuler comme :

Le terme t est-il une instance du terme m ?

Ce problème apparaît bien plus souvent qu'on ne le croit. Bien sûr, on le retrouve presque directement sous cette forme dans l'opération de filtrage des langages comme OCaml ou Haskell, où on teste si le résultat d'une évaluation est une instance d'un motif. On l'utilise presque tout aussi directement lorsqu'on résout des problèmes mathématiques et qu'on cherche à transposer un théorème général dans un contexte particulier : on se demande alors « ce contexte est-il une instance des hypothèses du théorème ? ». En fait, ce problème du filtrage est à la base même de bon nombre de raisonnements que nous effectuons quotidiennement, dans lesquels on cherche à construire un exemple particulier de règle générale.

Pour revenir à nos termes formels, résoudre un problème de filtrage, c'est calculer une substitution qui permet de passer d'un terme ou *motif* possédant généralement – mais pas nécessairement – des variables, à une instance de ce terme. En voici quelques exemples :

terme		motif	substitution
$\texttt{Succ(Succ(Zero))}$	$\leq$	$\texttt{Succ}(x)$	$\theta = [x \mapsto \texttt{Succ(Zero)}]$
$\texttt{Succ(Succ}(y))$	$\leq$	$\texttt{Succ}(x)$	$\theta = [x \mapsto \texttt{Succ}(y)]$
$\texttt{Succ(Succ}(x))$	$\leq$	$\texttt{Succ}(x)$	$\theta = [x \mapsto \texttt{Succ}(x)]$
$\texttt{G(F(A), F(A), F(A))}$	$\leq$	$\texttt{G}(x, x, x)$	$\theta = [x \mapsto \texttt{F(A)}]$
$\texttt{Succ}(x)$	$\not\leq$	$\texttt{Succ(Succ}(y))$	—
$\texttt{G(F(A), A, F(A))}$	$\not\leq$	$\texttt{G}(x, x, x)$	—

Algorithme de filtrage Nous appelons filtre l'algorithme de filtrage. Au lieu de le définir sur une paire de termes, nous le définissons sur une liste de paires de termes. En effet, lorsqu'on cherche à filtrer un terme t par un autre terme m (le *motif*), on est amené à décomposer en parallèle ces deux termes, pour se ramener à deux sous-problèmes de taille inférieure. On traitera donc une liste de problèmes de filtrage : le premier élément de cette liste sera le problème courant, les autres sont les problèmes en attente. Nous utilisons une syntaxe OCaml pour de telles listes. Le problème initial s'écrit filtre $[(m, t)]$ et se lit « le motif m filtre-t-il le terme t », ou bien « le terme t est-il une instance de m ? ».

L'algorithme filtre, donné à la figure 6.1, reçoit un argument supplémentaire, une substitution, qui vaut initialement $\emptyset$, et qui sert d'accumulateur. Cet argument est produit en résultat de l'algorithme en cas de réussite du filtrage.

Notons que dans le cas où les deux symboles S et T sont égaux (ligne 10), on sait qu'ils ont même arité et donc que $k = l$, car on ne considère que des termes bien formés. Remarquons aussi que la restriction de la solution du problème de filtrage de deux termes t et m à l'ensemble des variables apparaissant dans t est unique. L'algorithme ci-dessus calcule précisément cette restriction, commune à toutes les

```
1   filtre [(m, t)] ∅ est défini par
2     filtre : (terme × terme) list → substitution → substitution (ou échec)
3     filtre [ ] θ = θ
4     filtre [(x, t1); (m2, t2); ...; (mn, tn)] θ =
5       si x ∉ dom(θ) alors
6         filtre [(m2, t2); ...; (mn, tn)] (θ ⊕ [x ↦ t1]) sinon
7       si θ(x) = t1 alors filtre [(m2, t2); ...; (mn, tn)] θ
8       sinon échec
9     filtre [(S(q1, ..., qk), T(u1, ..., ul)); (m2, t2); ...; (mn, tn)] θ =
10      si S = T alors filtre [(q1, u1); ...; (qk, uk); (m2, t2); ...; (mn, tn)] θ
11      sinon échec
12    filtre n'importe quoi d'autre = échec
```

FIGURE 6.1 – L'algorithme de filtrage de t par m

solutions. Enfin, on peut remarquer aussi lorsque le motif ne contient aucune variable, l'algorithme de filtrage devient un test d'égalité entre deux termes.

On peut, à juste titre, se poser la question de savoir ce qui se passe lorsque les termes m et t partagent des variables. Dans ce cas, on peut être amené à utiliser une substitution θ qui à une telle variable partagée x associe x elle-même. La définition du domaine d'une substitution nous indique que $x \notin \mathsf{dom}(\theta)$, puisque $\theta(x) = x$, et notre algorithme est incorrect dans ce cas. Pour s'on convaincre, il suffit de calculer :

$$\mathsf{filtre}\ \emptyset\ [(\mathtt{Plus}(x, x), \mathtt{Plus}(x, 0))]$$

ce, en toute rigueur, nous dit que la substitution $[x \mapsto 0]$ change $\mathtt{Plus}(x, x)$ en $\mathtt{Plus}(x, 0)$, ce qui est clairement faux.

Il y a deux solutions possibles à ce problème :

- reformuler le problème de filtrage en disant qu'il n'est bien posé que s'il concerne deux termes n'ayant aucune variable en commun ;
- ou alors reformuler l'algorithme en définissant le paramètre auxiliaire θ non pas comme une substitution, mais comme une liste de couples (variable, terme), et remplacer le test de la ligne 5 par un test qui calcule s'il existe ou non un couple dans cette liste dont la première composante est la variable x.

6.4 Unification

Le problème de l'unification consiste, quant à lui, à comprendre si deux termes ont des instances communes. On voit aisément que cela peut être :

- quelquefois impossible (prendre par exemple deux constantes[2] `A` et `B`),
- d'autres fois aussi facile que le filtrage (prendre par exemple $\mathtt{Succ}(x)$ et $\mathtt{Succ}(\mathtt{Zero})$),
- et d'autres fois encore un peu plus difficile (prendre par exemple $\mathtt{Plus}(x, \mathtt{Succ}(\mathtt{Zero}))$ et $\mathtt{Plus}(\mathtt{Succ}(y), z)$).

On peut aussi voir ce problème d'unification comme la résolution d'une équation $t = u$ entre termes.

Le problème de l'unification se formule donc ainsi, étant donnés deux termes t et u : existe-t-il une substitution μ telle que $\mu(t) = \mu(u)$? On appelle une telle substitution μ un *unificateur* de t et u. Si deux termes admettent un unificateur, on dit qu'ils sont *unifiables*.

2. C'est-à-dire deux symboles d'arité 0.

Notons que si μ est un unificateur de t et u, alors pour toute substitution θ, $\theta \circ \mu$ en est aussi un.

Termes à unifier	résultat
$\texttt{Succ}(\texttt{Succ}(x))$ et $\texttt{Succ}(\texttt{Succ}(x))$	$\emptyset$
$\texttt{Plus}(\texttt{Zero}, \texttt{Zero})$ et $\texttt{Plus}(x, \texttt{Zero})$	$[x \mapsto \texttt{Zero}]$
$\texttt{Plus}(\texttt{Zero}, x)$ et $\texttt{Plus}(x, \texttt{Zero})$	$[x \mapsto \texttt{Zero}]$
$\texttt{Plus}(\texttt{Succ}(y), x)$ et $\texttt{Plus}(\texttt{Succ}(\texttt{Succ}(x)), \texttt{Succ}(\texttt{Zero}))$	$[y \mapsto \texttt{Succ}(\texttt{Succ}(\texttt{Zero})); x \mapsto \texttt{Succ}(\texttt{Zero})]$
$\texttt{Succ}(\texttt{Zero})$ et $\texttt{Succ}(\texttt{Succ}(\texttt{Zero}))$	—
$\texttt{Succ}(x)$ et $\texttt{Succ}(\texttt{Succ}(x))$	—

Le dernier exemple représente en fait une tentative de résolution d'équation récursive. On pourrait admettre qu'une solution possible est la substitution qui à x associe le terme infini solution de l'équation $y = \texttt{Succ}(y)$. En ce qui nous concerne, ce dernier terme n'entre pas dans la définition que nous avons donnée des termes, qui ne contenait que des termes finis.

L'algorithme qui va calculer un unificateur de deux termes t et u devra lui aussi comparer les symboles de tête des deux termes, et, s'il s'agit du même symbole, traiter récursivement les unifications des sous-termes respectifs de t et de u. L'algorithme de vérification recevra donc, tout comme celui du filtrage, une liste de paires de termes à unifier, ainsi qu'une substitution utilisée comme accumulateur, puis produite en résultat si l'unification réussit.

Avant de donner l'algorithme complet, examinons-en ses grandes étapes :

- soit à unifier $[(t_1, u_1), \ldots, (t_n, u_n)]$: on cherche μ telle que $\forall i = 1..n, \mu(t_i) = \mu(u_i)$
- si t_1 est unifiable avec u_1, produisant μ_1, on continue l'unification de

$$[(\mu_1(t_2), \mu_1(u_2)), \ldots, (\mu_1(t_n), \mu_1(u_n))]$$

- si cela réussit et produit μ', alors $\mu' \circ \mu_1$ unifie $[(t_1, u_1), \ldots, (t_n, u_n)]$.

```
1   unifier [(t, u)] ∅ est défini par
2    unifier : (term × term) list → substitution → substitution (ou échec)
3    unifier [ ] μ = μ
4    unifier [(x, u1); (t2, u2); ...; (tn, un)] μ =
5       si x = u1 alors unifier [(t2, u2); ...; (tn, un)] μ sinon
6       si x apparaît dans u1 alors échec sinon
7       soit μ1 = [x ↦ u1]
8       unifier (μ1(t2), μ1(u2)); ...; (μ1(tn), μ1(un))] (μ1 ∘ μ)
9    unifier [(t1, x); ...; (tn, un)] μ = unifier [(x, t1); ...; (tn, un)] μ
10   unifier [(S(a1, ..., ak), T(b1, ..., bl)); (t2, u2); ...; (tn, un)] μ =
11      si S = T alors unifier [(a1, b1); ...; (ak, bk); (t2, u2); ...; (tn, un)] μ
12      sinon échec
```

FIGURE 6.2 – L'algorithme d'unification de t et u

L'algorithme d'unification de t et de u est donné à la figure 6.2. Il utilise lui aussi une substitution auxiliaire θ qui sera produite en résultat si l'unification réussit. Comme dans le cas du filtrage, on a $k = l$ quand $S = T$ dans le dernier cas de la fonction d'unification.

Le résultat de l'algorithme d'unification est un unificateur de t et de u. Comme nous l'avons déjà remarqué, l'unificateur μ de deux termes donnés n'est pas unique, puisque pour toute substitution θ, $\theta \circ \mu$ est aussi un unificateur de ces deux termes. Il se trouve qu'il existe un unificateur que nous noterons μ qui est « meilleur » que tous les autres, au sens où tous les autres se déduisent de μ par compositions : si θ unifie t et u, alors il existe σ telle que $\theta = \sigma \circ \mu$. La substitution μ est appelé l'unificateur le plus général de t et u.

Cet unificateur est appelé l'unificateur *le plus général* (*most general unifier*, en anglais), et il est unique à un renommage près. C'est cet unificateur le plus général que calcule l'algorithme ci-dessus. Pour deux termes donnés t et u, on le note quelquefois $\mathsf{mgu}(t, u)$. L'algorithme ci-dessus a été inventé par Alan Robinson en 1965.

Chapitre 7

Vérification et inférence de types

La programmation non typée est d'une grande souplesse : elle permet le meilleur . . . comme le pire. C'est elle qui permet par exemple de coder dans le λ-calcul (et dans tout langage non typé, ou typé dynamiquement) des choses aussi puissantes que les combinateurs de point fixe, les entiers de Church, l'interprétation d'entiers machine comme des pointeurs, les écrans bleus de Windows, les erreurs de segmentation et les *kernel panic* d'Unix. Une grande expressivité et liberté de codage, en échange d'une absence de garde-fous.

Le langage C est un exemple de langage faiblement typé : il permet la conversion d'entier en pointeur sans vérification aucune, tout au plus le compilateur vous avertira qu'il décline toute responsabilité en cas d'accident. L'arithmétique qu'il autorise sur les pointeurs lui confère une grande efficacité dans les parcours de données stockées dans des espaces mémoire contigus comme les tableaux, mais fait aussi sortir très rapidement de l'espace mémoire autorisé. Un langage peu ou faiblement typé comme C permet ainsi à la fois une programmation structurée d'assez haut niveau et une programmation de très bas niveau, formant ainsi un outil très apprécié pour la réalisation de couches logicielles proches du matériel, comme le système d'exploitation ou les pilotes de périphériques. Le prix de cette liberté de codage est bien sûr la gravité des erreurs d'exécution, dues par exemple à des accès hors de la mémoire ou à des tentatives d'exécuter des codes arbitraires : en effet, tout est *a priori* autorisé, et aucune vérification n'a lieu à l'exécution. Il est d'ailleurs probable que l'efficacité recherchée dans les logiciels d'aussi bas niveau est telle que les tests dynamiques qui leur confèreraient un peu plus de sécurité seraient sans doute jugés trop coûteux.

Au lieu de ne presque rien vérifier, comme le langage C (dans sa version classique), il existe des langages qui vous laissent écrire des programmes, et qui vont vérifier *à l'exécution* la validité des opérations effectuées par les programmes. C'est le cas du langage Scheme (et tous les langages Lisp en général), et c'est aussi le cas des langages de script comme Perl, Javascript, Python, Ruby et autres. L'avantage de cette technique, est qu'elle laisse la même liberté de codage que celle dont vous bénéficiez lorsque vous programmez dans un langage n'effectuant que peu ou pas du tout de vérification. Bien sûr, l'inconvénient est la pénalité en termes d'efficacité des tests nécessaires *à chaque pas d'exécution* pour provoquer un échec relativement gracieux là où le langage C non typé vous aurait laissé aller dans le mur, voire même un peu au-delà du mur ! Un langage typé dynamiquement teste que chaque addition que vous effectuez implique bien des entiers, que la procédure que vous appelez existe bien, et que l'accès à tel ou tel champ d'enregistrement est bien possible, c'est-à-dire que le champ est effectivement présent. Le coût des tests supplémentaires n'est pas le seul surcoût de cette technique : il est aussi probable que des informations relevant du typage soient présents dans les données manipulées par les programmes, de sorte à faciliter les vérifications (noms de champs d'enregistrements, par exemple).

À l'autre extrémité du spectre, se trouvent les langages à typage statique : les types de leurs programmes sont vérifiés à la compilation, une fois pour toutes, factorisant ainsi toutes les vérifications dynamiques correspondantes. L'avantage est évident en termes d'efficacité – les programmes peuvent être exempts de tests dynamiques de type –, et en termes de fiabilité. Celle-ci est en effet accrue pour

les programmes qui passent l'épreuve du typage statique : on a généralement pour eux la garantie supplémentaire que le programme échappe à toute une classe d'erreurs, suite à la vérification dont il a été l'objet. Par contre, un compilateur effectuant une vérification statique de types peut compliquer – à première vue – la tâche du programmeur, en l'empêchant d'écrire des programmes qui ne satisfont pas la discipline de typage du langage tout en étant parfaitement correct, au sens où ils ne provoqueraient pas d'erreur d'exécution. Les langages de programmation qui imposent une vérification statique des types sont donc, en première approximation, moins expressifs que les autres. En fait, ce n'est pas vrai en théorie : les langages généralistes, fussent-ils statiquement typés, sont généralement complets au sens de Turing : ils permettent d'encoder toute fonction calculable. C'est par contre quelquefois vrai dans la pratique : la vérification statique de typage peut ne pas apprécier votre style de programmation ; mais, qu'on se rassure, il existe toujours un style alternatif, ne serait-ce que par le « théorème » précédent sur l'expressivité théorique des langages.

C'est à ces techniques de vérification statique et d'inférence, que ce chapitre est consacré. Nous nous concentrerons essentiellement sur l'inférence, qui relègue la vérification au rang de sous-problème : en présence d'annotations de type en assez grand nombre, le problème de l'inférence devient un problème de propagation et de vérification des informations de typage.

7.1 Le langage PCF, et le but du typage statique

Le langage que nous allons considérer ici est le noyau fonctionnel de PCF dans lequel nous précisons un peu les constantes (entières et booléennes). La syntaxe du langage s'écrit donc :

$$
\begin{array}{rcl}
e & ::= & n \in \mathbb{N} \mid b \in \mathbb{B} \mid x \mid e_1 e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
& \mid & e_1 + e_2 \mid e_1 = e_2 \\
& \mid & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
& \mid & \mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2 \\
& \mid & \mathsf{fun}\, x \rightarrow e
\end{array}
$$

La discipline de typage que nous allons imposer aux programmes PCF est assez évidente : nous allons par exemple limiter l'emploi des opérateurs d'addition et de comparaison aux entiers, et exiger que la conditionnelle ne teste que des conditions booléennes. Bien sûr, nous devrons vérifier qu'une expression en position « fonction » dans une application a effectivement un type fonctionnel. On voit clairement que le but de ces vérifications est précisément d'éviter les tests correspondants, qui devraient être inclus dans le code produit si la vérification de typage n'était pas effectuée à la compilation. Si les tests en question sont assez faciles à énumérer et à mettre en œuvre à l'exécution, garantir leur inutilité n'est pas aussi simple. Pour que les tests qui vérifient que les valeurs que l'on applique sont toujours des fonctions, il faut que dans le programme entier, *toutes* les applications disposent de la garantie que la valeur calculée par l'évaluation de la partie « fonction » de l'application sera bien une fonction. Or, on n'a pas le droit d'effectuer des calculs à ce stade (ils pourraient ne pas terminer, ou nécessiter des lectures ou des écritures qu'il est hors de question de réaliser à la compilation. Puisqu'il est impossible de calculer les valeurs exactes à la compilation, on doit se contenter d'approximations, et c'est exactement ce que sont les types : des approximations de valeurs.

7.2 Prédire statiquement impose de faire des choix

Puisque la prédiction statique des valeurs exactes de telle ou telle expression est impossible, il faut prédire les résultats de calculs non pas comme des valeurs précises, mais comme des ensembles de valeurs possibles. Ainsi, on prédira que le résultat de $e_1 + e_2$ sera un entier (en supposant que le calcul termine), si tant est que e_1 et e_2 sont eux aussi (prédits comme étant) des entiers.

Se parer à toutes les éventualités implique aussi (et surtout) de choisir de refuser des programmes contenant des sous-expressions dont le typage ne peut prédire aisément le résultat. Par exemple, si e_2 est un entier et si e_3 est une fonction, on interdira la conditionnelle « $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ », car si cette

conditionnelle était appliquée, ou si on en additionnait le résultat avec un entier, on ne pourrait pas dire si on devait l'autoriser ou l'interdire. En résumé, une discipline de typage impose généralement que chaque sous-expression ait un type, et on va utiliser ce type pour représenter l'expression là où sa valeur peut apparaître à l'exécution.

Définir une discipline de typage, c'est donc définir d'une part quels sont les types qui sont possibles, et d'autre part comment on calcule le type de chaque expression, comment on propage l'information de type dans les programmes, et quelles sont les vérifications auxquelles doivent être soumis les programmes afin de passer avec succès l'épreuve du typage.

7.3 Les types de PCF

Définissons maintenant les types que nous souhaitons attribuer à notre langage. Clairement, nous avons besoin d'un type int pour les entiers, d'un autre que nous appellerons bool pour les booléens, et d'un *constructeur de types* à deux places que nous noterons $(\_ \to \_)$ pour les fonctions, de sorte à pouvoir construire des types fonctionnels comme $\mathsf{int} \to \mathsf{bool}$, ou $\mathsf{bool} \to (\mathsf{int} \to \mathsf{int})$. Formellement, notre algèbre de types est constituée des termes suivants :

$$\tau \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \to \tau_2$$

Nous pouvons maintenant considérer chacune des formes d'expressions constituant notre langage, et tenter d'exprimer – informellement dans un premier temps – comment attribuer un type et effectuer au passage les vérifications que nous avons mentionnées ci-dessus. Nous décrivons ci-dessous comment *calculer* les types ; leur *vérification* consistera à tester l'égalité (ou, plus généralement, la compatibilité) du type calculé avec le type attendu.

cas $n \in \mathbb{N}$ **:** clairement, cette expression est du type int ;

cas $b \in \mathbb{B}$ **:** de façon identique au cas précédent, cette expression est du type bool ;

cas $(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)$ **:** on va ici calculer le type de e_1, et exiger qu'il soit compatible avec (c'est-à-dire égal à) bool ; à la suite de quoi on va calculer les types de e_2 et e_3, et exiger qu'ils soient compatibles entre eux, c'est-à-dire vérifier qu'ils sont égaux, voire les *rendre* égaux, comme on le verra plus tard ;

cas $(e_1 + e_2)$ **et** $(e_1 = e_2)$ **:** ces deux cas sont similaires : on va calculer les types de e_1 et e_2, et les identifier à int, ensuite, on produira le type int dans le premier cas, bool dans le second ;

cas $(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$ **:** ici, on va calculer un type pour e_1 et l'associer à x durant le typage de e_2. Tout comme pour l'évaluation, le typage de e_2 nécessitera un *environnement* qui va associer un type à chacune des variables qui sont accessibles à cet endroit du programme. On va nommer Γ un tel environnement, et on le dotera d'opérations similaires à celles dont étaient équipés nos environnements d'évaluation ou d'exécution ρ.

cas x **:** puisque nous sommes maintenant munis d'un environnement de typage Γ, le type de x sera le type $\Gamma(x)$ qui lui est associé dans Γ ;

cas $(\mathsf{fun}\ x \ \to \ e)$ **:** le type de cette expression sera nécessairement un type fonctionnel $\tau_1 \to \tau_2$; même si cela a l'air un peu magique à ce stade, on peut dire que l'on va obtenir le type τ_2 comme le type de e en faisant l'hypothèse que x est de type τ_1. La question qui se pose alors est naturellement la suivante : « comment fait-on le choix de τ_1 dans ce cas ? » Nous avons laissé entendre ci-dessus que nous étions capables de nous « arranger » pour rendre deux types égaux : c'est exactement ce phénomène qui va agir ici. Un type initial sera choisi pour x, et il sera raffiné au fur et à mesure que l'on découvre les usages faits de x dans e. Ce type initial, si on ne dispose d'absolument aucune information, ce sera une inconnue de type, une variable de type α. Le processus de raffinement qui va être utilisé, basé sur la possibilité de rendre deux types égaux, sera l'*unification* que nous avons déjà vue au chapitre 6, où les termes à unifier vont représenter des types à rendre égaux.

cas $(e_1\, e_2)$ **:** le calcul du type de cette expression passe par le calcul du type de e_1, en lui imposant d'être un type fonctionnel $\tau_2 \to \tau$. On calcule aussi un type τ_2' pour e_2, et on va forcer l'égalité de

τ_2 et τ'_2. Plus précisément, on va calculer μ, le meilleur unificateur de τ_2 et τ'_2, et on va produire $\mu(\tau)$ comme type résultant.

cas (let rec $f(x) = e_1$ in e_2) **:** ce cas n'est en rien spécial, si ce n'est qu'il faut deviner le type de f et de x en calculant le type de (fun $f \to$ (fun $x \to e_1$)) et l'identifiant au type de (fun $x \to e_1$), et ensuite calculer le type de e_2 dans un environnement associant le type produit à f. On note que si le type de f n'est pas complètement déterminé lors de sa mise dans l'environnement, il pourra être automatiquement précisé durant le typage de e_2, ou, plus généralement, du contexte de cette déclaration.

Nous avons maintenant une idée assez précise du mécanisme opérationnel de synthèse de types pour PCF : on a une algèbre de types avec des variables de types jouant le rôle d'inconnues, on matérialise les contraintes de type par des équations entre types, et on fait appel à l'unification pour résoudre ces équations. Nous allons maintenant donner une présentation plus formelle de la discipline de typage de PCF, dans un formalisme proche de celui que nous avons vu au chapitre précédent : des règles d'inférence.

7.4 Inférence de types pour PCF

Notre algèbre de types s'est maintenant enrichie de variables de types et devient :

$$\tau \quad ::= \quad \alpha \mid \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \to \tau_2$$

La discipline de typage de PCF a été donnée indépendamment par Roger Hindley et Robin Milner à la fin des années 70. Le système de Milner, qui équipait la toute première version du langage ML, était doté de polymorphisme (voir section suivante), ce qui a largement contribué à son succès. Cette discipline de typage été donnée initialement dans un formalisme bien différent de celui qui est présenté ici. Celui que nous utilisons, qui est maintenant devenu classique, a été proposé initialement par Kahn *et al.* dans un article où étaient présentées des applications de la sémantique naturelle.

Tout comme nous l'avons fait au chapitre précédent pour la sémantique opérationnelle, le système de types (présentation formelle de la discipline de typage) est donné par un système de règles définissant des jugements de la forme $\Gamma \vdash e : \tau$, que l'on lit comme suit : « dans l'environnement de typage Γ, l'expression e a le type τ ». Les règles sont à rapprocher des explications que nous avons données à la section précédente, qui en forment un commentaire de nature opérationnelle.

$$\Gamma \vdash n : \mathsf{int} \qquad \Gamma \vdash b : \mathsf{bool} \qquad \frac{x \in \mathsf{dom}(\Gamma) \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash (e_1 + e_2) : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash (e_1 = e_2) : \mathsf{bool}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \oplus [x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) : \tau_2}$$

$$\frac{\Gamma \oplus [x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash (\mathsf{fun}\, x \to e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1 e_2) : \tau}$$

$$\frac{\Gamma \oplus [f : \tau \to \tau_1;\ x : \tau] \vdash e_1 : \tau_1 \qquad \Gamma \oplus [f : \tau \to \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash (\mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2) : \tau_2}$$

Ce système appelle quelques remarques :

- tout comme au chapitre précédent, il n'y a pas de règle d'erreur : pour un jugement donné, ou bien on peut construire une dérivation qui le prouve, ou alors on ne le peut pas, et le jugement sera considéré comme invalide ;
- il y a exactement une règle par construction syntaxique : on est donc assez proche d'un algorithme qui transcrirait chacune de ces règles en un cas de filtrage d'une fonction récursive, comme nous l'avons suggéré pour l'interprète du chapitre précédent ;

- dans certaines règles, on peut trouver plusieurs occurrences d'un même type[1] τ : opérationnellement, c'est l'algorithme unification qui forcera ces différents types à être identiques, forçant au passage certaines inconnues (variables de types) à prendre des valeurs particulières. La synthèse de types est donc une affaire de résolution d'équations.

7.5 Polymorphisme à la ML, ou polymorphisme paramétrique

Nous allons maintenant rapprocher le système de types de celui du noyau fonctionnel des langages de la famille ML (dont OCaml fait partie). En effet, dans le système de types présenté ci-dessus, le programme suivant n'est pas typable :

$$\mathsf{let}\ f = \mathsf{fun}\, x\ \to\ x\ \mathsf{in}\ (f\ f)(1)$$

Pour s'en rendre compte, il suffit de construire une preuve de typage pour le jugement

$$\Gamma \vdash (\mathsf{let}\ f = \mathsf{fun}\, x\ \to\ x\ \mathsf{in}\ (f\ f)(1)) : \mathsf{int}$$

où le choix de int pour le type du résultat semble judicieux, puisque l'entier 1 est le résultat de l'évaluation de ce programme. En effet, quand on tente de construire une preuve de typage pour ce jugement, on se heurte très vite à un conflit entre les types attendus pour les deux occurrences de f : la première prenant la seconde en argument, et cette dernière étant le résultat de l'application $(f\ f)$, la seconde occurrence de f devrait avoir pour type $(\mathsf{int} \to \mathsf{int})$, alors que la première devrait être de type $(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$, ce qui est impossible puisque ces deux termes ne sont pas unifiables. La solution adoptée par Milner lorsqu'il a conçu le système de types de ML, est d'autoriser dans certaines circonstances la *généralisation* de variables de types dont la spécialisation n'apporterait rien de plus. C'est le cas du type $(\alpha \to \alpha)$ qu'il est possible d'affecter à la fonction identité $(\mathsf{fun}\, x\ \to\ x)$. On pourrait bien sûr spécialiser α en un type quelconque, mais cela ne ferait que restreindre artificiellement et inutilement les utilisations possibles de cette fonction. Au lieu de cela, on peut généraliser la variable de type α et attribuer à la fonction identité le *schéma de type* $\forall\alpha.(\alpha \to \alpha)$, nous autorisant ainsi à utiliser la fonction identité avec autant de types qu'il nous plaira, pourvu que chacun d'entre eux soit une instance de ce schéma. Un tel schéma est appelé *type polymorphe*. Ce polymorphisme est appelé *polymorphisme paramétrique*, car il est basé sur ces schémas de types, où les variables quantifiées universellement sont comme des paramètres formels, qui sont remplacés par des types lors du calcul d'instances de ces schémas.

La présentation formelle du système de types pour PCF avec polymorphisme nécessite la définition des schémas, et la méthode utilisée pour instancier un schéma en un type. En rappelant la définitions des types de notre langage, nous définissons les schémas de types comme des types préfixés par la quantification d'un ensemble de variables que l'on note $\vec{\alpha}$. Lorsque l'ensemble de variables quantifiées est vide, un schéma de types n'est rien d'autre qu'un type : les schémas de types forment donc un sur-ensemble de l'ensemble des types.

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \forall\vec{\alpha}.\tau \end{array}$$

On remarque aussi que la quantification des variables n'est autorisée qu'à l'extérieur des schémas : nous n'autorisons pas ici la présence de schémas non triviaux à l'intérieur des types.

Instanciation générique Le passage d'un schéma de types à un type est appelé *instanciation générique*, et consiste à remplacer dans le corps d'un schéma, les variables quantifiées universellement par des types. On dit que τ_2 est une instance générique du schéma $\forall\vec{\alpha}.\tau_1$, et on note $\tau_2 \leq \forall\vec{\alpha}.\tau_1$, si il existe une substitution θ dont le domaine est inclus dans $\vec{\alpha}$ et telle que $\tau_2 = \theta(\tau_1)$.

1. La règle concernant les applications, par exemple, force le domaine τ_2 de la fonction à être identique au type de l'argument.

Généralisation L'opération inverse, appelée généralisation, consiste, étant donnés un type τ et un environnement de typage Γ, à identifier quelles sont les variables de types apparaissant dans τ qui sont à même d'être généralisées. La définition de la généralisation considère que seules les variables de types qui n'apparaissent pas (libres) dans l'environnement Γ sont généralisables :

$$\mathsf{gen}(\tau, \Gamma) = \forall \vec{\alpha}.\tau \text{ où } \vec{\alpha} = \mathsf{vars}(\tau) \setminus \mathsf{fv}(\Gamma)$$

où on note $\mathsf{vars}(\tau)$ l'ensemble des variables apparaissant dans τ et $\mathsf{fv}(\Gamma)$ l'ensemble des variables *libres* de Γ. Comme nous le verrons dans quelques instants, la généralisation aura lieu dès que l'on a calculé le type d'une variable définie localement, et Γ est l'environnement à l'aide duquel ce type a été calculé.

Système de règles Nous donnons ici le système de règles permettant de calculer des types avec du polymorphisme à la ML à partir des programmes PCF.

$$\Gamma \vdash n : \mathsf{int} \qquad \Gamma \vdash b : \mathsf{bool} \qquad \boxed{\frac{x \in \mathsf{dom}(\Gamma) \qquad \tau \leq \Gamma(x)}{\Gamma \vdash x : \tau}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash (e_1 + e_2) : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash (e_1 = e_2) : \mathsf{bool}} \qquad \boxed{\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \oplus [x : \mathsf{gen}(\tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \vdash (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) : \tau_2}}$$

$$\frac{\Gamma \oplus [x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash (\mathsf{fun}\, x \to e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1 e_2) : \tau}$$

$$\boxed{\frac{\Gamma \oplus [f : \tau \to \tau_1;\ x : \tau] \vdash e_1 : \tau_1 \qquad \Gamma \oplus [f : \mathsf{gen}(\tau \to \tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \vdash (\mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2) : \tau_2}}$$

Ces règles ne présentent que deux différences très légères avec le système précédent : nous avons encadré ci-dessus les règles qui ont varié. D'une part, les règles sur les déclarations locales généralisent maintenant les types qu'elles ont calculés pour la valeur déclarée localement, ce qui fait que les environnements de typage contiennent maintenant des types ou des schémas de types. D'autre part, la règle de typage des identificateurs prend maintenant des instances du schéma de type associé à la variable correspondante dans l'environnement de typage courant.

Propriétés Nous énonçons seulement la propriété essentielle de la discipline de typage de PCF, et tout-à-fait informellement, de surcroît. Étant donné un programme e, un type τ et un environnement de typage Γ, si $\Gamma \vdash e : \tau$ est valide, alors étant donné un environnement d'exécution ρ compatible avec Γ (c'est-à-dire tel que $\rho : \Gamma$, en quelque sorte), si $\rho \vdash e \Rightarrow r$, alors r est une valeur v et son type est τ. En d'autres termes, cette propriété indique que le typage de PCF prédit correctement le type des valeurs effectivement calculées par les programmes. Un corollaire de cette propriété est que l'éxécution d'un programme typable ne peut produire une erreur dynamique de type. Le typage statique élimine donc effectivement toute une classe d'erreurs.

7.6 Algorithme d'inférence

Nous terminons ce chapitre en donnant une idée de l'algorithme. Une présentation assez formelle de l'algorithme utilise un calcul d'unificateur le plus général (appels à la fonction `unif` ci-dessous), et produit, pour une expression e à typer dans un environnement de typage Γ, un type pour cette expression, ainsi qu'une substitution à appliquer à Γ pour que l'expression puisse avoir effectivement ce type.

```
let rec infer Γ e = match e with
  | « n » → (int, [ ])
  | « x » → (instance(Γ(x), [ ])
  |« e1 e2 » →
      let (τ1, θ1) = infer Γ e1 in
      let (τ2, θ2) = infer (θ1(Γ)) e2 in
      let τ3 = nouvelle variable in
      let θ3 = unif (θ2(τ1)) (τ2 → τ3)) in
      (θ3(τ3), θ3 ∘ θ2 ∘ θ1)
  | « (fun x → e) » →
      let τ0 = nouvelle variable in
      let (τ1, θ1) = infer (Γ ⊕ [x : τ0]) e in
      ((θ1(τ0) → τ1), θ1)
  | « (let x = e1 in e2) » →
      let (τ1, θ1) = infer Γ e1 in
      let (τ2, θ2) = infer (θ1(Γ) ⊕ [x : gen(τ1, θ1(Γ))]) e2 in
      (τ2, θ2 ∘ θ1)
  | ...
```

L'algorithme est correct : si `infer` $\Gamma\ e{=}(\tau, \theta)$, alors $\theta(\Gamma) \vdash e : \tau$.

L'algorithme est complet : pour un programme typable, l'algorithme va calculer son type *le plus général*, c'est-à-dire celui dont tous ses autres types sont des instances.

Synthèse destructive Pour conclure, il nous faut noter que les algorithmes d'inférence effectifs sont très nettement optimisés par rapport à celui ci-dessus. Ces algorithmes utilisent notamment une unification *destructive* qui égalise physiquement deux types à unifier, au lieu de calculer une substitution.

Chapitre 8

Les objets

L'augmentation en complexité des structures de données utilisées dans un programme motive rapidement la recherche de constructions permettant d'agréger des données de base. Plusieurs constructions existent à cet effet, dont certaines ont été vues au cours des chapitres précédents. Nous allons revoir/voir dans ce chapitre ces diverses constructions.

Plus encore, il devient rapidement nécessaire de regrouper ces données avec les fonctions qui permettent de les manipuler. Deux écoles existent principalement, les *modules* et les *objets* et peuvent être vues comme des généralisations de la notion d'*enregistrements* (*records* en Anglais), c'est-à-dire de "structures" à la C.

Nous n'étudierons pas dans ce chapitre le mécanisme des modules en détail et nous concentrerons sur les enregistrements, puis sur les objets en tant que généralisation des premiers.

8.1 Quelques notes sur les modules

Pour donner une intuition du mécanisme de modules, il suffit d'imaginer un regroupement de définitions (valeurs, types, exceptions, modules) accessibles au travers du nom donné à ce regroupement. Ainsi, un module définit un espace de nommage qui lui est propre et encapsule toutes les définitions qui s'y trouvent.

```
module EvenInt = struct
  type t = int
  exception Not_even
  let from_int n = if not ((n mod 2) = 0) then raise Not_even else n
  let to_int e = e
  let add e1 e2 = e1 + e2
end
```

FIGURE 8.1 – Un module encapsulant des entiers pairs

Exemple Ce module (figure 8.1) permet de regrouper toutes les fonctions disponibles pour traiter des entiers naturels pairs. On remarque la présence d'une fonction de "création" d'entiers pairs à partir d'entiers naturels arbitraires : `from_int`. Cette fonction a en charge la vérification que l'entier à "convertir" en "entier pair" est effectivement pair. Dans le cas contraire elle lève une erreur. Une fois

ce module défini, il est utilisable en désignant ses définitions par la notation pointée : nom du module suivi du caractère "." suivi du nom de la définition :

```
# EvenInt.from_int 42 ;;
- : int = 42
# EvenInt.from_int 43 ;;
Exception: EvenInt.Not_even.
```

FIGURE 8.2 – Accès aux définitions d'un module par notation pointée

Un module possède naturellement une *interface* (l'équivalent de son type) qui regroupe les types de ses définitions (c.f. figure 8.3) :

```
module EvenInt : sig
    type t = int
    exception Not_even
    val from_int : int → int
    val to_int : α → α
    val add : int → int → int
  end
```

FIGURE 8.3 – Signature la plus générale du module `EvenInt`

Pour autant, par défaut celle-ci exporte toutes les définitions du module et y donne donc accès, y compris à la structure des types de données contenus. Dans l'exemple ci-dessus, il est clairement visible que le type `t` représentant les entiers pairs est un simple `int`. De ce fait, le programmeur peut passer outre les seules fonctions de manipulation fournies par le module et par là même, briser les éventuels invariants nécessaires au bon fonctionnement du module.

Pour palier ce problème, il est possible d'assigner à un module une signature plus restreinte n'exportant que certaines définitions ou en rendant abstraites (c'est-à-dire en cachant l'implémentation). C'est le cas dans la signature suivante (figure 8.4) où le type `t` est rendu abstrait, la fonction `add` n'est pas exportée et les fonctions `to_int` et `from_int` rendent et prennent respectivement une valeur de type `t` et non plus `int`, empêchant ainsi le mélange entre ces 2 types.

```
module type EvenIntSig = sig
    type t
    exception Not_even
    val from_int : int → t
    val to_int : t → int
  end
module EvenInt : EvenIntSig = struct ... end
```

FIGURE 8.4 – Restriction de la signature du module `EvenInt`

De cette manière, il devient obligatoire de manipuler les données encapsulées dans le module via les fonctions qu'il met à disposition.

Les systèmes de modules évolués comme ceux d'OCaml fournissent d'autres constructions que nous passons ici sous silence. Nous mentionnons toutefois la notion de *foncteur*, en quelque sorte une "fonction des modules vers les modules". Un foncteur prend donc en argument un module et retourne un module. Un foncteur étant lui même un module, le système de modules obtenu est dit *d'ordre supérieur* : un foncteur peut prendre en argument ou retourner en résultat un foncteur.

8.2 Les enregistrements

Comme nous l'avons évoqué en introduction et esquissé via les modules, le besoin d'agglomérer des données est fondamental pour l'élaboration de données de plus en plus complexes et la représentation d'informations structurées.

8.2.1 Introduction par le besoin et l'exemple

Le moyen le plus simple à notre disposition est la structure de *couple*, généralisée en *n-uplet*. Un couple est la juxtaposition de 2 valeurs. On dispose donc d'un constructeur et de deux projections, droite et gauche, permettant d'extraire la composante droite et la composante gauche d'un couple. L'exemple suivant représente la position de l'ENSTA par sa latitude (48,834279) et sa longitude (2,283596) sous forme d'un couple en OCaml. On notera les deux projections `fst` et `snd` prédéfinies en OCaml et la manière de les implanter (`myfst` et `mysnd`).

```
# let ensta = (48.834279, 2.283596) ;;
val ensta : float * float = (48.834279, 2.283596)

# fst ensta ;;
- : float = 48.834279
# snd ensta ;;
- : float = 2.283596

# let myfst = function (l, _) → l ;;
val myfst : α * β → α = <fun>
# let mysnd = function (_, r) → r ;;
val mysnd : α * β → β = <fun>

# myfst ensta ;;
- : float = 48.834279
# mysnd ensta ;;
- : float = 2.283596
```

FIGURE 8.5 – Exemple de couple en OCaml

Les n-uplet (ou *tuples*) sont une généralisation qui permet à la structure de donnée d'avoir un nombre arbitraire de composantes. On notera qu'il n'existe plus alors d'opérateur de projection prédéfini. En terme de typage, deux n-uplets sont compatibles si leur composantes sont compatibles deux à deux positionnellement. De ce fait, ils doivent évidemment avoir le même nombre de composantes. Ceci signifie que le type des "*4-uplets*" n'est pas le même que celui des "*42-uplets*".

```
# type n3_uplet = (int * bool * char) ;;
type n3_uplet = int * bool * char
# type n4_uplet = (int * bool * char * string) ;;
type n4_uplet = int * bool * char * string

# let v3_uplet = (1, true, char) ;;
# (v3_uplet : n3_uplet) ;;
- : n3_uplet = (1, true, 'A')

# let v4_uplet = (1, true, 'A', "O_o") ;;
val v4_uplet : int * bool * char * string = (1, true, 'A', "O_o")

# v3_uplet = v4_uplet ;;
Error: This expression has type int * bool * char * string
       but an expression was expected of type int * bool * char
```

FIGURE 8.6 – N-uplets en OCaml

La structuration en n-uplets présente toutefois un certain nombre d'inconvénients. En effet, les composantes sont repérées positionnellement et il devient rapidement difficile de se souvenir quelle composante correspond à quelle information ?

La définition **type** `event` = (`int` * `int` * `int` * `int`) est-elle suffisamment explicite pour dénoter le type d'un événement repéré dans l'espace et dans le temps par sa position en x, y, z, et sa date en secondes ? Lors de l'utilisation d'une telle structure de donnée il est très facile de se tromper de projection et de manipuler une autre composante que celle à laquelle on pensait :

```
# let speed ev1 ev2 =
    let (x1, y1, z1, t1) = ev1 in
    let (x2, y2, t2, z2) = ev2 in      (* !!! Confondus z2 et t2 !!! *)
    let dx = x2 - x1 in
    let dy = y2 - y1 in
    let dz = z2 - z1 in
    let delta = sqrt (float_of_int (dx * dx + dy * dy + dz * dz)) in
    delta /. (float_of_int (t2 - t1)) ;;

val speed : int * int * int * int -> int * int * int * int -> float = <fun>
```

FIGURE 8.7 – N-uplets et erreurs faciles

et pour autant, le type de notre fonction de calcul de vitesse reste visiblement correct !

D'autre part, comment faire la différence entre 2 n-uplets de même type mais ne représentant pas les mêmes genres d'informations ?

L'exemple 8.8 montre que du moment que les n-uplets sont structurellement compatibles et non abstraits, rien n'empêche leur confusion.

Pour pallier ces deux inconvénients, nous introduisons maintenant la notion d'*enregistrement* qui peut être vu comme un type de n-uplet dont chaque composante est nommée (ce que nous appellerons aussi

```
# type position = (int * int * int) ;;          (* x, y, z *)
# type date = (int * int * int) ;;              (* Jour, mois, année *)

# let (p : position) = (1, 0, 0) ;;
val p : position = (1, 0, 0)
# let (d : date) = (21, 12, 2012) ;;
val d : date = (21, 12, 2012)

# let next_year (d, m, y) : date = (d, m, y + 1) ;;
val next_year : int * int * int → date = <fun>

# next_year p ;;                                (* Année suivante sur ... une position ! *)
- : date = (1, 0, 1)
```

FIGURE 8.8 – Deux n-uplets compatibles en type mais de natures différentes

un *champ*. Ceci correspond donc aux `struct` que l'on retrouve en langage C. L'accès à une composante d'un enregistrement se fait en utilisant la notation pointée : valeur suivi du caractère "." suivit du nom de la composante.

```
# type position = { x : int ; y : int ; z : int } ;;
# type date = { day : int ; month : int ; year : int } ;;

# let p = { x = 1 ; y = 0 ; z = 0 } ;;
val p : position = {x = 1; y = 0; z = 0}
# let d = { day =21 ; month = 12 ; year = 2012 } ;;
val d : date = {day = 21; month = 12; year = 2012}

# let next_year d = { day = d.day ; month = d.month ; year = d.year + 1 } ;;
val next_year : date → date = <fun>

# next_year p ;;
Error: This expression has type position
       but an expression was expected of type date
```

FIGURE 8.9 – Définition d'enregistrements en OCaml

Il est important de noter que les enregistrements tels que nous allons les étudier et tels qu'ils sont disponibles en OCaml reposent sur la propriété qu'un nom de champ n'appartient qu'à un seul type enregistrement. C'est en particulier ce qui permet à l'inférence de type de déterminer sans aucune annotation que les valeurs *d* et *p* de notre précédent exemple sont respectivement de type *position* et *date*. La présence dans un enregistrement d'un champ détermine directement le type de cet enregistrement.

L'énumération de tous les champs d'un enregistrement lors d'une recopie, éventuellement partielle est fastidieuse et peut être évitée grâce à la construction `with` qui permet de créer une nouvelle valeur enregistrement à partir d'une pré-existante en conservant la valeurs des champs non redéfinis.

```
# let next_year d = { d with year = d.year + 1 } ;;
val next_year : date → date = <fun>
```

FIGURE 8.10 – Recopie (partielle) d'enregistrement en OCaml

8.2.2 Syntaxe abstraite des enregistrements

Nous étendons maintenant la syntaxe de PCF avec les 3 constructions permettant de gérer les enregistrements : la création d'enregistrement, l'accès à un champ et la recopie (partielle).

$$
\begin{array}{rcl}
e & ::= & n \in \mathbb{N} \mid b \in \mathbb{B} \mid x \mid e_1 e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
& \mid & e_1 + e_2 \mid e_1 = e_2 \mid \mathsf{fun}\, x\ \rightarrow\ e \\
& \mid & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2 \\
& & /* --- Nouveau --- */ \\
& \mid & \{\ell_1 = e_1; \ldots; \ell_n = e_n\} \\
& \mid & e.\ell \\
& \mid & \{e\ \mathsf{with}\ \ell = e'\}
\end{array}
$$

FIGURE 8.11 – Syntaxe abstraite de PCF avec enregistrements

Comme mentionné précédemment, on voit qu'aucune expression d'enregistrement ne fait apparaître d'annotation de type : ce dernier est implicitement déduit par la présence de noms de champs et leur l'appartenance à une définition d'enregistrement donnée.

8.2.3 Sémantique opérationnelle des enregistrements

Une fois la syntaxe étendue, il convient d'étendre également le sens que l'on donne aux nouvelles constructions. Nous le ferons ici par le biais d'une sémantique opérationnelle, c'est-à-dire en explicitant la manière dont se déroule le calcul des expressions ayant trait aux enregistrements (c.f. chapitre 5).

On étend l'ensemble des valeurs résultant d'évaluations correctes de la manière suivante en rajoutant la description des valeurs de type enregistrement.

$$
\begin{array}{rcl}
v & ::= & c \mid \mathsf{Valf}(x, e, \rho) \mid \mathsf{Valfr}(f, x, e, \rho) \\
 & & /* --- Nouveau --- */ \\
 & \mid & \mathsf{Record}((\ell_1, v_1), \ldots, (\ell_n, v_n))
\end{array}
$$

FIGURE 8.12 – Valeurs de PCF avec enregistrements

Nous pouvons donner maintenant les règles d'évaluation des expressions d'enregistrement, d'accès à un champ et de recopie en utilisant des jugements tels qu'introduits au chapitre 5.1.

$$
\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \ldots \rho \vdash e_n \Rightarrow v_n}{\rho \vdash \{\ell_1 = e_1; \ldots; \ell_n = e_n\} \Rightarrow \mathsf{Record}((\ell_1, v_1), \ldots, (\ell_n, v_n))}(Rec)
\qquad
\frac{\rho \vdash e \Rightarrow \mathsf{Record}(\ldots, (\ell, v), \ldots)}{\rho \vdash e.\ell \Rightarrow v}(RAcc)
$$

$$
\frac{\rho \vdash e \Rightarrow \mathsf{Record}((\ell_1, v_1), \ldots, (\ell, v), \ldots (\ell_n, v_n)) \qquad \rho \vdash e' \Rightarrow v'}{\rho \vdash \{e \text{ with } \ell = e'\} \Rightarrow \mathsf{Record}((\ell_1, v_1), \ldots, (\ell, v'), \ldots (\ell_n, v_n))}(RPCpy)
$$

FIGURE 8.13 – Sémantique opérationnelle de PCF avec enregistrements

La règle *Rec* exprime simplement que l'évaluation d'une expression d'enregistrement ayant des sous-expressions liées à des noms de champs s'évalue en une valeur enregistrement ayant ces mêmes noms de champs et étant liés à la valeur résultant de l'évaluation de leur sous-expression respective.

La règle *RAcc* qui représente la sémantique de l'accès à un champ de l'enregistrement exprime que l'évaluation de l'expression qualifiée par la notation pointée doit être une valeur de type enregistrement et possédant le champ utilisé pour l'accès. La valeur résultant est alors celle liée à ce champ dans la valeur enregistrement. Cette règle impose implicitement deux contraintes : la valeur dans laquelle on recherche le champ doit être un enregistrement et cet enregistrement doit héberger le champ recherché.

Pour terminer, la règle *RPCpy* donne la sémantique de la recopie partielle d'un enregistrement. Nous mentionnons ici explicitement "partielle" car telle que présentée dans notre syntaxe, si un enregistrement comporte plusieurs champs, alors la construction `with` recopiera tous les autres champs à l'exception de celui redéfini. Donc la copie sera alors bien "partielle". Cette règle exprime donc simplement que les champs de noms différents de l dans la valeur d'enregistrement à recopier resteront inchangés alors que celui redéfini se verra lié à la valeur résultant de l'évaluation de e'.

L'expression de recopie peut aisément être généralisée afin d'autoriser la redéfinition de plusieurs champs au lieu d'un seul tel que défini dans la syntaxe abstraite de notre langage. On écrira :

$$\{e \text{ with } \ell_1 = e_1; \ldots; \ell_n = e_n\}$$

pour :

$$\{\ldots\{e \text{ with } \ell_1 = e_1\} \ldots \ell_n = e_n\}$$

Cette extension de syntaxe est en réalité un ajout qui ne change pas la sémantique du langage ni ne l'étend. Elle permet seulement d'augmenter la facilité d'expression de programmes sans rajouter de fonctionnalité. Ce n'est qu'une écriture simplifiée pour une combinaison de constructions existantes. Ce genre d'extension est habituellement appelée "sucre syntaxique" (ou *syntactic sugar* en Anglais). La sémantique de la forme abrégée est exactement la même que celle de la forme complète, "expansée". Il est fréquent d'avoir recours à ce genre d'astuce lorsque l'on souhaite offrir des raccourcis, des généralisations pour des constructions qui seraient fastidieuses à écrire pour l'utilisateur du langage.

8.3 Des enregistrements aux variables mutables

Nous avons jusqu'à présent décrit PCF en martelant que les `let` établissaient une liaison **constante** entre un identificateur et une valeur : **let** `x` = 3 **in** ... `expr` ... lie la valeur 3 à l'identificateur x et cette valeur est définitivement fixée. Toute autre définition **let** `x` = ... se trouvant à l'intérieur de l'expression ...*expr*... définit un "nouvel" x. Nous souhaitons désormais étendre notre langage avec des affectations, c'est-à-dire des modifications en place de la valeur liée à une variable (autrement dit, le `... = ... ;` du langage C).

Ceci peut être réalisé de deux manières. Soit en introduisant explicitement une notion de *location* comme vu au chapitre 4.5, soit en considérant que les champs d'enregistrement peuvent être qualifiés de *mutables*, c'est-à-dire introduire eux-même la notion de location. Quelque soit la façon choisie, on arrive à un même mécanisme. En OCaml, le type *ref* qui représente les références est en fait un type d'enregistrement dont le seul champ est qualifié mutable :

```
# type α ref = { mutable contents : α } ;;
# let ( :=) r v = r.contents ← v ;;
val ( := ) : α ref → α → unit = <fun>
# let ( ! ) r = r.contents ;;
val ( ! ) : α ref → α = <fun>
```

FIGURE 8.14 – Références en OCaml

8.4 Les objets

Dans les sections précédentes, nous avons vu des moyens de regrouper des données. Grâce aux modules un moyen de regrouper des définitions ayant un rapport entre elles. Grâce aux enregistrements nous avons vu un moyen d'agréger des données pour construire des structures de données plus complexes. Pour autant, nous n'avons pas encore trouvé un moyen d'agréger à la fois des données et les fonctions qui les manipulent au sein d'une même entité. Le paradigme "objet" se propose de réaliser une telle unification.

Exemple : un point (2D) est donné par
- son abscisse `x`,
- son ordonnée `y`,
- une méthode de déplacement, qui modifie ses coordonnées.

8.4.1 Objet par enregistrement : 1er essai

On souhaite regrouper toutes les *attributs* (variables propres à l'objet) au sein d'un enregistrement ainsi que les *méthodes* (c'est-à-dire des fonctions) qui permettent d'y accéder. Ceci semble une idée satisfaisant à la fois notre souhait de regrouper données et fonctions sur ces données.

Comme le montre la tentative de modélisation de la figure 8.15, une représentation naïve par un simple enregistrement contenant autant de champs que de variables internes de l'objet et idem pour les méthodes censées permettre de les manipulater ne fonctionne pas car les champs d'un enregistrement n'étant pas mutuellement récursifs (et *a fortiori* inconnus les uns des autres), il est impossible dans la définition d'un champ de faire référence à un autre champ : x et y sont inconnus de *move*.

D'autre part, quand bien même x et y seraient connus de *move*, si cette dernière accédait directement aux variables x et y de l'objet (donc de l'enregistrement), alors son code ne pourrait pas être partagé

```
# type point = {
    x : int ref ;
    y : int ref ;
    move : int → int → unit
} ;;

# let p = {
    x = ref 3 ;
    y = ref 4 ;
    move =
      fun dx dy →
        x := !x + dx ;
        y := !y + dy
    } ;;
Error: Unbound value x
```

FIGURE 8.15 – Des objets avec des enregistrements (incorrect)

entre différents objets. En effet, dans le code de *move*, l'accès aux locations de x et y (en d'autres termes, à leurs *adresses*) impose que celles-ci figurent explicitement ("textuellement") dans le code compilé pour la fonction *move* propre à chaque valeur d'enregistrement.

8.4.2 Objet par enregistrement : amélioration...

Une solution consiste à paramétrer les méthodes de l'objet, donc les fonctions de l'enregistrement par l'objet courant. Ceci permet effectivement d'accéder aux champs de cet objet puisqu'il est explicitement fourni en argument des méthodes. Qui plus est, le code des méthodes devient alors générique pour tous les objets générés puisque chacun d'entre eux sera explicitement passé aux méthodes qui lui appartiennent et qui sont invoquées sur lui. La figure 8.16 montre une telle modélisation des objets en utilisant les enregistrements et dans laquelle l'objet courant est nommé *this* (dans la littérature et les différents langages, on trouve également *self*, *me*, *myself*).

L'appel de la méthode *move* du point p de l'exemple 8.16 sera fera donc par : `p.move p 1 1`. La mention explicite de *this* à chaque appel de méthode est fastidieuse. Il est alors souhaitable d'introduire une syntaxe permettant d'alléger l'écriture. En Ocaml, l'appel de méthode est dénoté par le signe `#` : `p#move 1 1`.

Cette syntaxe, `e#m`, peut être vue comme du sucre syntaxique donnant lieu à une réécriture de la forme : **let** `this = e` **in** `this.m this`.

8.4.3 Problème de polymorphisme

En conception orientée objet, il est admis que l'appel d'une méthode sur un objet est valable du moment que cet objet dispose de cette méthode, quelque soit les éventuelles autres qu'il possède. En d'autres termes, si un objet "point" et un objet "point coloré" possèdent tout deux une méthode *move*, alors il est possible d'appliquer *move* sur chacun d'eux, même si le "point coloré" possède des méthodes supplémentaires (par exemple *get_color*). L'idée est que tant qu'un objet satisfait une *interface* minimale requise pour une invocation de méthode, il est un candidat sûr. Typiquement, on souhaite écrire des fonctions de la forme :

Nous reprenons maintenant la technique de modélisation des objets basée sur les enregistrements vue dans la section précédente (8.4.2) et tentons de l'appliquer sur notre problème.

```
type point = {
    x : int ref ;
    y : int ref ;
    move : point → int → int → unit
} ;;

# let p = {
    x = ref 3 ;
    y = ref 4 ;
    move =
      fun this dx dy →
        this.x := !(this.x) + dx ;
        this.y := !(this.y) + dy  } ;;
val p : point = {x = {contents = 3}; y = {contents = 4}; move = <fun>}
```

FIGURE 8.16 – Des objets avec des enregistrements (mieux)

```
let p1 = { x = ref 4; y = ref 2; move = ... } ;;
let p2 = { x = ref 1; y = ref 3; move = ...; c = ref "rouge" } ;;
let do_move p dx dy = p#move dx dy ;;

do_move p1 1 0 ; do_move p2 0 1 ;;
```

Puisque le nom des champs d'enregistrements n'appartiennent qu'à un seul type d'enregistrement, la définition de *colored_point* vient masquer les champs *x*, *y* et *move* de *point*. Ainsi, en voulant définir une valeur de type *point* en ne donnant que des valeurs pour ces 3 champs, puisqu'ils sont maintenant hébergés par *colored_point*, la définition sera considérée comme de ce type, et il manque donc la spécification des champs *color* et *get_color*.

Ainsi, cet exemple montre la limitation imposée par l'encodage des objets par des enregistrements puis qu'il nous est impossible de créer des objets de types différents mais ayant des champs en commun, et *a fortiori* ayant un sous-ensemble des champs d'un autre objet. Il est bien évidemment impossible de vouloir contourner ce problème en renommant les champs du second type d'objet puisqu'il serait impossible d'écrire une fonction appliquant *move* sur les deux types d'objet, ce champ ayant alors un nom différent dans les deux objets.

Sans entrer dans des détails trop techniques, il existe un moyen de contourner ce problème en étendant le système de types du langage par le biais de *rangées* qui permettent l'introduction d'enregistrements extensibles. Intuitivement, les noms de champs ne sont plus l'exclusivité d'un et un seul type, les enregistrements deviennent une liste *ouverte* de champs qui peut s'allonger à la demande grâce à la présence d'une variable terminale. La compatibilité entre types enregistrements s'appuie alors sur la structure (*sous-typage structurel*) et non plus sur les noms des types. Plus de détails ainsi que des formalisations rigoureuses peuvent être trouvés dans les travaux de Didier Rémy (http ://pauillac.inria.fr/~remy/publications.html, "*Type Inference for Records in a natural Extension of ML*", "*Efficient Representation of Extensible Records*", etc.). Cette technique est d'ailleurs une de celles qui entrent en jeu dans les objets d'OCaml.

```
# type point = {
      x : int ref ;
      y : int ref ;
      move : point → int → int → unit
} ;;

# type colored_point = {
      x : int ref ;
      y : int ref ;
      color : (int * int * int) ;
      move : colored_point →int →int → unit ;
      get_color : colored_point →(int * int * int)
} ;;

# let p = {
    x = ref 3 ;
    y = ref 4 ;
    move =
      fun this dx dy →
        this.x := !(this.x) + dx ;
        this.y := !(this.y) + dy  } ;;

Error: Some record field labels are undefined: color get_color
```

FIGURE 8.17 – Problème de noms de champs

8.4.4 Objets et classes

Dans la présentation qui précède, nous avons parlé des objets comme structures agglomérant des définitions de données et de méthodes. Il est intéressant de remarquer au passage que la différence entre "donnée" et "méthode" est ténue puisqu'une donnée n'est qu'une méthode constante. La différence tient principalement en ce que les données sont (généralement) propres à chaque objet alors que les méthodes *peuvent* (ce n'est pas une obligation) être partagées par tous les objets de même type.

Puisque l'on parle de "tous les objets de même type", ceci pousse à penser que les objets d'un certain type sont le produit d'un "générateur d'objets de ce type". On peut effectivement, au lieu de construire "à la main" des objets, définir une fonction permettant la création d'objets :

On constate que nos objets disposent désormais de variables internes (souvent appelées variables *d'instance*), de méthodes et de constructeurs. Fort logiquement on souhaite regrouper tous ces aspects au sein d'une seule et même entité qui permet de générer à la fois des objets et leurs méthodes associées. C'est le rôle des *classes* de servir de tels "générateurs". On dira alors qu'un objet est une *instance* d'une classe dès lors qu'il est issu de ce générateur. Dans le code montré en 8.18, la fonction *class_point* peut être vue comme la classe représentant les objets points. On remarquera que la fonction *class_point_init* peut se revendiquer également le droit d'être considérée comme une classe en ce sens qu'elle crée également des objets *point*. Toutefois, celle-ci pourrait simplement être énoncée comme l'appel à *class_point* suivi d'un appel à la méthode *move* de l'objet obtenu :

En OCaml, de la syntaxe a été introduite pour dénoter l'appel au constructeur d'une classe (afin de créer un objet de cette classe). Dans les exemples qui précèdent, au lieu d'écrire : **let** `p = class_point ()` **in** ... on écrira **let** `p =` **new** `class_point` **in** ...

```
# type point = {
      x : int ref ;
      y : int ref ;
      move : point → int → int → unit
  } ;;

# let move o dx dy =
     o.x := !(o.x) + dx ;
     o.y := !(o.y) + dy ;;
val move : point → int → int → unit = <fun>

(* Constructeur par défaut, sans initialisation explicite. *)
# let class_point () = { x = ref 0 ; y = ref 0 ; move = fun this → move this } ;;
val class_point : unit →point = <fun>
(* Constructeur avec initialisation explicite. *)
# let class_point_init ix iy = {
   x = ref ix ; y = ref iy ; move = fun this →move this } ;;
val class_point_init : int →int →point = <fun>

# let p = class_point_init 42 46 ;;
val p : point = {x = {contents = 42}; y = {contents = 46}; move = <fun>}
# p.move p 4 7 ;;
- : unit = ()
# p ;;
- : point = {x = {contents = 46}; y = {contents = 53}; move = <fun>}
```

FIGURE 8.18 – Constructeurs d'objets

```
# let class_point_init ix iy =
    let p = class_point () in
    p.move p ix iy ;
    p ;;
val class_point_init : int →int →point = <fun>
```

8.4.5 Héritage

Il arrive couramment de dériver une structure de donnée plus riche d'une précédemment existante. Nous avons évoqué le cas dans l'exemple 8.17 où nous souhaitions enrichir le type de *point* avec une couleur pour obtenir des *colored_point*. L'idée est de ne pas avoir à redéfinir les méthodes déjà disponibles et qui fonctionneront dans le cadre des *colored_point* (par exemple *move*).

Différemment, on peut souhaiter définir une notion de point en 3 dimensions à partir de points en 2 dimensions en réutilisant les méthodes déjà disponibles sur les points en 2 dimensions pour écrire leurs équivalentes en 3 dimensions.

En fonction des langages, l'héritage simple ou multiple est autorisé. Dans le second cas, il est possible pour une classe d'hériter de plusieurs autres. C'est le cas entre autres pour C++, Python, Eiffel, OCaml, FoCaL. De ce fait, si plusieurs "ancêtres" disposent d'une même méthode, une règle de résolution (i.e.

```
let class2Dpoint (xinit, yinit) =
  { x = ref xinit; y = ref yinit ;
    move this dx dy = ... ;
    reset this = (this.x := 0 ; this.y := 0) }

let class3Dpoint (xinit, yinit, zinit) =
  let super = new class2Dpoint (xinit, yinit) in
  { x = super.x ; y = super.y ; z = ref zinit ;
    move this dx dy dz =
       (super.move this dx dy ;
        this.z := !this.z + dz) ;
    reset this = (super.reset this; this.z := 0) }
```

de choix) de la version de la méthode à conserver devra être énoncée afin de résoudre le conflit.

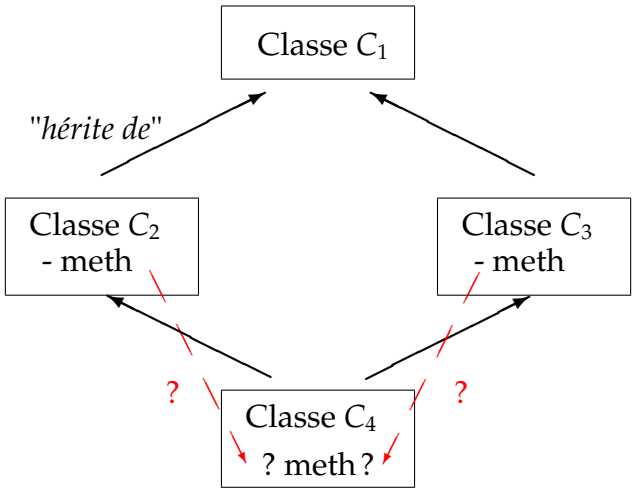


FIGURE 8.19 – Problème "du diamant" ou résolution de l'héritage multiple

Java de son côté a évité ce problème en interdisant l'héritage multiple mais en autorisant une classe à implémenter plusieurs *interfaces*. Les interfaces sont des formes de signatures et ne contiennent donc que des méthodes abstraites, c'est-à-dire pas de code exécutable. C'est alors à la classe elle-même d'implémenter toutes les méthodes de toutes ses interfaces.

8.4.6 Liaison tardive

Nous avons vu que par héritage, une classe peut redéfinir une méthode. D'autre part, des objets de classes héritant (à un ou plusieurs niveaux) d'une même classe sont "compatibles" avec les objets de cette classe parente. Ainsi, une liste de *class2Dpoint* peut *a fortiori* contenir des objets *class3Dpoint* puisque cette liste étant considérée d'objets de type $class2Dpoint \leq class3Dpoint$, ses éléments ne seront autorisés à être manipulés que par des méthodes de la classe *class2Dpoint*, méthodes forcément contenues dans *class3Dpoint* puisqu'elle hérite de *class2Dpoint*.

Si l'on considère maintenant une liste "hétérogène" comportant des objets de ces deux classes :
`let l = [new class2Dpoint ; new class3Dpoint ]`

puis applique sur chacun de ces éléments une fonction qui invoque sa méthode *reset* :
`List.iter (fun o → o#reset) l`
quelle sera la méthode *reset* appelée ? Celle de *class2Dpoint* ou celle de *class3Dpoint* ? Si l'on se place d'un point de vue simpliste, on peut se dire que le type des éléments de la liste sera le "plus petit" de tous ses éléments, donc *class2Dpoint*, et que donc, ce sera la méthode *reset* de cette classe qui sera invoquée.

Le modèle objet autorise la notion de *liaison tardive* (ou *late binding* en Anglais, grâce à laquelle la méthode appelée dépend de la classe **effective** de l'objet sur lequel elle est invoquée. Ainsi, la méthode sera sélectionnée *dynamiquement* en fonction de la classe de chaque objet comme le montre l'exemple de la figure 8.20.

```
# class foo =
    object
    method p = Printf.printf "foo\n"
    end ;;
class foo : object method p : unit end
# class bar =
    object
    inherit foo
    method p = Printf.printf "bar\n"
    end ;;
class bar : object method p : unit end

# let l = [new foo ; new bar] ;;
val l : foo list = [<obj>; <obj>]

# List.iter (fun o → o#p) l ;;
foo
bar
- : unit = ()
```

FIGURE 8.20 – Liaison tardive en OCaml

La fonctionnalité de liaison tardive implique qu'il n'est pas (toujours) possible de déterminer statiquement, c'est-à-dire lors de la compilation, quelle sera la méthode à appeler. Il est donc nécessaire de pouvoir la retrouver à l'exécution. La question de savoir où sont "logées" les méthodes prend alors tout son sens. Sont-elles stockées dans les objets eux-mêmes ou sont-elles stockées dans la classe ? La réponse varie en fonction des langages et de leurs choix d'implémentation. En OCaml, JavaScript, les méthodes sont dans les objets alors qu'en C++ chaque objet dispose d'un pointeur vers la table des méthodes de sa classe (*virtual table*).

8.5 Conclusion

Nous avons étudié ici quelques notions, les plus répandues et les plus simples des langages objets. D'autres traits plus flexibles et/ou plus poussés existent comme les initialiseurs, les destructeurs, des notions de visibilité des méthodes, des modification du partage des méthodes et des données, les classes paramétrées, les méthodes polymorphes, la coercion de type, les méthodes virtuelles, etc.

Certains aspects dynamiques s'avèrent difficiles à compiler efficacement. L'encodage que nous avons vu ici à partir d'enregistrements peut servir de base mais il nécessite rapidement des extensions plus complexes.

De même, le typage et l'inférence de types deviennent nettement plus complexes que celle de langages (fonctionnels ou non) simples.

Pour autant, la réalisation d'un interprète de PCF étendu par des objets "simples" s'avère relativement aisée, la sémantique correspondant à l'héritage simple et la liaison tardive restant relativement intuitive. Ce n'est bien sûr pas le cas dès lors que l'on décide d'étendre celle-ci aux notions supplémentaires que nous avons évoquées au début de cette conclusion.

Chapitre 9

Gestion de la mémoire

Durant l'exécution d'un programme, les éléments manipulés par le programme sont stockés dans différents types de mémoire. La **pile d'exécution** contient des éléments dont la durée de vie n'excède pas la durée d'un appel de fonction. La pile contient justement, en général, les différents éléments relatifs à un tel appel : les arguments, les variables locales, les adresses de retour, *etc.*

La mémoire allouée statiquement est elle aussi utilisée par des langages de programmation comme le langage C, lorsque, par exemple, on y déclare en dehors d'une fonction :

```
int tab[ ] = { 1024, 12345, −256 }
```

Le compilateur va alors réserver l'espace nécessaire à la représentation du tableau `tab` dans une zone mémoire dite *statique*, et sa durée de vie est l'exécution du programme.

La mémoire dite *dynamiquement* allouée va, quant à elle, être utilisée pour représenter toutes les données dont la création a été demandée par le programme durant son exécution. Lorsque les données en question ne survivent pas à l'appel de fonction durant lequel elles ont été créées (variables locales, paramètres), on peut les allouer dans la *pile* à l'entrée dans la fonction, et les désallouer de la pile au retour de la foncion. Ainsi, les variables locales des fonctions ou procédures ne pouvant survivre à l'appel qui les crée ne sont accessibles qu'à l'intérieur de leur portée.

Lorsque la durée de vie de données ne peut pas être *a priori* bornée, la mémoire utilisée pour les représenter est allouée à partir d'une zone d'allocation appelée le *tas* (*heap*, en anglais). C'est le tas qui est utilisé lorsqu'un programme C exécute un appel à la fonction `malloc`, ou lorsqu'un programme C++ appelle **new** ou encore quand un programme OCaml construit une valeur comme un enregistrement ou une valeur fonctionnelle. La mémoire allouée dans le tas doit être récupérée ou bien explicitement par le programme qui s'exécute (appels explicites à des fonctions de libération nommées `free` ou `delete`), ou alors automatiquement par un élément de la bibliothèque d'exécution qui est capable de détecter la mémoire qui est devenue inutile en n'étant plus référencée, dans le but de la recycler. Dans ce cas, l'appel à ce mécanisme de recyclage (nommé GC pour *garbage collection*, que l'on peut traduire ici en *glanage de cellules*) est typiquement exécuté automatiquement lorsqu'une allocation échoue par manque d'espace disponible.

Le recyclage de la mémoire devenue inutile peut donc être explicite comme en C ou C++, où des appels explicites à `free` ou `delete` vont procéder à ce recyclage, invalidant du même coup les éventuels pointeurs référençant la zone mémoire en question. Le recyclage peut aussi être automatique, comme c'est le cas dans les langages « modernes », comme OCaml, Java ou C#. Dans ce cas, la détection des espaces mémoire devenus inutiles garantit que plus aucun pointeur ne référence la mémoire en question, et les erreurs – classiques – d'utilisation de pointeurs devenus invalides ne peuvent survenir dans ce cas.

Ce chapitre est consacré à la gestion automatique de la mémoire et décrit les quelques techniques majeures procédant à cette récupération mémoire. Nous ne nous intéressons qu'aux programmes n'utilisant qu'une mémoire locale non partagée, et non pas à des programmes utilisant des mémoires distantes

et/ou partagées, qui nécessitent des techniques s'appuyant sur celles qui sont présentées ici pour la gestion locale, mais les étendent pour gérer des notions supplémentaires comme les pointeurs distants, par exemple.

9.1 La mémoire dynamiquement allouée

Les *durées de vie* des objets manipulés par un programme diffèrent en fonction de la nature de ces objets. Tout d'abord, certains objets n'ont pas besoin d'allocation de mémoire pour être représentés. Par exemple, les entiers natifs (dits aussi « entiers machine »), et les types de données dérivés (les caractères, booléens, *etc.*) n'ont pas besoin de représentation spécifique, puisque la taille de leur représentation est inférieure ou égale à la taille d'une adresse (32 bits, 64 bits) native du processeur considéré. Par contre, les autres objets (chaînes de caractère, tableaux, enregistrements, . . .) peuvent nécessiter un espace mémoire dédié (une zone de la mémoire comprise entre deux adresses). Une variable nécessite elle aussi un espace mémoire suffisamment grand pour pouvoir y écrire chacune des valeurs que prendra cette variable au cours de l'exécution du programme.

Lorsqu'on sait prédire statiquement la durée de vie de ces objets nécessitant une allocation mémoire, alors on génère, en compilant le programme, du code qui va allouer et recycler la mémoire correspondante. Par exemple, la fonction C suivante :

```
int fact(int n) {
  int res = 1;
  while (n > 1) {
    res = res * n;
    n = n-1;
  }
  return res;
}
```

devra disposer d'espace mémoire pour allouer les variables locales `n`, `res`. Or on sait, dans ce cas, calculer les durées de vie de ces variables : elles « naissent » à chaque exécution de la fonction `fact`, et disparaissent après que la fonction `fact` ait exécuté sa dernière instruction. De fait, le compilateur C va générer du code qui va stocker ces deux variables dans la pile d'exécution lors de l'appel à `fact`, et qui va libérer l'espace de pile correspondant au retour de la fonction. Puisqu'on sait calculer à la compilation la durée de vie de `n` et `res`, on dit que leur durée de vie est *statique*.

Considérons maintenant un autre exemple en C :

```
struct enregistrement {
   ...
};

struct enregistrement *f() {
  struct enregistrement *ptr;
  ptr = (struct enregistrement *) malloc(sizeof(struct enregistrement));
   ...;
  return ptr;
}
```

qu'on pourrait écrire en C++ comme :

```
struct enregistrement {
   ...
};
```

```
enregistrement *f() {
  enregistrement *ptr;
  ptr = new enregistrement;
   ...;
  return ptr;
}
```

On sait ici que l'allocation de la structure a lieu dès l'appel de la fonction, mais puisque le pointeur `ptr`, qui pointe vers cette structure [1], est produit en résultat par la fonction, sa durée de vie est potentiellement illimitée, et ne peut pas, dans le cas général, être prédite de façon sûre à la compilation. On sera sûr que la zone mémoire correspondante sera inutilisée lorsque que *plus aucun pointeur accessible ne la référencera*. La véracité de cette dernière propriété dépend dans le cas général de l'exécution du programme, et est donc indécidable. C'est pourquoi on dit que la durée de vie de cette structure est *dynamique*.

Dans un langage comme OCaml, de nombreux objets ont une durée de vie dynamique. N'importe quelle donnée structurée (enregistrement, tableau, chaîne, application de constructeur de données non constant, listes, *etc.*) est allouée en mémoire et il est généralement impossible de prédire leur durée de vie, même si celle-ci est généralement très courte. Notons que les variables locales de fonction peuvent elles aussi avoir une durée de vie dynamique. Par exemple, la fonction suivante :

```
let f n m =
  if n = m then 1
  else n+m−1
```

a deux variables locales dont la durée de vie serait facilement prévisible si `f` était toujours appelée avec deux arguments. Pour `f` e_1 e_2, on évalue e_1 et e_2, puis on stocke leur valeur dans la pile, où le code de `f` irait les chercher pour les utiliser, tout comme le ferait un code C.

Mais dans un langage fonctionnel comme OCaml, il est possible d'appeler `f` avec seulement le premier de ces arguments, comme dans `f` e_1. Le code de la fonction `f` va dans ce cas allouer une zone mémoire pour y stocker son code ainsi que la valeur de l'argument e_1 comme valeur de `n` (il s'agit d'une *fermeture*). L'ensemble représentera la fonction de type `int` → `int` qui attend un argument `m` et qui exécutera le code de `f` en prenant la valeur sauvegardée de e_1 pour valeur de `n`.

9.2 Allocation : la mémoire libre disponible

Avant de tenter de comprendre comment *récupérer* de la mémoire, il faut décrire comment l'allouer. Une allocation de la mémoire se présente comme une requête de la forme :

[je veux] recevoir un pointeur vers une zone mémoire de taille N

émise ou bien explicitement par le programme, ou alors par son *gestionnaire mémoire* dans le cas de langages comme OCaml, Java ou C#. Cette demande d'allocation est exécutée par une fonction de bibliothèque qui renvoie un pointeur valide si la mémoire est effectivement disponible. Si trop peu de mémoire est disponible, alors la fonction en charge de cette allocation peut soit :

- simplement échouer en renvoyant un pointeur invalide (`NULL` en C) et laisser au programmeur le soin de réagir à ce manque de mémoire,
- demander au système d'exploitation une extension de la mémoire du processus dans le but de satisfaire la demande,
- ou alors (dans le cas d'un gestionnaire automatique de la mémoire) tenter de récupérer une partie de la mémoire que le programme a déjà alloué et qui est devenue inaccessible, donc inutilisable et inutilisée en l'état.

1. C'est-à-dire que `ptr` est une variable contenant l'adresse du début de la zone mémoire représentant la structure.

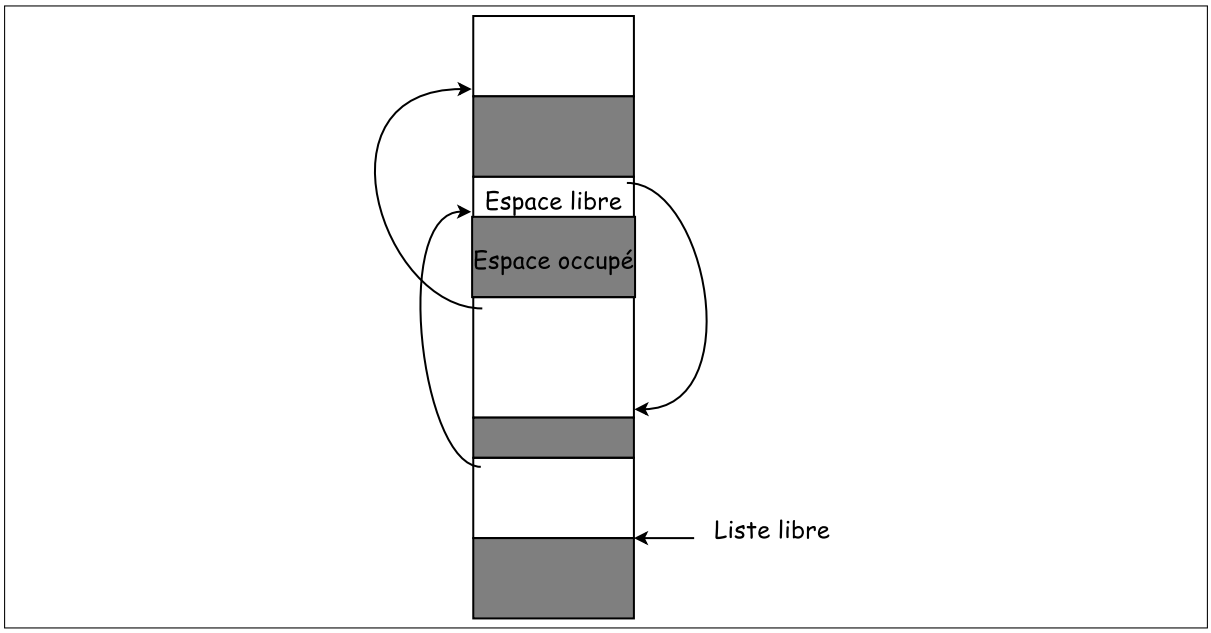


FIGURE 9.1 – La méthode de la liste libre

Représentation de la mémoire disponible

Il y a essentiellement deux méthodes pour représenter dans un processus la mémoire disponible.

La méthode dite « de la liste libre » consiste à chaîner entre eux les blocs de mémoire libre, de sorte que chaque bloc contienne un pointeur vers le bloc suivant. La liste ainsi créée est appelée la *liste libre* ou *free list* en anglais. Quand un bloc mémoire est nécessaire, il est choisi dans la liste libre et en est extrait. À l'inverse, lorsqu'un bloc est recyclé, il est inséré dans la liste libre. Le premier désavantage de cette représentation est le temps qui peut être nécessaire à trouver un bloc de la bonne taille (ou bien un bloc de taille supérieure, qu'il faudra « casser » pour en extraire une partie de la taille souhaitée) : en effet, il est possible que bien que la somme de l'espace mémoire disponible soit suffisant pour satisfaire une requête, chacun des blocs disponibles soit de taille insuffisante pour la satisfaire à lui seul. Il faut dans ce cas procéder au *compactage* de la mémoire libre.

Un autre problème, tout aussi important, est le fait qu'à force d'allouer et de recycler de la mémoire de la sorte, la mémoire occupée finit par être fragmentée, c'est-à-dire que des blocs pointent vers d'autres blocs qui sont physiquement éloignés d'eux, ce qui a un coût à l'exécution, puisque les accès successifs à ces blocs peuvent entraîner des relectures de segments de mémoire vive dans de la mémoire rapide (mémoire cache), voire des utilisations de la mémoire d'échange (le *swap*) qui entraînent des accès disques, bien plus coûteux que des accès mémoire. En bref, cette représentation de la mémoire libre impose un coût d'allocation qui peut être important et une perte de localité des données, en contrepartie d'une récupération rapide, puis qu'il s'agit d'une simple insertion du bloc à recycler en début de liste chaînée.

La méthode dite du pointeur vers le tas (*heap pointer*, en anglais) consiste, quant à elle, à maintenir la mémoire libre disponible dans un gros bloc de mémoire contiguë. Un pointeur référence le début de cette zone, et l'allocation d'un bloc consiste à produire ce pointeur en résultat, et à le déplacer à la fin de la zone qui vient d'être allouée, vers le début de ce qui reste de la zone libre. En apparence, le désavantage de cette méthode est que le coût de la solution naïve de recyclage de la mémoire semble important : copier les blocs mémoire utilisés vers le début de la zone mémoire, afin que les blocs libres

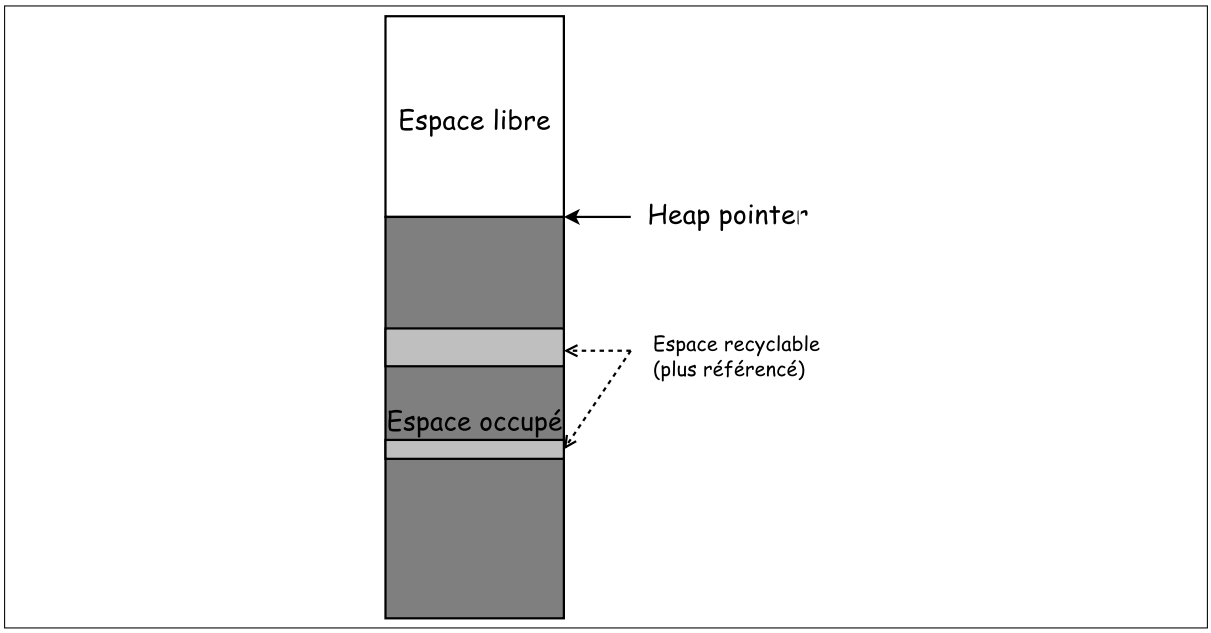


FIGURE 9.2 – La méthode du pointeur vers le tas

« remontent » en une zone contiguë [2]. Par contre, l'allocation d'un bloc mémoire est extrêmement rapide.

9.3 Récupération de la mémoire

La récupération et le recyclage de la mémoire se font « à la main » dans les langages classiques tels C, C++, Ada, *etc.* Les fonctions de récupération de la mémoire sont fournies aux programmeurs, ainsi qu'une bibliothèque d'attribution et de recyclage de la mémoire, et c'est le programmeur qui doit garantir qu'il libère de la mémoire à bon escient (toute tentative d'accès à une zone mémoire ainsi libérée provoque une erreur fatale qui peut ou non arrêter le programme, et lui faire produire des résultats en apparence aléatoire).

Les langages plus récents comme OCaml, Java, C#, les langages de script, *etc.* sont quant à eux dotés d'une gestion automatique de la mémoire qui gère à la fois *l'identification* de la mémoire inaccessible (et qui donc ne sera plus utilisée par le programme), ainsi que sa récupération. Un tel mécanisme de gestion automatique est appelé *garbage collector* en anglais et on le désigne souvent sous le nom de GC. Afin de donner (du moins . . . de tenter pour ne pas dire faire semblant) à cet acronyme un sens en bon Français, on lui a attribué le sens de *Glaneur de Cellules*.

Nous divisons la (petite) famille des GC en deux groupes : les GC à balayage et copiants d'une part, qui effectuent un parcours de la mémoire lors des récupérations, et les GC à compteurs de référence d'autre part, qui fonctionnent de façon beaucoup plus incrémentale, puisqu'ils sont capables de libérer très rapidement un bloc de mémoire devenu inutilisé.

9.3.1 Les GC à balayage et les GC copiants

Lorsque le tas est plein et qu'il n'y a donc plus de mémoire disponible, les GC de ce groupe vont parcourir la mémoire (selon des algorithmes différents lorsqu'il s'agit de GC à balayage ou de GC copiant) afin de déterminer l'ensemble des objets (blocs mémoire) effectivement utilisés, qu'on appelle aussi les objets *vivants*, par opposition aux objets devenus inutiles, dont on dira qu'ils sont

2. Regardez Windows compacter ses données sur le disque afin de « défragmenter », par exemple.

morts. Identifier les objets vivants détermine aussi les objets morts, puisque ces derniers forment le complémentaire des premiers dans le tas. Il est important de noter que pour parcourir (de façon exacte) le graphe mémoire accessible, il est indispensable de savoir distinguer les pointeurs des autres données (les entiers, typiquement), et de connaître la taille des blocs alloués [3]. Il faut donc que la mémoire soit organisée pour être balayée.

À l'exécution, les objets alloués dans le tas peuvent être référencés de diverses façons : ils peuvent être les valeurs de variables locales ou globales, ou bien être référencés par d'autres blocs mémoire (enregistrements ou tableaux, par exemple). On dira qu'un objet est *vivant* si l'une des deux conditions suivantes sont satisfaites :

1. son adresse est la valeur d'une variable (un pointeur) qui est ou bien globale, ou alors stockée de façon temporaire dans la pile ou dans un registre ;
2. il y a un objet alloué dans le tas (structure, tableau, environnement de valeur fonctionnelle, *etc.*) qui est lui-même vivant et qui contient un pointeur vers cet objet.

Clairement, tous les objets vivants dans le tas peuvent être identifiés par un parcours du graphe mémoire, dont les points de départ sont les variables locales stockées dans la pile et les registres et les variables globales. On appelle ces points de départ des *racines* et tout objet qui n'est pas accessible depuis ces racines sera considéré comme *mort* et pourra être recyclé.

Les GC à balayage et les GC copiants effectuent tous un parcours de graphe visant à identifier les objets vivants, mais diffèrent par les actions qu'ils exécutent après le parcours.

Les collecteurs à balayage

Les collecteurs à balayage disposent d'un bit dit *marque* équipant chacun des blocs de la mémoire. Lorsque la mémoire est pleine, l'exécution est stoppée, et le GC prend le contrôle. Il commence par parcourir entièrement le graphe de la mémoire accessible depuis les racines, marquant chacun des blocs mémoire rencontrés. Cette phase est appelée *phase de marquage*.

Lorsque le marquage est terminé, tous les objets vivants auront été marqués. Les blocs mémoire qui n'ont pas été marqués durant cette phase peuvent être libérés : ils sont alors chaînés dans la *liste libre*. Cette phase de glanage des blocs non marqués est appelée *phase de balayage*. Ces GC sont appelés GC à marquage/balayage (*mark and sweep*, en anglais).

En pseudo-code, les fonctions d'un GC à marquage/balayage pourraient s'écrire de la manière suivante :

```
collecte_par_marquage_balayage () {
  pour chaque racine r :
       marquer (r) ;
  balayer () ;
}

marquer (p) {
  si le bloc pointé par p n'est pas marqué, alors :
    on marque le bloc pointé par p ;
    pour chaque champ r de ce bloc :
      si r est un pointeur alors :
           marquer (r) ;
}

balayer () {
  pour chaque bloc mémoire b du tas :
    si b est non marqué, alors :
         insérer (b, freelist) ;
```

3. La taille des blocs peut par exemple être inscrite dans un champ dédié de chaque bloc.

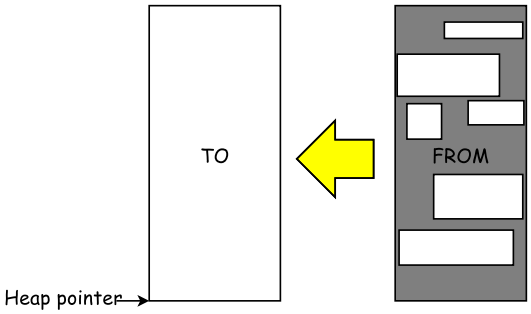
}

Si L est la taille totale occupée par les objets vivants, et M la taille du tas, alors le coût d'un algorithme de GC à marquage/balayage est $\Theta(L) + \Theta(M)$, que l'on peut ramener à $\Theta(M)$. On peut dont retenir que ces GC ont un coût proportionnel à la taille du tas, quelque soit le taux d'occupation de celui-ci.

Le désavantage des algorithmes à marquage/balayage est, comme nous l'avons déjà dit, qu'ils ne sont pas compactants, forçant l'usage d'une liste de blocs libres (par opposition à un bloc libre unique). Par ailleurs, leur complexité fait qu'ils induisent une pause à l'exécution proportionnelle à la taille de la mémoire. Le GC à marquage/balayage se révèle donc mal adapté à des environnements contraints tels les environnements temps-réel où les réactions des programmes doivent être quasi-immédiates, ou, au moins, bornées dans le temps. Cette méthode de récupération de la mémoire est appelée *stop-and-collect* en anglais.

Les collecteurs à recopie

Ces collecteurs fonctionnent avec un espace mémoire divisé en deux tas, dont seulement un est utilisé à un instant donné. Celui qui est utilisé est appelé *tas origine* (*from space*) et l'autre est appelé *tas destination* (*to space*). Les objets sont donc alloués dans le tas origine, qui est un gros bloc mémoire dont une partie est libre. Lorsque le tas origine est plein, la procédure de récupération de la mémoire commence : le GC parcourt le graphe des blocs vivants à partir des racines. Au fur et à mesure qu'il effectue son parcours, le GC recopie dans le tas destination les blocs qu'il rencontre. Lorsque le parcours est complet, tous les blocs vivants ont été recopiés dans le tas destination. À ce moment, on échange les tas origine et destination, et l'exécution reprend.



Une bonne propriété de la récupération par recopie est la suivante : les GC copiants sont des GC compactants : chaque objet vivant est recopié dans le premier espace libre disponible. Lorsque la recopie est terminée, l'espace libre se présente sous la forme d'un bloc unique, donc non fragmenté.

Lorsque les objets sont recopiés dans le tas destination, les pointeurs vers ces objets doivent être mis à jour de sorte qu'ils pointent vers la nouvelle adresse : cela est effectué en laissant une « adresse distante » à la place de l'ancien objet une fois qu'il a été déplacé. Ainsi, lorsqu'un pointeur vers l'ancien objet est rencontré plus tard, le GC se rendra compte qu'il s'agit d'une adresse distante [4] et mettra à jour le pointeur avec cette nouvelle adresse. En effet, un pointeur ordinaire du tas d'origine ne peut pointer vers le tas destination.

En pseudo-code, l'algorithme de GC copiant peut s'écrire :

```
collecte_par_recopie () {
  pour chaque pointeur racine p :
    p := transférer (p) ;   (* Récupérer la nouvelle racine. *)
}

transférer (p) {
```

4. Rappelons que le GC sait distinguer les pointeurs des autres données : il sait donc repérer une telle « adresse distante » sans difficulté.

```
    si p pointe vers un bloc contenant une adresse distante q, alors :
        renvoyer q ;
    sinon :
        soit q = copie (p, TO) ;
        écrire q comme adresse distante
                  à la place du bloc pointé par p ;
        (* Parcours des champs après copie, donc depuis TO. *)
        soit flds les champs de q contenant des pointeurs ;
        pour chaque champ qf de flds,
            qf := transférer (qf) ;
        renvoyer q ;
}
```

Si L est la taille de la mémoire occupée par les objets vivants et M la taille de chaque tas, alors le coût d'une récupération par recopie est en $\Theta(L)$. À l'inverse du GC par marquage/balayage, le coût est proportionnel au volume de mémoire utilisée, et non pas à la taille de la mémoire totale.

Puisque la taille de chaque tas est la moitié de la taille totale de la mémoire, les GC par recopie seront deux fois plus fréquents que les GC par marquage/balayage. Cela dit, avec un ordinateur disposant de mémoire virtuelle, on peut aussi doubler la taille de la mémoire, le tas inutilisé restant sur le disque tant que le GC n'a pas lieu.

Les GC par recopie, même s'ils sont plus efficaces que les GC à marquage/balayage, sont tout de même mal adaptés aux contraintes temps-réel.

Les collecteurs à générations

Ce type de GC est une variante de collecteur copiant et s'appui sur un constat pratique dans certains langages :

- les blocs jeunes (récemment créés) meurent vite,
- les blocs qui survivent vivent vieux,
- les jeunes pointent vers les vieux, le plus souvent.

En effet, de nombreux blocs sont éphémères, provenant de calculs intermédiaires, de données temporaires. Par exemple, lors de l'évaluation de l'expression $(3+5)*2$, il va falloir temporairement mémoriser le résultat de 3 + 5 qui va être ensuite immédiatement utilisé pour calculer le résultat de sa multuplication par 2. Cette valeur intermédiaire 8 ne vit donc que très peu de temps. Notons qu'un compilateur un tant soit peu intelligent aura effectué ce calcul à la compilation et que les calculs sur des valeurs entières ne nécessitent pas forcément d'allocation de blocs. Néanmoins, le principe reste valable pour des expressions ne pouvant être calculées à la compilation ou des calculs plus compliqués.

Les blocs restant utiles longtemps sont souvent des structures de données importantes du programme puisqu'elles servent tout au long de son exécution. Bien entendu, parmi ces données se trouvent les variables globales puisqu'elles peuvent être accédées à tout moment par n'importe quelle fonction.

Pour terminer, généralement un bloc récent utilise (pointe vers) des blocs anciens puisque sa valeur va avoir été calculée à partir des données déjà en mémoire au moment de sa fabrication. Nous reviendrons sur ce point un peu plus loin.

Un GC à génération va exploiter le constat précédent en tentant de ne parcourir que les blocs qui ont la plus grande probabilité d'être morts : les blocs récents. Il convient donc que les blocs soient regroupés par âge similaire. Le GC dispose donc de plusieurs tas (au minimum 2), nommés *générations*, dans lesquels les blocs seront disposés par « âge similaire ». L'allocation de blocs se fait dans la « jeune génération » qui va donc être celle qui contient les blocs ayant la plus forte probabilité d'être morts en cas de besoin de recyclage. Lorsque le GC se déclenche, il parcourt la jeune génération. Cela réduit le coût du GC puisqu'il ne parcourt plus l'ensemble des blocs vivants, mais seulement les blocs vivants présents dans la jeune génération. À l'issue de la détermination des blocs vivants, ces derniers sont alors copiés dans la génération plus vieille : ils « vieillissent ». Lorsque la génération plus vieille est pleine, elle doit être recyclée à son tour, provoquant un déplacement de ses blocs vivants dans la génération

encore plus vieille. Ainsi, le GC travaille incrémentalement sur les générations, ne recyclant que celles se trouvant pleines à un instant donné.

Nous avons dit que les blocs jeunes pointaient vers les vieux, « le plus souvent ». Considérons un langage purement fonctionnel, donc sans effets de bord (pas de références en particulier). Dans ce cas, un calcul ne peut utiliser que des valeurs déjà calculées, donc déjà présentes en mémoire, donc dans une génération supérieure ou égale à la génération d'allocation courante. Pour évaluer `n * fact (n - 1)`, il faut avoir calculé la valeur de `n`, celle de `n - 1` et celle de `fact (n - 1)`. Ainsi, le bloc où stocker le résultat de la multiplication ne peut qu'utiliser des blocs plus vieux que lui : il sera créé après eux pour justement mémoriser le résultat. De ce fait, dans un langage purement fonctionnel, il est impossible d'avoir des blocs d'une génération pointant vers des blocs d'une génération plus jeune : c'est toujours dans l'autre sens.

Si maintenant nous avons des références, alors il est possible de mettre à jour un bloc ancien avec une nouvelle valeur. Le bloc ancien peut être dans une génération et la nouvelle valeur dans une génération plus jeune. Considérons le programme suivant dans un langage imaginaire.

```
...
var t[20]
...
i := 0
while i < 20 do
  t[i] = foobar (i)
  i := i + 1
done
```

Le tableau `t` est créé avant les blocs qui seront créés dans le corps de la boucle. Ainsi, lorsque l'on met à jours `t[i]` les blocs de `t` vont possiblement avoir migré vers une génération plus ancienne depuis la création de `t`. Les blocs alloués pour retourner le résultat de la fonction `foobar`, plus récents, seront peut-être dans une génération plus jeune et seront mémorisés dans des vieux blocs : des vieux pointeront sur des jeunes. Nous voyons que pour obtenir cette situation, il a fallu mettre à jour des anciens blocs. Ce n'est donc possible qu'avec un moyen de « revenir sur une valeur déjà calculée », de la modifier, donc en utilisant des affectations sur des références (puisque elles seules sont mutables).

Si l'on ne regarde que les racines pointant vers la jeune génération, nous ne verrons pas qu'un bloc d'une ancienne génération pointe vers un bloc jeune. Si ce bloc jeune n'est pas pointé par les racines ou un autre bloc jeune accessible depuis ces dernières, alors il pourra être (à tort) libéré. Ainsi, le pointeur dans le bloc vieux deviendra invalide.

Pour éviter une telle libération, il faut modifier l'affectation afin qu'elle déclare si un bloc vieux pointe vers un bloc jeune. Si tel est le cas, le bloc vieux est enregistré comme *référence entrante* (nouvelle racine) de la jeune génération. Ainsi, chaque génération se doit de disposer de sa propre liste de références entrantes.

Les GC à génération bénéficient des mêmes avantages que les GC copiants. De plus, limitant le parcours des blocs vivants à un nombre minimal de générations, ils sont plus efficaces en terme de temps d'éxécution. Leur désavantage est de requérir deux fois plus de mémoire, la moitié étant inutilisée la majorité du temps.

9.3.2 Les GC à compteur de références

Les GC à compteurs de références sont, quant à eux, fondés sur une idée complètement différente, et assez simple : chaque bloc contient un champ pouvant contenir un entier appelé *compteur de références*. À l'exécution, le compteur de références d'un bloc contient le nombre d'objets (variables, autres blocs)

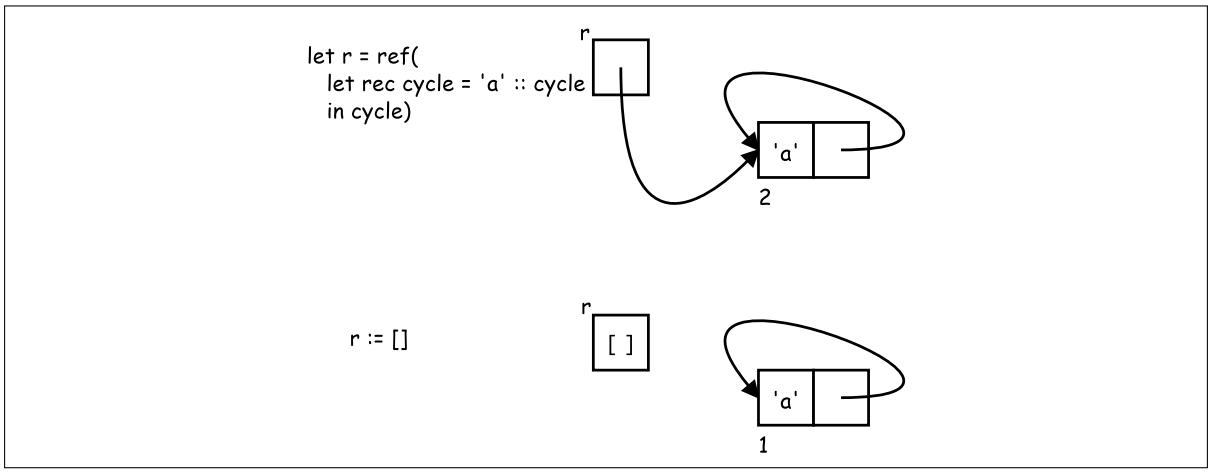


FIGURE 9.3 – Cycles et compteurs de références

pointant *directement* vers ce bloc. Lorsqu'un bloc est alloué et qu'un pointeur vers ce bloc est créé, le compteur de références de ce bloc est initialisé à 1. Lorsqu'un nouveau pointeur vers ce bloc est créé, le compteur de références du bloc est incrémenté. À l'inverse, lorsqu'un pointeur vers un bloc est détruit, le compteur de références de ce bloc est décrémenté.

Par exemple, si une variable `x` contient une référence vers un bloc b_1, et que l'on exécute l'affectation `x` := b_2 le compteur de références de b_1 est décrémenté et celui de b_2 est incrémenté.

Lorsque le compteur de références d'un bloc devient nul, le bloc est libéré, et peut être recyclé, en ayant au préalable détruit les pointeurs contenus dans le bloc, décrémentant ainsi les compteurs de références des blocs correspondants. Le recyclage de l'objet consiste ensuite à le chaîner dans la liste libre.

Il est difficile de comparer le coût du comptage de références avec les autres techniques de GC. L'avantage principal de cette technique est que le coût du GC est réparti sur l'exécution du programme : le GC ne provoque donc presque pas de pause, au contraire des autres techniques. En fait, les pauses sont possibles dans des circonstances très particulières : lorsque la libération d'un pointeur provoque à son tour la libération de toute une hiérarchie, ce qui peut provoquer un parcours de graphe mémoire similaire à ce que font les autres techniques. Ces circonstances sont toutefois relativement rares.

Cette technique de récupération de la mémoire présente tout de même quelques inconvénients. Le premier est que les structures cycliques formées dans la mémoire peuvent ne pas être récupérées. Prenons un exemple de structure cyclique que l'on crée, et dont on supprime un pointeur. On s'aperçoit alors qu'il n'y a aucun moyen de faire tomber à zéro les compteurs de références des blocs composant le cycle, sauf à parcourir le cycle, ce qui n'est pas dans la philosophie du comptage de références. C'est pourquoi les GC à compteurs de références sont souvent accompagnés d'un GC à marquage/balayage capable de récupérer les cycles qui sinon resteraient alloués tout en étant devenus inutiles.

Il se trouve aussi que le comptage de références est assez difficile à mettre en œuvre, car il ne faut oublier aucun pointeur créé ou copié lorsqu'on crée des variables temporaires, lors des passages d'arguments aux fonctions/procédures, *etc*. De plus, les compteurs eux-même sont consommateurs de mémoire, tout en étant à même de déborder dans certaines circonstances.

Chapitre 10

Le lambda-calcul

Le λ-calcul est un système formel basé sur la notion de fonction. Créé par le logicien Alonzo Church dans les années 1930, le λ-calcul a servi à Church à définir la notion de fonction calculable en 1936. En parallèle, Alan Turing créait une classe de « machines » que l'on a appelées par la suite *machines de Turing* et a, lui aussi, défini une notion de fonction calculable associée à ces machines. Turing a montré en 1937 que ces deux classes de fonctions étaient les mêmes, et donc que ces deux systèmes formels avaient exactement la même puissance d'expression.

Conçu bien avant l'ordinateur et donc le premier langage de programmation, le λ-calcul a eu une influence forte sur la majorité des langages que nous connaissons aujourd'hui. Comme déjà remarqué dans les premiers chapitres de ce document, les premiers langages de programmation ont surtout cherché à rapprocher la programmation de l'écriture de formules arithmétiques. Les notions de fonction ou de procédure sont ensuite venues aider à la structuration des programmes, prenant la place des sous-programmes ou *subroutines*, qui n'étaient qu'un détournement temporaire du flot de contrôle. Fonctions et procédures ont pris une grande importance suite à la création du langage Algol, dont tous les langages structurés que nous connaissons aujourd'hui sont les descendants plus ou moins directs.

Bien sûr, les langages fonctionnels de programmation sont ceux qui héritent le plus directement du λ-calcul : ils autorisent la création de fonctions anonymes, les fonctions d'ordre supérieur (recevant d'autres fonctions en argument et/ou produisant des fonctions en résultat). Parmi ceux-ci, outre le langage Scheme, lui-même héritier du langage Lisp dont la création a été contemporaine de celle d'Algol (1958), on notera bien sûr les langages de la famille ML, dont OCaml fait partie, ainsi que le langage Haskell, où les arguments de fonctions sont traités différemment de ce que nous connaissons en général.

La sémantique dénotationnelle, que nous avons vue au chapitre précédent, utilise des fonctions sémantiques qui produisent elles-même d'autres fonctions (mathématiques). Nous avons vu au chapitre précédent que l'utilisation de langages fonctionnels pour implémenter les sémantiques dénotationnelles était assez naturelle : il est donc tout aussi naturel d'utiliser la syntaxe du λ-calcul pour décrire les fonctions qui sont produites par de telles sémantiques. Et c'est généralement le cas.

Nous verrons dans ce chapitre comment sont construits les termes du λ-calcul, et quelles en sont les propriétés principales.

10.1 Termes

Les termes du λ-calcul, aussi appelés λ-termes, sont donnés par la grammaire suivante :

$$\begin{array}{rcll} M,\, e & ::= & x & \text{identificateurs} \\ & | & e\, e & \text{applications} \\ & | & \lambda x.e & \text{abstractions} \end{array}$$

On appelle souvent *variables* les identificateurs, mais il faut comprendre ce terme au sens de *variable muette* en mathématiques, et non pas comme une variable ou référence dans un langage comme C. Les abstractions ($\lambda x.e$) sont à rapprocher des fonctions anonymes d'OCaml (**fun** x $\rightarrow$ e), et les applications ($e_1 e_2$) des applications de fonctions en OCaml.

Lorsqu'on écrit des termes complexes, on utilise des parenthèses pour en grouper les différentes composantes. Du point de vue de la syntaxe concrète (la notation), on a les équivalences suivantes :

- l'application « associe à gauche » :

$$e_1 e_2 e_3 \equiv (e_1 e_2) e_3$$

- l'abstraction « porte » aussi loin que possible :

$$\lambda x.\lambda y.\lambda z.xzyz \equiv \lambda x.(\lambda y.(\lambda z.xzyz))$$

On reconnaît ici les règles syntaxiques adoptées par OCaml, et on peut voir ci-dessus e_1 comme une fonction à deux arguments ou, de façon équivalente, comme une fonction qui, lorsqu'elle reçoit son argument e_2 produit une nouvelle fonction qui attend un argument e_3.

10.1.1 Constantes prédéfinies

On équipe quelquefois le λ-calcul avec des constantes et fonctions primitives pour le rapprocher un peu plus des langages de programmation : le doter ainsi de tels objets primitifs permet par exemple de parler des entiers ou les opérations arithmétiques sans avoir à les encoder dans le λ-calcul (ce qui est possible, comme nous le verrons à la fin de ce chapitre).

10.1.2 Variables libres, variables liées

Dans une abstraction ($\lambda x.e$), le terme e est appelé le *corps* de l'abstraction lorsqu'on y pense comme à un sous-terme. Lorsqu'on veut le relier à la variable x, on dit que e est la *portée* de x. Il s'agit là de la même notion de portée que celle que l'on rencontre dans les langages de programmation. La portée de x est le terme (c'est-à-dire de la plage dans le programme) à l'intérieur duquel les occurrences de la variable x font référence à cette même variable x introduite par le « λx ».

Les occurrences d'une variable x dans la portée d'un λx sont dites *liées* par ce λx. Dans une abstraction ($\lambda x.e_1$), il faut faire attention aux éventuels sous-termes de e_1 qui seraient de la forme ($\lambda x.e_2$) – liant une variable de même nom x que la précédente –, excluant e_2 de la portée du λx le plus externe. On dit que le λx le plus interne met le x le plus externe *hors de portée* dans le sous-terme correspondant. C'est exactement ce qui se passe en programmation lorsqu'une variable locale masque – localement – une variable existante : l'introduction de la variable locale met l'autre hors de portée.

Les occurrences de variables qui ne sont pas liées dans un terme sont dites *libres*. Une variable est dite *libre* dans un terme, si elle a au moins une occurrence libre dans ce terme.

L'ensemble des variables libres d'un terme peut se calculer de la façon suivante :

$$\begin{aligned}
&\mathsf{freeVars}(x) = \{x\} \\
&\mathsf{freeVars}(e_1 e_2) = \mathsf{freeVars}(e_1) \cup \mathsf{freeVars}(e_2) \\
&\mathsf{freeVars}(\lambda x.e) = \mathsf{freeVars}(e) - \{x\}
\end{aligned}$$

La notion de variable libre ou liée est très importante en λ-calcul, car c'est cette notion qui détermine le rôle de la variable dans la majorité des traitements que l'on fait subir aux λ-termes. Les variables et paramètres formels des langages de programmation sont eux aussi touchés directement par ces concepts, qui formalisent l'intuition qu'on a à leur sujet.

10.2 Substitutions

Prenons une fonction OCaml dont le paramètre formel est x. Lorsque cette fonction est appliquée, la valeur calculée par cette application sera celle du corps de la fonction où les occurrences de x auront

été remplacées par l'argument. *Toutes* les occurrences ? Non [1] ! Seulement les occurrences de x qui sont *libres* dans le corps de la fonction.

L'opération de substitution dans le λ-calcul formalise très précisément cette opération de « remplacer les occurrences de x par l'argument », dans un contexte un peu plus général que celui d'un langage de programmation comme OCaml.

La substitution d'un terme M à une variable x dans le terme e, notée $[M/x]e$, est définie comme le terme résultant du remplacement de toutes les occurrences libres de x par M dans e :

$$\begin{array}{lcl}
[M/x]x & = & M \\
[M/x]y & = & y, \text{ pour } y \neq x \\
[M/x](e_1 e_2) & = & ([M/x]e_1)([M/x]e_2) \\
[M/x](\lambda x.e) & = & (\lambda x.e) \\
[M/x](\lambda y.e) & = & \lambda y.[M/x]e \text{ pour } x \neq y \text{ et } y \notin \mathsf{freeVars}(M) \\
[M/x](\lambda y.e) & = & \lambda z.[M/x]([z/y]e) \\
 & & \text{pour } z \notin \mathsf{freeVars}(e) \cup \mathsf{freeVars}(M) \\
 & & \text{et } z \neq x \text{ et } y \neq x
\end{array}$$

10.3 Règles de réduction

Les règles d'équivalence que nous donnons ici sont des équations entre termes. Nous les notons par une double flèche $\longleftrightarrow$, car nous en considérerons certaines par la suite comme des règles de réduction que nous noterons $\rightarrow$.

10.3.1 α-équivalence

Cette règle dit que le nom des variables liées (c'est-à-dire des variables muettes) importe peu : on peut les changer à volonté, à condition toutefois de le faire de façon cohérente, et de ne pas « capturer » des occurrences d'autres variables au passage. L'équivalence s'écrit :

$$(\alpha) \quad \lambda x.e \longleftrightarrow \lambda y.[y/x]e, \text{ pour } y \text{ n'apparaissant ni libre, ni liée dans } e$$

L'α-équivalence est une relation d'équivalence. On peut l'utiliser sur des sous-termes de termes plus grands : ainsi utilisée, elle devient une *congruence*. En fait, lorsqu'on considère des termes du λ-calcul, c'est généralement modulo α-conversion : il est en effet très naturel d'identifier par exemple $(\lambda x.x)$ et $(\lambda y.y)$.

10.3.2 β-équivalence

La seconde équivalence, nommée β est la plus importante : c'est elle qui explique comment l'application d'une fonction à un argument peut se réduire en un résultat :

$$(\beta) \quad (\lambda x.e)M \longleftrightarrow [M/x]e$$

Cette règle nous dit que pour calculer l'application d'une fonction de x à un argument M, il suffit de calculer le corps e de la fonction dans lequel on a substitué l'argument N au paramètre formel x. Lue dans l'autre sens, elle indique que l'on peut extraire une ou plusieurs occurrences d'un même sous-terme M d'un terme, les remplacer par une variable x bien choisie, *abstraire* x et produire l'application de la fonction obtenue au terme M que l'on a extrait. Cette opération est à rapprocher de la construction **let** ... **in** ... en OCaml : la règle β dit que (**fun** $x \rightarrow e_2$) e_1 est équivalent à **let** $x = e_1$ **in** e_2.

En fait, cette règle est majoritairement utilisée comme règle de calcul, de la gauche vers la droite.

1. Toute similitude éventuelle entre cette exclamation et une autre, plus célèbre et liée à un village gaulois, n'est que pure coïncidence.

10.3.3 η-équivalence

Cette troisième équivalence nous rappelle que la fonction f et celle qui à x associe $f(x)$ (que nous notions au collège $x \mapsto f(x)$), sont équivalentes :

$$(\eta) \quad \lambda x.ex \longleftrightarrow e, \text{ si } x \notin \mathsf{freeVars}(e)$$

Elle est plus rarement utilisée comme règle de calcul. Cela dit, on l'utilise souvent en OCaml par exemple pour retarder (ou exécuter à chaque application) le calcul de e en écrivant (**fun** () $\rightarrow e$ ()).

10.3.4 Notations

Nous utiliserons généralement ces équivalences ou réductions dans des sous-termes de termes et pas seulement en tête des termes que nous considérons. En particulier, la règle β est *a priori* utilisable dans n'importe quel contexte.

Lorsque nous parlerons de réductions ci-dessus (et plus particulièrement de β-réduction), on notera $M \xrightarrow{*} N$ lorsqu'il existe une chaîne éventuellement vide de réductions reliant M à N. Lorsqu'il s'agit d'une séquence de conversions $\longleftrightarrow$, on écrira $M \xleftrightarrow{*} N$

On écrira $M =_\alpha N$ pour signifier que M est α-équivalent à N, c'est-à-dire $M \xleftrightarrow{*}_\alpha N$.

Un terme M qui ne peut être réduit est dit *irréductible*. On dit aussi qu'il est *normalisé* ou encore *en forme normale*.

10.3.5 Propriétés

La β-réduction, vue comme règle de calcul, est utilisable sur des sous-termes arbitraires d'un terme à réduire. On espère ainsi réaliser un calcul complet en itérant le processus consistant à identifier un sous-terme réductible par β (on appelle un tel sous-terme un *radical* [2]) et à réduire ce radical dans son contexte, obtenant ainsi un nouveau terme que l'on cherchera à réduire à son tour, *etc.*

Une telle procédure soulève un certain nombre de questions parmi lesquelles on peut citer :

- la terminaison : cette procédure termine-t-elle toujours ?
- la *confluence* : le choix du radical à réduire à chaque étape influe-t-il sur le résultat ?
- quelles sont les stratégies de choix du prochain radical ? Qu'est-ce qui les différencie ?

Terminaison

La réponse à la question de la terminaison des calculs par β-réduction est intuitivement claire : si le λ-calcul est, comme on l'a dit, équivalent aux langages de programmation comme C ou OCaml, il doit permettre d'encoder un programme qui boucle. On admet aisément qu'on n'est que peu avancé avec une telle réponse : nous serions plus à l'aise avec un exemple concret de λ-terme dont le calcul par β-réduction ne peut pas terminer. En posant $\Delta = \lambda x.xx$, et $\Omega = \Delta\Delta$, si on tente de réduire Ω par la règle β, on voit aisément qu'on peut appliquer la réduction β à l'infini, sans jamais progresser d'un pouce :

$$(\lambda x.xx)(\lambda x.xx) \rightarrow_\alpha (\lambda x.xx)(\lambda x.xx) \rightarrow_\alpha (\lambda x.xx)(\lambda x.xx) \rightarrow_\alpha \dots$$

On peut catégoriser un peu plus précisément les termes dont le calcul termine, en distinguant ceux dont le calcul *peut* terminer, et ceux dont le calcul termine *toujours*.

Définition On dit qu'un terme e est *fortement normalisant* si et seulement si toutes les réductions partant de e sont finies.

Définition On dit que e est *faiblement normalisant* si et seulement si il existe une réduction finie menant à un terme irréductible.

2. *Redex* en anglais.

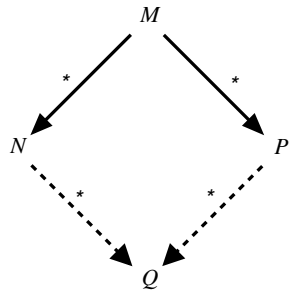
Le terme Ω que nous avons vu ci-dessus n'entre bien sûr dans aucune de ces deux catégories. Par contre, le terme $(\lambda x.(\lambda y.y))\Omega$ est faiblement normalisant, mais pas fortement.

On voit ici que le choix du radical influe sur le calcul, au moins sur sa terminaison.

Confluence

L'un des théorèmes essentiels du λ-calcul est le théorème qui indique que la β-réduction est *confluente*, c'est-à-dire que toutes chaînes de réduction partant d'un terme peuvent atteindre les mêmes termes. La conséquence de cette propriété est que le λ-calcul considéré comme langage de programmation dans lequel l'ordre d'évaluation n'est pas spécifié, est un langage *déterministe*.

Théorème Pour tout terme M, N et P, si $M \xrightarrow{*}_\beta N$ et $M \xrightarrow{*}_\beta P$, alors il existe un terme Q tel que $N \xrightarrow{*}_\beta Q$ et $P \xrightarrow{*}_\beta Q$. (On considère ici des termes modulo α-conversion.)



Corollaire Les formes normales sont uniques : étant donné un terme M, si deux réductions de M mènent à deux termes irréductibles N_1 et N_2, alors $N_1 =_\alpha N_2$.

Nous laissons pour le moment de côté les questions relatives aux stratégies de choix du radical à réduire : nous y reviendrons à la fin du chapitre.

10.4 Le λ-calcul et les langages de programmation

Il est très facile d'étendre le λ-calcul en un langage de programmation plus ou moins réaliste en y ajoutant des constantes (entiers, booléens, caractères, *etc.*), des fonctions primitives ainsi que des structures de données. D'une certaine façon, on peut comprendre les langages fonctionnels comme construits de cette façon, tout en étant conscient que les « vrais » langages de programmation fixent l'ordre d'évaluation, ce que nous ne faisons pas ici. (Voir les stratégies de réduction du lambda-calcul à la fin de ce chapitre.)

Au lieu de procéder à une telle extension, qui ne présente que peu d'intérêt dans le cadre de ce cours, nous nous attachons maintenant à *encoder* certains des éléments ci-dessus au lieu de les sortir d'un chapeau en les prédéfinissant. De fait, les éléments d'encodage donnés ci-après rendent crédible la puissance d'expression théorique du λ-calcul, qui est capable, rappelons-le, d'encoder toutes les fonctions calculables, c'est-à-dire rien de moins que tout ce que peut calculer un ordinateur !

Opérations booléennes

$$\texttt{True} \equiv \lambda x.\lambda y.x$$
$$\texttt{False} \equiv \lambda x.\lambda y.y$$

L'expression conditionnelle peut être codée comme une fonction du λ-calcul. À noter qu'elle n'est pas (et ne peut pas être) une fonction en OCaml, par exemple.

$$\texttt{If}\ A\ B\ C \equiv A\ B\ C$$

On vérifie aisément que :

$$\texttt{If True}\ B\ C \quad \overset{*}{\rightarrow}_\beta \quad B$$

et que :

$$\texttt{If False}\ B\ C \quad \overset{*}{\rightarrow}_\beta \quad C$$

Données structurées

Listes :

$$\texttt{Pair}\ A\ B \equiv \lambda x.\texttt{If}\ x\ A\ B \equiv \lambda x.x\ A\ B$$
$$\texttt{First}\ A \equiv A\ \texttt{True}$$
$$\texttt{Rest}\ A \equiv A\ \texttt{False}$$

On vérifie aisément que :

$$\texttt{First}\ (\texttt{Pair}\ A\ B) \quad \overset{*}{\rightarrow} \quad A$$
$$\texttt{Rest}\ (\texttt{Pair}\ A\ B) \quad \overset{*}{\rightarrow} \quad B$$

La liste vide :

$$\texttt{Nil} \equiv \lambda x.\texttt{True}$$
$$\texttt{Null}\ A \equiv A\ (\lambda x.\lambda y.\texttt{False})$$

On vérifie que :

$$\texttt{Null Nil} \quad \overset{*}{\rightarrow} \quad \texttt{True}$$
$$\texttt{Null}\ (\texttt{Pair}\ A\ B) \quad \overset{*}{\rightarrow} \quad \texttt{False}$$

Arithmétique

Les nombres :

$$\texttt{Zero} \equiv \lambda f.\lambda x.x$$
$$\texttt{One} \equiv \lambda f.\lambda x.f\ x$$
$$\texttt{Two} \equiv \lambda f.\lambda x.f\ (f\ x)$$
$$\texttt{Three} \equiv \lambda f.\lambda x.f\ (f\ (f\ x))$$

Un nombre n est un itérateur fonctionnel prenant une fonction f, et produisant la composée n-ème de f avec elle-même.

$$\texttt{Succ} \equiv \lambda n.\lambda f.\lambda x.f\ ((n\ f)\ x)$$
$$\texttt{IsZero} \equiv \lambda n.n\ (\lambda x.\texttt{False})\ \texttt{True}$$
$$\texttt{Plus} \equiv \lambda m.\lambda n.(m\ \texttt{Succ})\ n$$
$$\texttt{Mult} \equiv \lambda m.\lambda n.\lambda f.m\ (n\ f)$$
$$\texttt{Power} \equiv \lambda m.\lambda n.(n\ (\texttt{Mult}\ m))\ (\texttt{Succ Zero})$$

Récursion et points fixes

Rappelons qu'un *point fixe* d'une fonction f est un élément a tel que $a = f(a)$. Dans le λ-calcul, un point fixe d'un terme f est un terme e tel que :

$$e \leftrightarrow f\ e$$

Un *opérateur de point fixe* est un terme Fix tel que :

$$\mathsf{Fix}\ M \leftrightarrow M\ (\mathsf{Fix}\ M)$$

Fix M est donc un point fixe de M.

Dans le λ-calcul, les opérateurs de point fixe sont définissables :

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

est l'un d'entre eux. On le montre aisément par la chaîne suivante de conversions :

$$\begin{array}{rcl} YM & \leftrightarrow & (\lambda x.M(xx))(\lambda x.M(xx)) \\ & \leftrightarrow & M((\lambda x.M(xx))(\lambda x.M(xx))) \\ & \leftrightarrow & M(YM) \end{array}$$

On peut utiliser cette propriété caractéristique des opérateurs de point fixe pour construire dans le λ-calcul l'équivalent des fonctions récursives : pour une fonction f définie récursivement par une équation $f = ... \ f \ ...$, on peut construire la *fonctionnelle* $(\lambda f \ . \ ... \ f \ ...)$, et prendre son point fixe en utilisant un opérateur de point fixe du λ-calcul.

Sans entrer dans le détail de la construction, on remarque qu'il existe plusieurs points fixes différant essentiellement par leur domaine de définition. Par exemple, la fonction factorielle peut être considérée comme indéfinie pour des arguments négatifs. Néanmoins, des points fixes de la fonctionnelle $(\lambda \mathit{fact} \, . \, \lambda n \ . \ ... \ n * \mathit{fact}(n-1) \ ...)$ peuvent décider de produire par exemple -10 ou 1234 pour $n = -1$. Les solution de l'équation $\mathit{fact} = \mathsf{Fix}(\mathit{fact})$ sont donc multiples, et « plus ou moins définies » (cf. l'ordre partiel des CPO). La solution qui est effectivement calculée est *la plus petite* : on l'appelle *le plus petit point fixe*.

Par exemple, en nous plaçant – par souci de simplicité – dans un λ-calcul doté des primitives nécessaires [3] et en utilisant une syntaxe OCaml, pour :

```
let rec fact n =
  if n = 0 then 1 else n * fact(n−1)
```

La fonctionnelle associée est :

```
let ffact fact = fun n →
  if n = 0 then 1 else n * fact(n−1)
```

Si on se dote d'un opérateur de point fixe Fix, on a alors :

(Fix `ffact`) 3
$\leftrightarrow_{\mathsf{Fix}}$ (`ffact` (Fix `ffact`)) 3
$\leftrightarrow_{\beta}$ (**fun** n → **if** n=0 **then** 1 **else** n * (Fix `ffact`)(n−1)) 3
$\leftrightarrow_{\beta,\delta}$ **if** 3=0 **then** 1 **else** 3 * (Fix `ffact`)(3−1)
$\leftrightarrow_{\delta}$ 3 * ((Fix `ffact`) 2)

$\leftrightarrow_{\mathsf{Fix}}$ 3 * ((`ffact` (Fix `ffact`)) 3)
$\leftrightarrow_{\beta}$ 3 * ((**fun** n → **if** n=0 **then** 1 **else** n * (Fix `ffact`)(n−1)) 2)
$\leftrightarrow_{\beta,\delta}$ 3 * (**if** 2=0 **then** 1 **else** 2 * (Fix `ffact`)(2−1))
$\leftrightarrow_{\delta}$ 3 * 2 * ((Fix `ffact`) 1)

$\leftrightarrow_{\mathsf{Fix}}$ 3 * 2 * ((`ffact` (Fix `ffact`)) 1)
$\leftrightarrow_{\beta}$ 3 * 2 * ((**fun** n → **if** n=0 **then** 1 **else** n * (Fix `ffact`)(n−1)) 1)
$\leftrightarrow_{\beta,\delta}$ 3 * 2 * (**if** 1=0 **then** 1 **else** 1 * (Fix `ffact`)(1−1))
$\leftrightarrow_{\delta}$ 3 * 2 * 1 * ((Fix `ffact`) 0)

$\leftrightarrow_{\mathsf{Fix}}$ 3 * 2 * 1 * ((`ffact` (Fix `ffact`)) 0)
$\leftrightarrow_{\beta}$ 3 * 2 * 1 * ((**fun** n → **if** n=0 **then** 1 **else** n * (Fix `ffact`)(n−1)) 0)

3. Avec les règles de réduction nécessaires (addition, test d'égalité, *etc.*). On note de façon générique $\leftrightarrow_{\delta}$ les conversions correspondantes.

$\longleftrightarrow_{\beta,\delta}$ $3 * 2 * 1 *$ (**if** 0=0 **then** 1 **else** $0 *$ (Fix `ffact`)(0−1))
$\longleftrightarrow_{\delta}$ $3 * 2 * 1 * 1$
$\longleftrightarrow_{\delta}$ 6

La difficulté posée par Y est que YF ne se *réduit* pas à proprement parler en $F(YF)$, il est seulement *inter-convertible* avec lui. Il existe bien d'autres combinateurs de point fixe, dont certains ont la propriété souhaitée. Par exemple, on peut vérifier que le terme ci-dessous, noté Θ :

$$(\lambda x.\lambda f.f(xxf))(\lambda x.\lambda f.f(xxf))$$

est un combinateur de point fixe tel que $\Theta M \rightarrow^*_\beta M(\Theta M)$. Un autre combinateur, noté Z, est le suivant : $\lambda f.((\lambda x.f(\lambda y(xx)y))(\lambda x.f(\lambda y(xx)y)))$, et il est particulièrement utile pour simuler la récursion dans la classe des langages à laquelle appartient OCaml.

10.5 Stratégies d'évaluation

Nous avons vu plus haut que le choix du radical n'avait pas d'influence sur le résultat des réductions, quand ces réductions terminaient sur des termes irréductibles. En revanche, nous avons aussi vu que le choix du radical influait sur la capacité à trouver un résultat pour des termes qui ne sont que faiblement normalisants. Par exemple, il existe des réductions infinies du terme $(\lambda x.(\lambda y.y))\Omega$ (avec $\Omega = (\lambda x.xx)(\lambda x.xx)$) que l'on obtient en choisissant toujours le radical le plus interne (Ω). Si par contre on choisit le radical externe (ici, la racine du terme lui-même), on fait purement et simplement disparaître Ω, et on obtient en une seule étape une forme irréductible : $(\lambda y.y)$. Effectuer un choix *systématique* de radical définit une *stratégie*. Les stratégies courantes sont relativement simples. En représentant les termes du λ-calcul comme des arbres, elles consistent généralement à choisir le prochain radical selon deux axes : le plus externe ou le plus interne, le plus à gauche, ou le plus à droite.

10.5.1 Stratégies internes

Les stratégies internes consistent à aller choisir le prochain radical à l'intérieur des termes. Dans le cas de $(\lambda x.(\lambda y.y))\Omega$, une stratégie interne va nécessairement choisir Ω comme prochain radical. Les stratégies internes correspondent partiellement [4] à ce qui se passe dans un langage de programmation comme C ou OCaml : lorsqu'une fonction est appliquée à des arguments, ceux-ci sont évalués *avant* que (la β-réduction correspondant à) l'application de la fonction ne soit exécutée.

Parmi les stratégies internes, on peut identifier celles qui iront choisir le radical le plus interne le plus à droite ou le plus à gauche. Par exemple, une fonction OCaml à plusieurs arguments procèdera d'abord à l'évaluation du dernier (le plus à droite). On le vérifie aisément en exécutant :

```
let f x y = (Printf.printf "exécution de f; %!") in
f (Printf.printf "arg 1; %!") (Printf.printf "arg 2; %!")
```

ce qui imprimera :

```
arg 2; arg 1; exécution de f;
```

L'avantage des stratégies internes est qu'elles permettent de ne procéder qu'une seule fois à l'évaluation des arguments des fonctions, puis de les associer déjà évalués aux paramètres formels de la fonction. Comme on l'a déjà vu, l'inconvénient est que leur évaluation aura lieu même si les paramètres formels correspondants ne sont pas utilisés. C'est exactement ce qui se passe dans le terme $(\lambda x.(\lambda y.y))\Omega$, qui ne contient aucune utilisation du paramètre x. Ces stratégies sont donc en quelque sorte prévoyantes, mais elles le sont un peu trop.

4. Dans les langages de programmation, les corps de fonction, procédure ou méthode, ne sont évalués que lorsque les arguments sont fournis, c'est-à-dire après que la β-réduction correspondant à leur application ait été effectuée.

Lorsqu'on encode la récursion en utilisant un opérateur de point fixe, les stratégies internes nécessitent une certaine prudence dans le choix du combinateur à utiliser. Le combinateur Θ donné ci-dessus ne convient pas, par exemple : puisque $\Theta M \xrightarrow{*}_\beta M(\Theta M)$, on voit aisément qu'une stratégie interne ira réduire ΘM dans $M(\Theta M)$, conduisant nécessairement à une réduction infinie.

10.5.2 Stratégies externes

Les stratégies qui vont aller chercher les radicaux externes vont travailler assez différemment : elles vont procéder aux appels de fonction sans en évaluer les arguments. De la sorte, l'évaluation des arguments inutilisés n'aura pas lieu, et c'est ce qui permet à ces stratégies de trouver la forme normale de $(\lambda x.(\lambda y.y))\Omega$, puisqu'elles vont immédiatement éliminer Ω.

Sur un terme de la forme $(\lambda x.e_1)e_2$, une stratégie interne va d'abord normaliser e_1 et e_2 en v_1 et v_2 respectivement, puis évaluer $v_1[v_2/x]$. Une stratégie externe, quant à elle, ira évaluer $e_1[e_2/x]$: si donc il n'y a pas d'occurrence libre de x dans e_1, l'évaluation de e_2 n'aura pas lieu. Par contre, l'évaluation de e_2 pourra avoir lieu plusieurs fois si x a plusieurs occurrences libres dans e_1.

En revanche, les stratégies externes offrent un plus grand choix de combinateurs de point fixe utilisables pour encoder la récursion.

10.5.3 Réduction faible

Nous avons déjà mentionné le fait que dans les langages de programmation, on ne procédait à l'exécution des corps de fonctions que lorsque les arguments étaient disponibles et que la fonction était en cours d'application. En d'autres termes, on ne procède pas à l'évaluation « sous les λ ». Les stratégies qui, ainsi, ne vont pas « réduire sous les λ » sont appelées *stratégies de réduction faible*. Par opposition, les stratégies qui vont réduire les corps des abstractions sont appelées *stratégies de réduction forte*.

Les stratégies faibles sont trop faibles pour garantir l'obtention de formes normales : elles ne peuvent à l'évidence réduire plus avant $\lambda y.(\lambda x.x)(\lambda x.x)$: il reste un radical, mais celui-ci étant sous un λy, il est hors du champ d'action des stratégies de réduction faible.

On peut formaliser les réductions faibles en définissant ce qu'est une *valeur*, c'est-à-dire une forme déjà évaluée, sur laquelle une réduction faible ne peut plus agir :

$$v ::= xv_1 \dots v_n \mid \lambda x.e \qquad \text{avec } n \geq 0$$

Une valeur est donc ou bien une variable (premier cas, avec $n = 0$), une application dont le premier terme (en position fonction) est une variable, ou alors une abstraction dont le corps est un λ-terme quelconque. Si on note v (v_1, v_2, *etc.*) de telles valeurs, une stratégie de réduction faible peut se décrire au moyen des règles suivantes :

$$\frac{}{(\lambda x.e)v \to e[v/x]}(\beta) \qquad \frac{e_1 \to e_1'}{e_1e_2 \to e_1'e_2}(Fun) \qquad \frac{e_2 \to e_2'}{e_1e_2 \to e_1e_2'}(Arg)$$

Il faut lire une règle $\dfrac{P_1 \dots P_n}{Q}$ comme « si P_1 et … et P_n, alors Q » : c'est ce qu'on appelle une *règle d'inférence*. Les P_i sont appelés les *prémisses* : lorsqu'ils sont inexistants, on écrit des règles sans numérateur comme : $\dfrac{}{Q}$, ou simplement Q, et on dit que la règle correspondante est un *axiome*. On nomme quelquefois les règles en notant leur nom entre parenthèses à la droite de chaque règle.

Ici, le seul axiome du système nous indique que la β-réduction ne s'applique que si l'argument de l'abstraction est une valeur. Les deux règles d'inférence nous disent dans quels contextes on peut procéder à des réductions.

Les stratégies internes de réduction faible sont utilisées par les langages de programmation comme C ou OCaml, puisque les fonctions évaluent leurs arguments avant de s'exécuter elles-même, et aucune

réduction n'est opérée dans les corps de fonction tant que celles-ci ne sont pas appliquées. On appelle aussi ce mode d'évaluation *appel par valeur* ou *évaluation stricte*.

Ce mode d'évaluation est la raison pour laquelle la conditionnelle est toujours une construction spéciale dans ces langages au lieu d'être une fonction. En effet, si la conditionnelle était une fonction dans de tels langages, elle évaluerait ses trois arguments avant de produire l'un des deux derniers selon la valeur booléenne du premier, et perdrait du même coup tout son intérêt. Par contre, dans les langages dont le mode d'évaluation correspond à une stratégie externe, la conditionnelle peut sans problème être une fonction.

10.5.4 Standardisation

La réduction externe gauche [5] cherche à résoudre un radical le plus proche possible de la racine du terme (vu sous forme d'arbre), et ira chercher ce radical le plus à gauche. Aussi bizarre que cela puisse paraître, cette stratégie est très particulière : si le terme à réduire est normalisable (même seulement faiblement), cette stratégie trouvera à coup sûr sa forme normale. Alors qu'elle apparaît dupliquer du travail lorsqu'une fonction utilise plusieurs occurrences de son argument, cette stratégie est *sûre*, comme l'indique le théorème suivant :

Théorème de standardisation *Si le terme e est faiblement normalisant, alors la stratégie externe gauche trouvera sa forme normale en un nombre fini d'étapes.*

Nous ne donnerons pas ici la preuve de ce théorème. Intuitivement, la stratégie externe gauche n'effectue pas de réductions inutiles (réduire un argument inutilisé) : elle n'effectue donc que du travail utile.

Si on transpose les stratégies externes dans les langages de programmation, on les retrouve notamment sous le vocable *d'appel par nom*, qui représente la substitution du nom d'un paramètre formel par le *texte* de l'argument correspondant.

La stratégie d'évaluation correspondante peut être définie par les règles suivantes :

$$\frac{}{(\lambda x.e_1)e_2 \to e_1[e_2/x]}(\beta) \qquad \frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2}(Fun)$$

La seconde règle indique qu'il faut aller chercher le radical dans la partie « fonction » d'une application, et la première indique que la règle de β-réduction est applicable sans contrainte.

La difficulté avec cette manière de procéder est que si l'argument est réductible et si le paramètre initial a plusieurs occurrences *utiles* au calcul, alors l'argument sera réduit plusieurs fois. Une autre façon de procéder consiste à mémoriser le résultat v du premier calcul de l'argument en question, et à transmettre v directement en résultat (sans calcul) lorsqu'une autre occurrence de ce même calcul sera nécessaire. Ce mode d'évaluation, adopté par toute une famille de langages dont le langage Haskell est maintenant le représentant, est appelé *évaluation paresseuse*, ou *évaluation non stricte*. Dans ces langages, seulement les calculs nécessaires à l'obtention du résultat seront opérés. Par exemple, le calcul d'une donnée structurée ne nécessite pas immédiatement le calcul de chacune de ses composantes : la liste (e_1::e_2) ne nécessitera l'évaluation de e_1 ou de e_2 que si (et lorsque) leur valeur est nécessaire. Cela permet aux langages dits *paresseux* de manipuler des données structurées représentant des données infinies comme par exemple la liste des entiers naturels (**let rec** `nats` = 0 :: `List.map succ nats`) et, de ce fait, d'exprimer très élégamment et de façon très concise la résolution de nombreux problèmes qui nécessiteraient un développement impliquant plus de codage dans un langage classique.

5. *Leftmost outermost*, en Anglais

Chapitre 11

Compilation (naïve) vers du code assembleur

Nous avons vu dans le chapitre 5 qu'il était relativement aisé produire du code pour une machine virtuelle à partir de la sémantique opérationnelle d'un langage. L'exécution de ce code s'appuyait sur la structure de la machine virtuelle et le comportement de ses instructions. Ce travail de compilation était grandement facilité par la structure relativement haut-niveau de la machine virtuelle, qui exploitait des structures de données OCaml : types sommes, listes et même ne serait-ce que les chaînes de caractères. Pour représenter les valeurs possibles dans la pile, nous utilisions le type `vm_val` sans nous soucier de la représentation bas-niveau de ses valeurs. De même, pour représenter du code, nous utilisions une liste d'instructions elles-mêmes représentées par un type somme. Autrement dit, nous nous appuyions très fortement sur les facilités offertes par notre langage d'implantation.

Lorsque l'on veut générer du code que le microprocesseur peut directement interpréter, nous perdons toutes ces facilités. Les instructions sont celles imposées par le microprocesseur. La mémoire (dont la pile) n'est qu'une suite d'octets sans type ni structure qu'il faut explicitement gérer (allocation de variables, passage d'arguments aux fonctions). Le flot de contrôle du programme (boucles, tests, appels de fonctions) doit être explicité en terme d'instructions du microprocesseur. Ainsi, le travail devient nettement plus compliqué.

Dans ce chapitre, nous allons faire une introduction à la compilation vers du code assembleur. Le but n'est pas de produire du code efficace comme le font les compilateurs d'aujourd'hui, mais de se rendre compte que même pour un compilateur relativement naïf, certains problèmes absents de la compilation vers une machine virtuelle apparaissent. Nous allons donc réaliser un compilateur d'un petit langage impératif restreint vers du code assembleur X86-64. La phase de génération de code de ce compilateur sera faite en une seule passe afin de simplifier la présentation. Cela explique la naïveté du code produit (il est impossible de produire du code efficace pour un langage réaliste en une seule passe). Les compilateurs « intelligents » comportent de nombreuses passes qui permettent d'effectuer différentes transformations et optimisations.

Le code généré sera constitué de mnémoniques assembleur. Il restera encore du travail à effectuer pour obtenir un fichier exécutable. Tout d'abord, *l'assemblage* qui consiste à transformer ces mnémoniques en code binaire objet. C'est le rôle de *l'assembleur* (on utilise souvent ce même terme pour parler du *logiciel d'assemblage* et du code source qu'il traite). On peut schématiser le travail de ce dernier comme l'encodage des mnémoniques en suite d'entiers suivant la structure des instructions que représentent ces mnémoniques. Dans les faits, le travail de l'assembleur n'est pas aussi simple car il doit effectuer des calculs d'adresses, peut encore faire des optimisations et dispose souvent d'un système de macros qu'il faut traiter. Une fois qu'un fichier objet est obtenu pour chaque fichier assembleur du programme, il faut regrouper ces objets pour en faire un exécutable final. C'est le rôle de *l'éditeur de liens*. Finalement, au moment de l'exécution, un dernier acteur apparaît sur les OS modernes : *l'éditeur de liens dynamique*

qui va se charger de déterminer les adresses des symboles en provenance de bibliothèques dynamiques.

Tous ces aspects sont en dehors du cadre de ce document. Nous partons du principe que nous avons à disposition un assembleur et un OS qui nous permettront de générer un exécutable à partir du code assembleur produit puis de l'exécuter.

11.1 Langage source

Le langage source considéré, nommé IMP, est un petit langage impératif ne disposant que d'entiers (signés) et de tableaux d'entiers (de taille fixe) comme types de données. Les variables sont locales aux fonctions (pas de variables globales). Les paramètres de fonction sont uniquement des entiers (en particulier pas des tableaux). Parmi les fonctions (possiblement récursives), une seule doit être nommée `main` et sert de point d'entrée du programme. La syntaxe abstraite de ce langage est résumée dans le tableau ci-dessous.

$$
\begin{array}{rcl}
e & ::= & i \mid x \mid x(e_1, \ldots, e_n) \mid x[e] \\
& \mid & e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \mid -e \\
& \mid & e_1 == e_2 \mid e_1 != e_2 \mid e_1 < e_2 \mid e_1 <= e_2 \mid e_1 > e_2 \mid e_1 >= e_2 \\
i & ::= & \texttt{while}\ e\ \texttt{do}\ i\ \texttt{done} \mid \texttt{if}\ e\ \texttt{then}\ i_1\ \texttt{else}\ i_2\ \texttt{endif} \mid \texttt{if}\ e\ \texttt{then}\ i\ \texttt{endif} \\
& \mid & i\ ;\ i \mid x = e \mid x[e_1] = e_2 \mid \texttt{return}\ e \mid \texttt{return} \mid x(e_1, \ldots, e_n) \mid \texttt{print}\ (e) \\
d & := & \texttt{var}\ x \mid \texttt{var}\ x = e \mid \texttt{var}\ x[i] \\
f & := & x(x_1, \ldots x_n)\ \texttt{begin}\ d_1 \ldots d_n\ i\ \texttt{end} \\
t & := & f \mid \texttt{extern}\ x(x_1, \ldots x_n) \\
p & := & t_1 \ldots t_n
\end{array}
$$

On remarque parmi les éléments globaux (t pour *toplevel*) la présence de prototypes de fonctions (`extern`). Cela est nécessaire afin que le compilateur, en présence d'une fonction définie plus tard, sache combien elle a de paramètres. C'est la raison pour laquelle en OCaml les fonctions d'un même fichier doivent être définies avant d'être utilisées et les fichiers d'interfaces existent pour permettre de connaître le type des fonctions des autres fichiers. En C, ce sont les fichiers d'en-tête et la directive `#include` qui jouent ce rôle.

11.2 Assembleur cible

La passe de génération de code assembleur diffère selon le microprocesseur ciblé puisque chacun dispose d'instructions propres. Dans les compilateurs, un grand nombre de passes sont communes quel que soit l'assembleur visé, néanmoins la production d'instructions effectives (et certaines optimisations) doit être différenciée en fonction du processeur visé.

Le choix effectué dans ce document est l'assembleur pour les processeurs compatibles avec le jeu d'instructions X86-64 (Intel, AMD). Ces processeurs sont largement répandus et équipent un grand nombre d'ordinateurs (personnels et professionnels).

Pour être capable de compiler vers l'assembleur d'un microprocesseur il est nécessaire de connaître son architecture et ses instructions. La présentation complète de l'architecture X86-64 est totalement hors du périmètre de ce document (rien que le document *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2* fait presque 2200 pages !). Nous allons nous restreindre à l'infime partie que nous utiliserons durant la compilation, en adoptant la syntaxe AT&T.

11.2.1 Registres

Nous disposons de 16 registres « de travail » permettant de manipuler des entiers (ou des adresses) sur 64 bits (il existe d'autres registres spécialisés) : `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`, `%r8`,

`%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14` et `%r15`. Il est possible d'accéder aux 32 bits « bas » de ces registres (pour travailler sur des entiers sur 32 bits) au travers de : `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`, `%esp`, `%r8d`, `%r9d`, `%r10d`, `%r11d`, `%r12d`, `%r13d`, `%r14d`, et `%r15d`. Le registre `%rsp` joue le rôle de pointeur de pile et contient l'adresse de la dernière case utilisée de la pile (il est automatiquement mis à jour par certaines instructions comme `pushq` et `popq`).

Afin de garantir l'interopérabilité entre les codes produits par différents compilateurs, des conventions (*ABI* pour *Application Binary Interface*) sont spécifiées par les systèmes d'exploitation (par exemple System V AMD64 ABI est respectée par Linux, MacOS, FreeBSD et Solaris). Ces conventions décrivent l'ordre d'allocation des variables, comment les paramètres sont passés aux fonctions (registres ou pile, dans quel ordre), quels sont les registres devant être sauvegardés avant un appel de fonction, quels sont les usages spécifiques de certains registres, comment doit être organisée la pile, etc.

Nous n'allons pas suivre d'ABI particulière puisque nous ne cherchons pas à créer un compilateur réaliste : nous déciderons de la nôtre. Nous n'utiliserons que très peu de registres (majoritairement `%rax`/`%eax`, `%rsp`, `%rbp` et très occasionnellement un ou deux autres).

11.2.2 Mémoire

Le modèle de mémoire adopté est dit *plat* : toute la mémoire peut être accédée linéairement, avec des adresses sur 64 bits. La pile d'exécution est allouée dans cette mémoire au début de l'exécution d'un binaire. Cette pile croît vers les adresses décroissantes (on empile en diminuant le pointeur de pile). C'est dans cette zone que le programme va explicitement stocker les variables locales et les arguments des appels de fonction (le code généré contenant explicitement les instructions de manipulation de pile). C'est également dans cette zone que l'adresse de retour d'un appel de fonction va automatiquement être stocké par l'instruction dédiée (`call`). Comme précédemment indiqué, le registre `%rsp` désigne à tout instant l'adresse de la dernière case utilisée dans la pile.

Il est important de comprendre comment les octets sont stockés en mémoire. Les microprocesseurs Intel et AMD sont dits *little endian* (en français *petit boutiste*) : lorsqu'un mot de 32 bits est stocké en mémoire, son octet de poids faible est à l'adresse la plus faible, et son octet de poids le plus fort est à l'adresse la plus forte. Si l'on considère une variable située à l'adresse 10 dont la valeur est `0x33221100` (en hexadécimal), alors l'organisation individuelle des octets est :

Adresse	@10	@11	@12	@13
Octet	$0x00$	$0x11$	$0x22$	$0x33$

11.2.3 Opérandes, modes d'adressage

Nous utiliserons 3 modes d'adressage des opérandes :

- $*imm* : l'adressage immédiat qui représente la valeur constante entière *imm* (écrite en décimal) sur 32 bits.
- *reg* : qui représente la valeur contenue dans le registre *reg*.
- $*imm*(*base*) : l'adressage (indirect) basé, qui représente la valeur contenue en mémoire à l'adresse *base* + *imm* où *base* est un registre.
- $*imm*(*base*, *index*, *scale*) : l'adressage (indirect) basé indexé avec facteur qui représente la valeur contenue en mémoire à l'adresse *base* + *index* $*$ *scale* où *base* et *index* sont des registres et *scale* est 1, 2, 4 ou 8.

11.2.4 Instructions

Le tableau qui suit liste les seules instructions dont nous allons avoir besoin. De nombreuses instructions existent avec 4 suffixes en fonction de la taille de leur(s) opérandes() : `b` → 1 octet, `w` → 2 octets, `l` → 4 octets et `q` → 8 octets. Dans la majorité des cas, les instructions ne peuvent utiliser qu'une seule opérande représentant un accès mémoire.

`movl`/`movq` s, d	$d \leftarrow s$
`addl`/`addq` s, d	$d \leftarrow d + s$
`subl`/`subq` s, d	$d \leftarrow d - s$
`imull` s, d	$d \leftarrow d * s$ (troncature du résultat sur 32 bits stocké dans le registre d)
`idivl` s	$\%eax \leftarrow (\%edx : \%eax) / s$ et $\%edx \leftarrow (\%edx : \%eax)\ mod\ s$
`negl` s	$s \leftarrow -s$
`cmpl` s, d	compare s et d en faisant $d - s$ (et positionne les bits du registre de statut)
`pushq` s	$\%rsp \leftarrow \%rsp - 8$ puis $(\%rsp) \leftarrow s$
`popq` d	$d \leftarrow (\%rsp)$ puis $\%rsp \leftarrow \%rsp + 8$
`callq` *label*	empile l'adresse de l'instruction suivante et saute à l'étiquette *label*
`retq`	dépile l'adresse du sommet de la pile et saute à cette adresse
`jmp` *label*	saute à l'étiquette *label*
`j`*cc label*	saute à l'étiquette *label* si la condition *cc* est vraie (cf. explications suivantes)

L'instruction `j`*cc* est en fait une famille d'instructions où *cc* représente le mnémonique d'une condition. Lorsqu'une instruction est exécutée, différents bits du *registre de statut* sont positionnés pour refléter différentes conditions survenues (résultat nul, résultat négatif, débordement, parité, retenue, etc.). En particulier, on fait suivre l'instruction `cmpl` d'un saut conditionnel `j`*cc* pour exploiter ces bits et réaliser un branchement conditionnel. Les différents sauts conditionnels (relatifs à des comparaisons entre entiers signés) qui nous intéressent sont décrits dans le tableau suivant.

`je`	si égal
`jne`	si différent
`jle`	si inférieur ou égal
`jl`	si inférieur
`jge`	si supérieur
`jg`	si supérieur ou égal

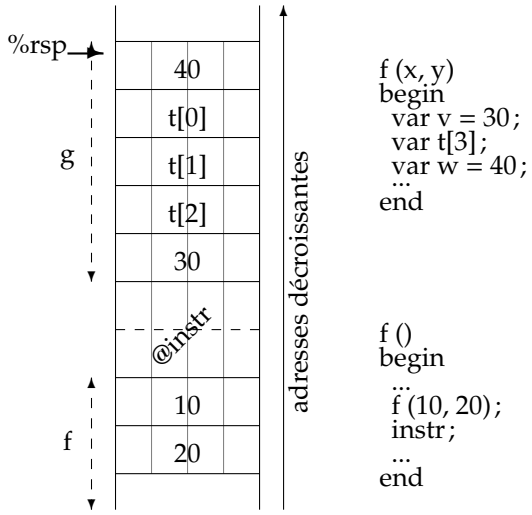
11.3 Modèle de compilation

À des fins de simplification, nous avons fait le choix de ne pas avoir de variables globales dans notre langage. Ainsi, toutes les variables pourront être allouées sur la pile d'exécution. Ces variables seront allouées sur la pile dans l'ordre de leur déclaration. Étant toutes de type entier, elles prendront chacune 4 octets.

De manière similaire, nous faisons le choix de passer les arguments par la pile lors des appels de fonction. Bien évidemment, ceci n'est pas optimal et ne respecte en particulier pas l'ABI System V dont il a été question dans la section 11.2.1, qui impose l'utilisation des registres pour les 6 premiers arguments. Cela présente néanmoins l'avantage d'avoir un traitement homogène quel que soit le nombre d'arguments d'une fonction et évite de devoir calculer quels sont les registres à sauvegarder puis restaurer lors d'un appel. Par choix, les arguments seront empilés de droite à gauche. Puisque notre langage ne dispose que d'entiers, chaque argument occupera 4 octets.

Pour terminer, nous faisons le choix de n'utiliser que le registre `%eax` et la pile pour évaluer les expressions. Bien que peu efficace car provoquant une utilisation intensive de la pile, ce modèle de compilation reste relativement simple. Il se rapproche donc de celui présenté dans le chapitre 5.

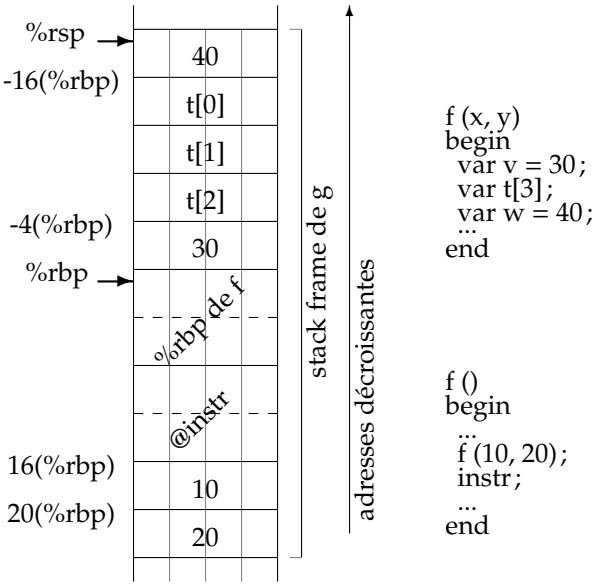
L'une des différences importantes avec le modèle de ce précédent chapitre est le besoin de manipuler explicitement la pile, besoin renforcé par l'absence de registre d'environnement via lequel nous sauvegardions les liaisons entre identificateurs et valeurs. Lors de l'exécution d'une fonction, la pile contient ses arguments, l'adresse de retour de l'appel et les variables locales de la fonction. Par ailleurs, au cours des calculs, des valeurs temporaires pourront également apparaître. Comme indiqué précédemment, variables et arguments occuperont chacun 4 octets. L'adresse de retour, elle, occupera 8 octets comme imposé par le microprocesseur.



Le schéma ci-contre nous montre l'état de la pile lors d'un appel à la fonction g par la fonction f. On remarque en particulier que le pointeur de pile `%rsp` pointe sur la mémoire où est stockée w (plus exactement sur l'adresse contenant l'octet de poids faible de w puisque, comme indiqué en section 11.2.2, le microprocesseur est little endian : les octets de poids plus forts suivent aux adresses croissantes). Comment savoir où accéder dans la pile pour retrouver les valeurs des paramètres et des variables locales ? On est tenté de la faire par rapport à `%rsp`. Malheureusement, ce dernier bouge au gré des calculs intermédiaires effectués dans le corps de g. Une solution simple consiste à utiliser un registre (`%rbp`) pour mémoriser la valeur du pointeur de pile à l'entrée de la fonction. Ainsi, en prenant soin de ne pas y toucher dans le corps de la fonction, il sera aisé d'accéder aux variables et aux paramètres. Le registre `%rbp` est alors appelé *frame pointer*. Une *frame* (*bloc d'activation*, *activation frame* ou *activation record*) est l'ensemble des informations enregistrées dans la pile correspondant à un appel de fonction en cours (non terminé).

Si à l'entrée de chaque fonction, le code généré sauvegarde `%rsp` dans `%rbp`, il faut être capable de restaurer ce dernier en fin de fonction afin que l'appelant le retrouve intact à l'issue de l'appel. Ainsi, le prélude et le postlude de chaque fonction doivent être de la forme :

```
fct_label:
  pushq %rbp
  movq %rsp, %rbp
  ... compilé du corps de la fonction
  popq %rbp
  retq
```



Nous obtenons alors l'organisation de la pile telle que décrite dans le schéma ci-contre dans lequel nous voyons que nous pouvons accéder aux paramètres en utilisant un décalage positif par rapport à `%rbp` et aux variables locales utilisant un décalage négatif. Ces décalages sont prévisibles à la compilation et ce sera le travail du compilateur que de les calculer.

Une subtilité vient se rajouter si l'on souhaite pouvoir appeler des fonctions compilées par d'autres compilateurs, en particulier respectant l'ABI System V. Nous avions dit en section 11.2.1 et 11.3 que nous ne chercherions pas à respecter cette ABI. Nous allons quand même nous y plier au minimum, juste pour pouvoir compiler l'instruction `print` qui nécessitera d'appeler la fonction `printf` de la bibliothèque standard du compilateur utilisé pour ultimement assembler notre code produit. La seule contrainte que nous allons respecter est qu'au moment où une instruction d'appel de fonction (`callq`) va être exécutée, le pointeur de pile `%rsp` doit contenir une adresse multiple de 16 octets. On dit que la pile doit être « alignée sur 16 octets ».

Cela va nécessiter deux choses : garder trace pendant la compilation du décalage actuel de la pile depuis l'entrée dans la fonction et générer une éventuelle « correction » de la pile avant d'empiler les paramètres d'un appel pour que cette dernière soit bien alignée à l'issue de ces empilements. Cette correction va consister en la soustraction d'un certain nombre d'octets à `%rsp`, qui devra être rectifiée par une addition d'autant à l'issue de l'appel. Cette correction doit être effectuée juste avant d'empiler les arguments afin de ne pas rendre faux les calculs de décalages qui permettront d'accéder aux paramètres du côté de l'appelé. En effet, si l'on avait fait cette correction après avoir empilé les arguments, alors l'appelé n'aurait aucun moyen de savoir de combien la pile aurait été décalée (la correction dépendant de l'état de la pile de l'appelant et non de l'appelé). Ainsi, il ne pourrait plus accéder aux arguments.

Pour garder trace du décalage actuel de la pile par rapport à un alignement sur 16 octets, nous devons nous baser sur l'hypothèse que la fonction courante a également été appelée avec la pile alignée sur 16 octets au moment du `callq` (c'est en quelque sorte une hypothèse de récurrence). Attention, après l'exécution du `callq`, l'adresse de retour aura été empilée ! Cela signifie qu'à l'entrée d'une fonction la pile n'est plus alignée sur 16 octets, mais est décalée de -8 octets (taille de l'adresse de retour). Il faudra donc garder trace de ce décalage initial à l'entrée de chaque fonction. Ensuite, à chaque fois que l'on générera une instruction qui modifie `%rsp`, on mettra à jour le décalage actuel (`pushq` l'augmentera, `popq` le diminuera). Lorsque l'on effectuera un appel de fonction, on devra savoir quelle taille ses arguments prendront sur la pile et calculer l'ajustement à faire pour qu'une fois les arguments empilés, la pile soit alignée. La correction à appliquer s'obtient donc par :

$$\mathit{correction} = \begin{cases} 0 & \text{si } \mathit{décalage} \,\%\, 16 = 0 \\ 16 - (\mathit{décalage} \,\%\, 16) & \text{sinon} \end{cases}$$

en considérant que *décalage* est le nombre (positif) d'octets alloués sur la pile entre le point de programme courant et l'entrée dans la fonction. La *correction* devra donc être soustraite de `%rsp` pour aligner la pile. À l'issue de l'appel de fonction, *correction* devra rajoutée à `%rsp` pour annuler l'alignement.

On comprend maintenant la nécessité que les compilateurs ont de connaître le prototype des fonctions : pour réaliser l'appel, il est nécessaire de savoir combien de place prennent les arguments sur la pile. On pourrait être tenté de dire qu'il n'y a qu'à regarder les arguments effectifs au moment de l'appel pour calculer cette place. Cela est incorrect car une fonction prenant un `long int` en argument, pourra être appelée avec un `short int`, ce dernier étant automatiquement étendu sur 64 bits (par un mécanisme dit de *promotion* comme c'est le cas en C). Ainsi, seule la fonction appelée peut décrire clairement ce qu'elle attend et comment elle va accéder à la pile pour manipuler ses paramètres.

11.3.1 Compilation des expressions

Maintenant que nous avons examiné l'organisation de la pile, nous pouvons donner la fonction de compilation des expressions. Nous dénotons par $[\![e]\!]_{\rho,s}$ le code généré pour l'expression e dans l'environnement ρ avec le décalage courant de pile s. L'environnement ρ fait correspondre à un identificateur le déplacement nécessaire par rapport à `%rbp` pour y accéder. Comme vu précédemment, si l'identificateur représente un paramètre, le déplacement est positif, s'il représente une variable locale, il est négatif. Pour ne pas surcharger d'avantage la notation, on considérera que si l'identificateur est une fonction, alors ρ retourne le nombre d'octets global de ses paramètres (pour être plus propre, il faudrait un second environnement, ce qui est recommandé dans une implémentation du compilateur). Pour bien différencier le code émis, nous le préfixons par le symbole $\rightarrow$.

$[\![i]\!]_{\rho,s}$ = → `movl` $\$i$, `%eax`	$[\![x]\!]_{\rho,s}$ = → `movl` $\rho(x)$(`%rbp`), `%eax`
$[\![t[e]]\!]_{\rho,s}$ = $[\![e]\!]_{\rho,s}$ → `movl` $\rho(t)$(`%rbp`, `%rax`, 4)	$[\![f\ (e_1, \ldots, e_n)]\!]_{\rho,s}$ = $s' = (s + \rho(f)) \,\%\, 16$ $corr = 0$ si $s' = 0$ sinon $16 - s'$ si $corr \neq 0$ alors → `subq` $\$corr$, `%rsp` $[\![(e_n, \ldots, e_1)]\!]_{\rho,s,0}$ si $\rho(f) \neq 0$ alors → `subq` $\$\rho(f)$, `%rsp` → `callq` si $\rho(f) \neq 0$ alors → `addq` $\$\rho(f)$, `%rsp` si $corr \neq 0$ alors → `addq` $\$corr$, `%rsp`
$[\![-e]\!]_{\rho,s}$ = $[\![e]\!]_{\rho,s}$ → `negl` `%eax`	$[\![e_1 + e_2]\!]_{\rho,s}$ = $[\![e_2]\!]_{\rho,s}$ → `pushq` `%rax` $[\![e_1]\!]_{\rho,s+8}$ → `addl` 0(`%rsp`), `%eax` → `addq` $8, `%rsp`
$[\![e_1 - e_2]\!]_{\rho,s}$ = $[\![e_2]\!]_{\rho,s}$ → `pushq` `%rax` $[\![e_1]\!]_{\rho,s+8}$ → `subl` 0(`%rsp`), `%eax` → `addq` $8, `%rsp`	$[\![e_1 * e_2]\!]_{\rho,s}$ = $[\![e_2]\!]_{\rho,s}$ → `pushq` `%rax` $[\![e_1]\!]_{\rho,s+8}$ → `imull` 0(`%rsp`), `%eax` → `addq` $8, `%rsp`
$[\![e_1/e_2]\!]_{\rho,s}$ = $[\![e_2]\!]_{\rho,s}$ → `pushq` `%rax` $[\![e_1]\!]_{\rho,s+8}$ → `cdq` → `idivl` 0(`%rsp`) → `addq` $8, `%rsp`	$[\![e_1 \% e_2]\!]_{\rho,s}$ = $[\![e_2]\!]_{\rho,s}$ → `pushq` `%rax` $[\![e_1]\!]_{\rho,s+8}$ → `cdq` → `idivl` 0(`%rsp`) → `movl` `%edx`, `%eax` → `addq` $8, `%rsp`

Pour les opérateus binaires, nous voyons que nous empilons le résultat des calculs intermédiaires (stockés dans `%rax` issus des sous-expressions. Fort logiquement, en fin de calcul il faut corriger la pile pour défaire cet empilement. On remarque que l'on empile toujours le registre `%rax` (donc 8 octets) et non seulement `%eax` (les 4 octets qui représentent un entier). La raison est que l'instruction `pushq` ne permet d'empiler qu'un registre de 8 octets. Si nous avions voulu n'empiler que les 4 octets réellement utilisés par nos calculs, il aurait fallu le gérer « manuellement », avec un `movl ...,-4(%rsp)` suivi d'un `subq $4,%rsp`. En conséquence, s est incrémenté de 8 à chaque fois que l'on a empilé une valeur intermédiaire de `%eax` (calcul de $[\![e_1]\!]_{\rho,s+8}$).

L'accès dans un tableau utilise l'adressage basé indexé avec facteur. Le facteur est 4 puisque nous ne manipulons que des entiers sur 32 bits : chaque « case » du tableau fait 4 octets. Ce mode d'adressage permet donc d'éviter le calcul explicite du décalage induit par la « taille des cases ». Si notre langage disposait d'autres longueurs d'entiers, ce décalage dépendrait de la taille effective des entiers du tableau adressé : 1 pour un tableau de `char`, 2 pour un tableau de `short`, 8 pour un tableau de `long`. Dans le cas d'un tableau de données plus grandes que des `long`, le facteur d'échelle ne pourrait plus être utilisé et nous devrions expliciter son calcul par une instruction de multiplication (le code généré serait donc plus gros).

Lors d'un appel de fonction, on effectue bien la correction d'alignement de pile avant la compilation des arguments effectifs. La compilation de ces arguments est donnée par la fonction $[\![(e_1, \ldots, e_n)]\!]_{\rho,s,o}$ décrite ci-après. Notons que nous appelons cette fonction avec la liste des arguments effectifs en ordre inverse : nous avons choisi d'empiler les arguments de droite à gauche (cf. 11.3).

Nous pouvons remarquer l'instruction `cdq` utilisée dans la compilation de la division et du modulo. Elle sert à mettre étendre la valeur de `%eax` dans `%edx`. Ainsi, l'ensemble `%edx:%eax` ne forme qu'une seule valeur sur 64 bits, comme attendu par l'instruction `idivl`. Nous ne pouvons pas nous contenter de mettre `%edx` à 0 sous prétexte que nous effectuons une division sur 4 octets (valeur contenue dans `%eax`). En effet, si le dividende est négatif, son bit de signe doit être propagé dans `%edx`.

La compilation des arguments d'un appel de fonction est donnée par $[\![(e_1, \ldots, e_n)]\!]_{\rho,s,o}$ où o (pour *offset*) représente le décalage nécessaire pour écrire l'argument dans la pile (vu du point de vue de l'appelant). Étant donné que notre langage ne permet de manipuler que des entiers sur 4 octets, le décalage de o est toujours incrémenté de 4. Notons que o est un décalage en « valeur absolue » : pour adresser la pile et empiler les arguments, il faut se déplacer de $-o$ par rapport à `%rsp`.

$$
\begin{array}{lcl}
[\![()]\!]_{\rho,s,o} & = & \emptyset \\
[\![(e, \ldots)]\!]_{\rho,s,o} & = & [\![e]\!]_{\rho,s} \\
 & & o' = o + 4 \\
 & & \to \texttt{movl \%eax, } -o\texttt{(\%rsp)} \\
 & & [\![(\ldots)]\!]_{\rho,s,o'}
\end{array}
$$

11.3.2 Compilation des instructions

La difficulté majeure de la compilation des instructions est la gestion des conditions et des boucles. En effet, ces constructions doivent être compilées en utilisant des sauts (conditionnels et inconditionnels). Afin de donner une présentation générale des schémas de compilation de ces constructions, nous considérons que nous avons deux pseudo-instructions : `jtrue` qui permet de sauter si le résultat de la condition qui le précède et vraie et `jfalse` si elle est fausse. Le tableau suivant décrit le schéma de compilation pour chaque construction.

<table>
<tr>
<td>if e then i_1 else i_2</td>
<td>$[\![e]\!]$
jfalse .else
$[\![i_1]\!]$
jmp .exit
.else :
$[\![i_2]\!]$
.exit :</td>
<td>if e then i</td>
<td>$[\![e]\!]$
jfalse .exit
$[\![i]\!]$
.exit :</td>
</tr>
<tr>
<td colspan="4">while e do i done</td>
<td>.entry :
$[\![e]\!]$
jfalse .exit
$[\![i]\!]$
jmp .entry
.exit :</td>
</tr>
</table>

Le problème qui vient ensuite est de réellement compiler les conditions. Même si celles-ci sont des expressions telles que celles dont nous avons donné la compilation en section 11.3.1, elles n'ont plus du tout la même sémantique puisqu'elles doivent permettre de changer le flot d'exécution du programme.

De surcroît, les opérateurs && et || doivent (comme dans tout langage) être paresseux : ils n'évaluent leur arguments qu'en cas de besoin. Ainsi, l'exécution d'une condition e_1 && e_2 n'évaluera e_2 que si e_1 est vraie. À l'inverse, e_1 ||, e_2 n'évaluera pas e_2 si e_1 est vraie. Autrement dit, dès que l'on sait que la condition ne peut être vraie ou bien est toujours vraie, on arrête d'évaluer le reste de la condition. Ainsi, lorsque l'on compile un &&, on doit tester la négation de la condition et sauter à l'étiquette « fausse », sinon on continue à évaluer le reste de la condition. Lorsque l'on compile un ||, on doit tester la condition et sauter à l'étiquette « vraie », sinon on continue à évaluer le reste de la condition.

e_1 == 5 && e_2 == 6	$e_1 \neq 5$? `jfalse .false` $e_2 \neq 6$? `jfalse .false` *code gardé par la condition* `.jfalse` :

e_1 == 5 || e_2 == 6	e_1 == 5?
	`jtrue .true`
	e_2 == 6?
	`jtrue .true`
	`jmp .false`
	`.true` :
	code gardé par la condition
	`.false`

Nous pouvons remarquer que la forme du code produit pour un `||` n'est pas optimale. Nous aurions pu économiser une instruction de saut en testant la négation de la dernière sous-condition :

e_1 == 5?
`jtrue .true`
e_2 == 6?
`jfalse .false`
`.true` :
code gardé par la condition
`.false`

Générer un tel code est possible mais complique la fonction de compilation des conditions. Aussi, sous couvert de simplicité, nous nous contenterons du schéma présenté initialement.

La compilation s'étend à une combinaison de && et de `||`. Il faut donc, à chaque expression, savoir si l'on doit compiler en testant la condition pour sauter à l'étiquette « vraie » si la condition est vraie ou bien en testant la négation de la condition pour sauter à l'étiquette « fausse » si cette négation est vraie. La compilation des conditions est donnée par la fonction $[\![e]\!]_{\rho,s,t,lt,lf}$ où t est un booléen disant si l'on doit tester la condition ou sa négation (et sauter le cas échéant à l'étiquette adéquate), lt est l'étiquette où sauter si la condition est satisfaite et lf où sauter si sa négation l'est. On considère donnée une fonction `gen_label ()` qui retourne un nouveau nom d'étiquette.

$[\![e_1 \,\&\&\, e_2]\!]_{\rho,s,t,lt,lf}$ = *and2_lbl = gen_label ()* $[\![e_1]\!]_{\rho,s,false,and2\_lbl,lf}$ $\rightarrow$ *and2_lbl*: $[\![e_2]\!]_{\rho,s,false,lt,lf}$	$[\![e_1 \,\|\|\, e_2]\!]_{\rho,s,t,lt,lf}$ = *or2_lbl = gen_label ()* $[\![e_1]\!]_{\rho,s,true,tl,or2\_lbl}$ $\rightarrow$ *or2_lbl*: $[\![e_2]\!]_{\rho,s,true,lt,lf}$
$[\![e_1 == e_2]\!]_{\rho,s,t,lt,lf}$ = $[\![e_1]\!]_{\rho,s}$ $\rightarrow$ `pushq %rax` $[\![e_2]\!]_{\rho,s+8}$ $\rightarrow$ `addq $8, %rsp` $\rightarrow$ `cmpl %eax, -8(%rsp)` si t alors $\rightarrow$ `je` lt sinon $\rightarrow$ `jne` lf	$[\![e_1 < e_2]\!]_{\rho,s,t,lt,lf}$ = $[\![e_1]\!]_{\rho,s}$ $\rightarrow$ `pushq %rax` $[\![e_2]\!]_{\rho,s+8}$ $\rightarrow$ `addq $8, %rsp` $\rightarrow$ `cmpl %eax, -8(%rsp)` si t alors $\rightarrow$ `jl` lt sinon $\rightarrow$ `jge` lf
$[\![e_1 <= e_2]\!]_{\rho,s,t,lt,lf}$ = $[\![e_1]\!]_{\rho,s}$ $\rightarrow$ `pushq %rax` $[\![e_2]\!]_{\rho,s+8}$ $\rightarrow$ `addq $8, %rsp` $\rightarrow$ `cmpl %eax, -8(%rsp)` si t alors $\rightarrow$ `jle` lt sinon $\rightarrow$ `jg` lf	$[\![e_1 != e_2]\!]_{\rho,s,t,lt,lf}$ = ... $[\![e_1 > e_2]\!]_{\rho,s,t,lt,lf}$ = ... $[\![e_1 >= e_2]\!]_{\rho,s,t,lt,lf}$ = ...
$[\![e]\!]_{\rho,s,t,lt,lf}$ = $[\![e]\!]_{\rho,s}$ $\rightarrow$ `cmpl $0, %eax` si t alors $\rightarrow$ `jne` lt sinon $\rightarrow$ `je` lf	

On remarque une bizarrerie dans la compilation des opérateurs relationnels : la correction de `%rsp` (le `addq $8`) est effectuée avant la comparaison (le `cmpl`), nécessitant d'adresser la pile avec un décalage de −8. Faire cette correction après la comparaison aurait affecté les bits du registre de statut, nous faisant alors perdre l'état de la condition pour le saut conditionnel. En effet, comme la majorité des instructions, `addq` modifie les bits de statut. Nous aurions alors effectué un saut en fonction du résultat de `addq`, ce qui aurait été incorrect. Nous restaurons donc la pile en premier (donc décalons `%rsp` de +8) puis nous enchaînons directement la comparaison et le saut conditionnel.

Comme exposé lors de la présentation des instructions (section 11.2.4), `cmpl s, d` calcule $d - s$ (sans conserver le résultat) pour positionner les bits du registre de statut. Il est donc important de compiler e_2 avant e_1 pour que `cmpl %eax, -8(%rsp)` calcule $e_1 - e_2$.

La compilation des expressions autres que les opérateurs relationnels s'appuie sur le fait que « faux » est représenté par la valeur 0 et que toute autre valeur représente « vrai ». Ainsi, si l'expression calculée est différente de 0, on saute à l'étiquette « vraie », sinon à l'étiquette « fausse ».

Maintenant que nous savons compiler les expressions relationnelles, nous pouvons donner la fonction de compilation des instructions $[\![i]\!]_{\rho,s,le}$ où *le* est l'étiquette de sortie de la fonction courante (celle où sauter en cas de `return`).

<table>
<tr>
<td>$[\![$`while` e `do` i `done`$]\!]_{\rho,s,le}$ =
$entry\_lbl = gen\_label()$
$body\_lbl = gen\_label()$
$exit\_lbl = gen\_label()$
$\rightarrow entry\_lbl$:
$[\![e]\!]_{\rho,s,false,body\_lbl,exit\_lbl}$
$\rightarrow body\_lbl$:
$[\![i]\!]_{\rho,s,le}$
$\rightarrow$ `jmp` $entry\_lbl$
$exit\_lbl$:</td>
<td>$[\![$`if` e `then` i `endif`$]\!]_{\rho,s,le}$ =
$then\_lbl = gen\_label()$
$exit\_lbl = gen\_label()$
$[\![e]\!]_{\rho,s,false,then\_lbl,exit\_lbl}$
$\rightarrow then\_lbl$:
$[\![i]\!]_{\rho,s,le}$
$\rightarrow exit\_lbl$:</td>
</tr>
<tr>
<td>$[\![$`if` e `then` i_1 `else` i_2 `endif`$]\!]_{\rho,s,le}$ =
$then\_lbl = gen\_label()$
$else\_lbl = gen\_label()$
$exit\_lbl = gen\_label()$
$[\![e]\!]_{\rho,s,false,then\_lbl,else\_lbl}$
$\rightarrow then\_lbl$:
$[\![i_1]\!]_{\rho,s,le}$
$\rightarrow else\_lbl$:
$[\![i_2]\!]_{\rho,s,le}$
$\rightarrow exit\_lbl$:</td>
<td>$[\![t[e_1] = e_2]\!]_{\rho,s,le}$ =
$[\![e_2]\!]_{\rho,s}$
$\rightarrow$ `movl` %`eax`, %edi
$[\![e_1]\!]_{\rho,s}$
$\rightarrow$ `movl` %`edi`,$\rho(t)$(%`rbp`, %`rax`, 4)</td>
</tr>
<tr>
<td>$[\![i_1 \ ; \ i_2]\!]_{\rho,s,le}$ =
$[\![i_1]\!]_{\rho,s,le}$
$[\![i_2]\!]_{\rho,s,le}$</td>
<td>$[\![$`return`$]\!]_{\rho,s,le}$ = $\rightarrow$ `jmp` le</td>
</tr>
<tr>
<td>$[\![$`return` $e$$]\!]_{\rho,s,le}$ =
$[\![e]\!]_{\rho,s}$
$\rightarrow$ `jmp` le</td>
<td>$[\![f(e_1, \ldots, e_n)]\!]_{\rho,s,le}$ = $[\![f(e_1, \ldots, e_n)]\!]_{\rho,s}$</td>
</tr>
<tr>
<td colspan="2">$[\![$`print` $(e)$$]\!]_{\rho,s,le}$ =
$s' = (s + \rho(f)) \,\%\, 16$
$corr = 0$ si $s' = 0$ sinon $16 - s'$
si $corr \neq 0$ alors $\rightarrow$ `subq` $corr, %rsp
$[\![e]\!]_{\rho,s}$
$\rightarrow$ `leaq _print_fmt(%rip), %rdi`
$\rightarrow$ `movl %eax, %esi`
$\rightarrow$ `movb $0, %al`
`callq printf`
si $corr \neq 0$ alors $\rightarrow$ `addq` $corr, %rsp</td>
</tr>
</table>

On remarque que la compilation des conditions est initiée avec le booléen *false*. Ainsi on commence toujours par tester la négation de la condition pour sortir de l'instruction si cette négation est fausse. Dans le cas d'un `if-else` l'étiquette « fausse » désigne le code de la partie `else` : si la condition est fausse, on exécute le `else`. Dans le cas d'un `if` cette étiquette désigne le code qui suit le `if` : si la condition est fausse, on exécute la suite du code sans exécuter la partie `then`. Cette méthode de compilation des instructions conditionnelles peut générer des sauts inutiles, ce qui augmente inutilement la taille du code. Si l'on compile :

```
if (i == 0) || ((i > 5) && (i < 10)) then
  i = i + 1 ;
endif
```

le code généré aura la forme :

```
  ... comparaison avec 0
  cmpl %eax, -8(%rsp)
  je lbl_tthen_2
lbl_orl_4:
  ... comparaison avec 5
  cmpl %eax, -8(%rsp)
  jle lbl_texit_3
lbl_andl_5:
   ... comparaison avec 10
  jge lbl_texit_3
  jmp lbl_texit_3
lbl_tthen_2:
   ... code de i = i + 1
lbl_texit_3:
```

où l'on voit apparaître `jge lbl_texit_3` immédiatement suivi de `jmp lbl_texit_3`. Le saut conditionnel est ici inutile. Il est difficile d'éliminer ce genre d'inefficacités dans un compilateur avec une seule passe.

L'instruction `return` est celle qui utilise l'étiquette de fin de fonction courante. À cette étiquette se trouvera le postlude de la fonction, qui restaure `%rbp` et retourne à l'appelant. Notons que si le `return` est la dernière instruction de la fonction, un saut (inutile) va quand même être généré, vers l'instruction suivante.

La compilation de l'instruction `print` est très particulière et s'appuie sur l'appel à la fonction `printf` de la bibliothèque standard C fournie par le système d'exploitation ou le compilateur. Comme `print` n'affiche qu'une seule valeur de type entier signé, le code généré est un cas particulier d'appel général à `printf`. Si nous nous étions pliés à l'ABI System V, cette instruction aurait été compilée comme n'importe quel appel de fonction. Deux arguments sont fournis : le format (fixe, `"%d\n"`, généré de manière aveugle en tant que variable globale dont l'étiquette est `_print_fmt`) passé dans `%rdi` et l'entier passé dans `%esi`. L'utilisation de ces deux registres est dictée par l'ABI System V. Comme imposé par le protocole d'appel des fonctions à nombre d'arguments variable, dans `%al` est passé le nombre d'arguments transmis dans les registres de type SSE (registres permettant de gérer des instructions particulières que nous n'examinerons pas) : ici 0 puisque les deux arguments sont passés par registres entiers. On remarque le mode d'adressage `leaq _print_fmt(%rip)` qui permet d'adresser relativement au pointeur d'instructions `%rip` : l'adresse de l'étiquette est calculée par un déplacement par rapport à celle de l'instruction courante. Cela permet un code plus compact qu'avec un adressage direct.

11.3.3 Compilation des fonction

Nous avons vu la compilation des expressions, des conditions et des instructions. Pour pouvoir être exécutées, ces dernières doivent constituer le corps de fonctions. Par convention, une seule étiquette `main` doit exister et dénote le point d'entrée de la fonction principale du programme.

Une fonction est composée d'éventuels paramètres, d'éventuelles variables globales et d'un corps obligatoire. La compilation d'une fonction doit donc réserver de la mémoire sur la pile pour allouer les variables locales puis générer le code du corps de la fonction. On notera qu'aucun code n'est à produire pour les arguments : c'est le travail de l'appelant de les avoir disposés sur la pile. Néanmoins, il faudra mettre à jour l'environnement ρ afin de savoir où trouver les paramètres lors de la compilation du corps de la fonction.

La fonction $EnvOfPrms\,((x_1,\ldots,x_n),\rho,o)$ réalise l'extension de l'environnement avec les paramètres, où les x_i représentent la liste des paramètres de la fonction, ρ l'environnement courant à étendre et o le décalage courant par rapport à `%rbp`. Initialement, o sera égal à +16 car l'adresse de retour aura été empilée au-dessus des arguments lors de `callq` et le prélude d'une fonction commence toujours par sauvegarder `%rbp` en l'empilant.

$$\begin{array}{lcl} EnvOfPrms\,((),\rho,o) & = & \rho \\ EnvOfPrms\,((x,\ldots),\rho,o) & = & EnvOfPrms\,((\ldots),(x,o)\oplus\rho,o+4) \end{array}$$

La fonction $EnvOfLocals\,(d_1\ldots d_n,\rho,o)$ effectue l'extension de l'environnement avec les variables locales et calcule en même temps la taille globale de ces variables (accumulée dans o initialement égal à 0). Dans l'environnement, on associe à chaque variable locale le déplacement négatif par rapport à `%rbp` pour y accéder. Cette fonction ne compile pas les éventuelles initialisations des variables locales (le cas `var` x s'applique qu'il y ait une expression d'initialisation ou non). Ce travail sera effectué dans un second temps.

$$\begin{array}{lcl} EnvOfLocals\,((),\rho,o) & = & (\rho,o) \\ EnvOfLocals\,((\texttt{var}\ x,\ldots),\rho,o) & = & o'=o+4 \\ & & EnvOfLocals\,((\ldots),(x,-o')\oplus\rho,o') \\ EnvOfLocals\,((\texttt{var}\ x[i],\ldots),\rho,o) & = & o'=o+4\times i \\ & & EnvOfLocals\,((\ldots),(x,-o')\oplus\rho,o') \end{array}$$

La compilation des initialisations de variables locales est donnée par la fonction $[\![d]\!]_{\rho,s}$ qui s'apparente à la compilation d'instructions d'affectation. Notons que l'environnement utilisé contient déjà les liaisons des variables locales afin de pouvoir stocker dans la pile les résultats d'initialisation.

$$\begin{array}{lcl} [\![()]\!]_{\rho,s} & = & \emptyset \\ [\![\texttt{var}\ x,\ldots]\!]_{\rho,s} & = & [\![\ldots]\!]_{\rho,s} \\ [\![\texttt{var}\ x=e,\ldots]\!]_{\rho,s} & = & [\![e]\!]_{\rho,s} \\ & & \rightarrow \texttt{movl}\ \%\texttt{eax},\rho(x)(\%\texttt{rbp}) \\ & & [\![\ldots]\!]_{\rho,s} \\ [\![\texttt{var}\ x[i],\ldots]\!]_{\rho,s} & = & [\![\ldots]\!]_{\rho,s} \end{array}$$

Nous pouvons désormais donner la fonction $[\![f\ldots]\!]_\rho$ qui génère le code d'une définition de fonction. Rappelons que l'environnement ρ mémorise les déplacement des variables locales et des paramètres, mais également la taille globale des arguments d'une fonction (cf. section 11.3.1).

$[\![f(x_1, \ldots, x_n) \texttt{ begin } d_1 \ldots d_n \; i \texttt{ end}]\!]_\rho \quad =$
$\rightarrow f:$
$\rightarrow$ `.cfi_startproc`
`pushq` %`rbp`
`movq` %`rsp`, %`rbp`
$\rho' = EnvOfPrms((x_1, \ldots, x_n), \rho, 16)$
$\rho'' = (f, 4 \times n) \oplus \rho'$
$(\rho''', \delta) = EnvOfLocals(d_1 \ldots d_n, \rho'', 0)$
Si $\delta > 0$ alors $\rightarrow$ `subq` $\$\delta$, %`rsp`
$[\![d_1 \ldots d_n]\!]_{\rho''', 16+\delta}$
$exit\_lbl = gen\_label()$
$[\![i]\!]_{\rho''', 16+\delta, exit\_lbl}$
$exit\_lbl$:
Si $\delta > 0$ alors $\rightarrow$ `addq` $\$\delta$, %`rsp`
`popq` %`rbp`
`retq`
`.cfi_endproc`

Comme précisé au début de la section 11.3, le prélude de la fonction sauvegarde `%rbp`. Lorsque nous étendons l'environnement avec f (pour obtenir ρ''), nous utilisons la notation informelle $4 \times n$ pour représenter un déplacement de « nombre d'arguments fois 4 ». Les directives `.cfi_` sont à destination de l'assembleur (du monde Linux) et lui permettent de détecter les début et fin de *stack frame*. Elles peuvent être ignorées mais sont présentées par soucis de conformité avec le code usuellement produit.

11.3.4 De l'administratif autour

Suivant la syntaxe des assembleurs, une fonction doit être précédée de pseudo-instructions permettant à l'assembleur de savoir dans quel type de zone mémoire le binaire produit doit être logé, quel alignement initial doit être respecté, etc. Ainsi, sur MacOS, avec la chaîne de compilation basée sur clang, le prélude d'une fonction `foo` serait :

```
  .section __TEXT,__text,regular,pure_instructions
  .globl  _foo
  .p2align 4, 0x90
```

où l'on remarque que les noms de fonction sont préfixés par un _. Sur Linux avec la chaîne de compilation basée sur gcc, l'en-tête de cette fonction serait :

```
  .text
  .globl foo
  .p2align 4, 0x90
  .type foo, @function
```

Pour compiler l'instruction `print`, nous avons utilisé une variable globale pour représenter le format `"%d\n"`. Il faut donc que cette variable soit définie dans le code généré. Cela se fait en définissant explicitement une nouvelle *section*. Sous Mac OS cette section est :

```
  .section __TEXT,__cstring,cstring_literals
_print_fmt: .asciz "%d\n"
```

alors que sous Linux elle contient :

```
  .section .rodata
_print_fmt: .string "%d\n"
```

11.4 Ce que l'on n'a pas étudié

Le compilateur que nous avons réalisé reste très naïf. D'une part le langage compilé est modeste (pas de pointeurs, pas de structures de données, pas d'entiers de différentes tailles, pas de flottants, pas de variables globales, etc.). D'autre part, la présence d'une seule passe empêche la production de code efficace exploitant tous les registres, des instructions spécifiques, l'élimination d'instructions inutiles, etc. Un compilateur plus évolué devrait comporter de nombreuses étapes supplémentaires que nous décrivons succinctement maintenant en nous inspirant de l'architecture de `CompCERT` [1] (simplifiée).

À l'issue de l'analyse lexicale et syntaxique un AST est créé. La première passe consiste à effectuer une analyse de portée (*scoping*) pour déterminer à quelle définition chaque occurrence d'identificateur se rapporte. Cela permet d'une part de différencier les identificateurs homonymes et d'autre part de déterminer si une variable correspond à un accès global ou local. À l'issue du scoping, deux variables de même nom mais liées par une définition différente seront annotées par une étiquette (en pratique, un entier) différente. Ainsi, le programme ci-dessous (dans une version plus évoluée du langage) serait annoté de la façon suivante :

```
var x1 = 5 ;

f (x2, y3)
begin
var i4 = x2 + y3 ;
while i4 < 10 do
  var x5 = x2 + 1 ;
  y3 = x5 − i4 ;
done
end
```

Une passe de typage est ensuite nécessaire, comme nous l'avons vu dans le chapitre 7 afin de rejeter les programmes auxquels il est impossible d'attribuer une sémantique d'exécution. Cette analyse permet également de déterminer la taille des structures de données utilisées et les conversions implicites à réaliser.

À partir de l'AST typé, une représentation intermédiaire (RTL pour *Register Transfer Langage*) est créée sous forme d'un graphe (*de flot de contrôle* ou *CFG*) dans lequel les instructions du microprocesseur ont été explicitées ainsi que le flot de contrôle du programme. Ce dernier perd ainsi sa structure d'arbre syntaxique pour devenir un graphe dont les nœuds sont des instructions assembleur et les arcs représentent les chemins d'exécution entre les instructions. Toutes les constructions de contrôle (boucles, conditionnelles et même le `goto`) donnent naissance à des arcs du graphe. Dans cette représentation les instructions utilisent des *pseudo-registres* en nombre illimité et non les registres réels du microprocesseur. Notons que la transformation en graphe et la *sélection d'instructions* peuvent être réalisées en deux passes différentes.

Il faut ensuite rendre explicites les conventions d'appel imposée par l'ABI que l'on respecte. Cela est réalisé lors de la transformation du code en une nouvelle représentation intermédiaire (ERTL pour *Explicit Register Transfer Language*) qui est encore un graphe. Les registres réels du processeur commencent à faire leur apparition localement (au niveau des appels et d'instructions particulières comme la division, le calcul du modulo) et leur sauvegarde / restauration (pour survivre aux appels de fonction) est déterminée.

1. http://compcert.inria.fr

La passe suivante doit procéder à *l'allocation de registres*, c'est-à-dire le remplacement des pseudo-registres par les registres réels du microprocesseur. Lorsque l'on rencontre un manque de registre, on en sauvegarde un sur la pile pour le libérer temporairement. L'allocation est effectuée en travaillant sur le graphe pour déterminer la durée de vie des registres (quelle est la portée durant laquelle la valeur contenue dans un registre est utile), puis construire un graphe d'interférence (pour déterminer quels pseudo-registres ne peuvent être remplacés par un même registre au même instant) et enfin réellement attribuer les registres par coloriage du graphe d'interférence.

Une dernière représentation intermédiaire est synthétisée (LTL pour *Location Transfer Language*) dans laquelle les pseudo-registres sont remplacés par les registres et les emplacements de pile déterminés lors de l'allocation. Les instructions de transfert entre la pile et les registres sont insérées et la sauvegarde / restauration des registres à la charge de la fonction appelée sont explicitement insérées dans le graphe.

Finalement, le LTL est transformé en source assembleur lors d'un parcours du graphe. Il reste ensuite à transformer ce source en code machine (binaire), ce qui est le travail du logiciel d'assemblage que nous n'abordons pas ici.

Durant les différentes passes de compilation de nombreuses optimisations peuvent être appliquées pour produire *in fine* du code plus efficace ou plus compact. Certaines de ces optimisations sont compliquées à implanter et peuvent augmenter le temps de compilation de manière significative.

Nous voyons qu'un compilateur réaliste nécessite des techniques plus compliquées que celles que nous avons mises en œuvre dans ce chapitre. En 2012, le nombre de lignes de code du compilateur GCC (qui supporte plusieurs langages) était estimé à 7.3 millions.

Chapitre 12

Vers du code assembleur plus efficace

Dans le chapitre 11, nous avons vu comment générer du code assembleur pour l'architecture X86-64. Néanmoins, comme le titre du chapitre le suggérerait, il s'agissait d'une compilation très naïve puisqu'elle n'exploitait qu'un seul registre en faisant un usage intensif de la pile. À la fin de ce même chapitre, ont été décrites les nombreuses étapes nécessaires pour l'obtention d'un compilateur plus réaliste, exploitant tous les registres du processeur et effectuant quelques optimisations.

Ce nouveau chapitre va permettre de rentrer dans le détail de la création d'un compilateur plus réaliste, sans toutefois rivaliser avec l'efficacité et les subtilités des compilateurs représentant l'état de l'art actuel. La lecture du chapitre 11 est nécessaire pour comprendre ce nouveau chapitre puisqu'un certain nombre de concepts vont être grandement réutilisés (instructions assembleur, structure de la pile, traduction des conditions, alignement sur 16 octets aux sites d'appel, etc.).

12.1 Point de départ

Nous considérons un langage très similaire à celui du chapitre 11, dans lequel nous retirons la gestion des tableaux. Notre langage ne permet donc de manipuler que des entiers de taille fixe. Les prototypes de fonctions sont également omis car ils n'ont d'utilité que pour la phase de typage ou pour la génération de code mais en présence de types de données plus variés que seulement des entiers de taille fixe, ou pour des optimisations.

$$
\begin{array}{rcl}
e & ::= & i \mid x \mid x\ (e_1, \ldots, e_n) \\
& \mid & e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \mid -e \\
& \mid & e_1 == e_2 \mid e_1 != e_2 \mid e_1 < e_2 \mid e_1 <= e_2 \mid e_1 > e_2 \mid e_1 >= e_2 \\
i & ::= & \texttt{while}\ e\ \texttt{do}\ i\ \texttt{done} \mid \texttt{if}\ e\ \texttt{then}\ i_1\ \texttt{else}\ i_2\ \texttt{endif} \mid \texttt{if}\ e\ \texttt{then}\ i\ \texttt{endif} \\
& \mid & i\ ;\ i \mid x = e \mid \texttt{return}\ e \mid \texttt{return} \mid x\ (e_1, \ldots, e_n) \mid \texttt{print}\ (e) \\
d & := & \texttt{var}\ x \mid \texttt{var}\ x = e \\
f & := & x\ (x_1, \ldots x_n)\ \texttt{begin}\ d_1 \ldots d_n\ i\ \texttt{end} \\
p & := & f_1 \ldots f_n
\end{array}
$$

Comme décrit précédemment, une passe d'analyse de portée puis une passe de typage sont préalables à la compilation. Nous les considérons déjà faites et partons d'un AST où chaque expression est bien typée (et annotée par son type si besoin). Notez qu'au lieu d'utiliser des entiers standard d'OCaml (dont 1 bit est amputé pour que le GC puisse fonctionner), nous utilisons des `Int64.t` qui permettent de représenter des entiers sur 64 bits. Cela est important dans le cas où le source à compiler comporterait des constantes numériques requérant effectivement 64 bits.

```
type var_t = string

type expr_t =
  | E_int of Int64.t | E_ident of var_t | E_app of (var_t * expr_t list)
  | E_unop of (string * expr_t) | E_binop of (string * expr_t * expr_t)

and instr_t =
  | I_while of (expr_t * instr_t) | I_if of (expr_t * instr_t * instr_t option)
  | I_assign of (var_t * expr_t) | I_seq of (instr_t * instr_t) | I_return of expr_t option
  | I_app of (var_t * expr_t list) | I_print of expr_t

type var_decl_t = (var_t * (expr_t option))

type fun_def_t = {
  fdef_name : var_t ;
  fdef_params : var_t list ;
  fdef_vars : var_decl_t list ;
  fdef_body : instr_t
}
```

12.2 Plan d'action

Une fois de plus, nous allons nous inspirer de l'architecture de CompCERT [1] en la simplifiant (beaucoup) toutefois. Le schéma 12.1 représente l'enchaînement des différentes passes que nous allons décrire plus en détail et leurs principaux effets.

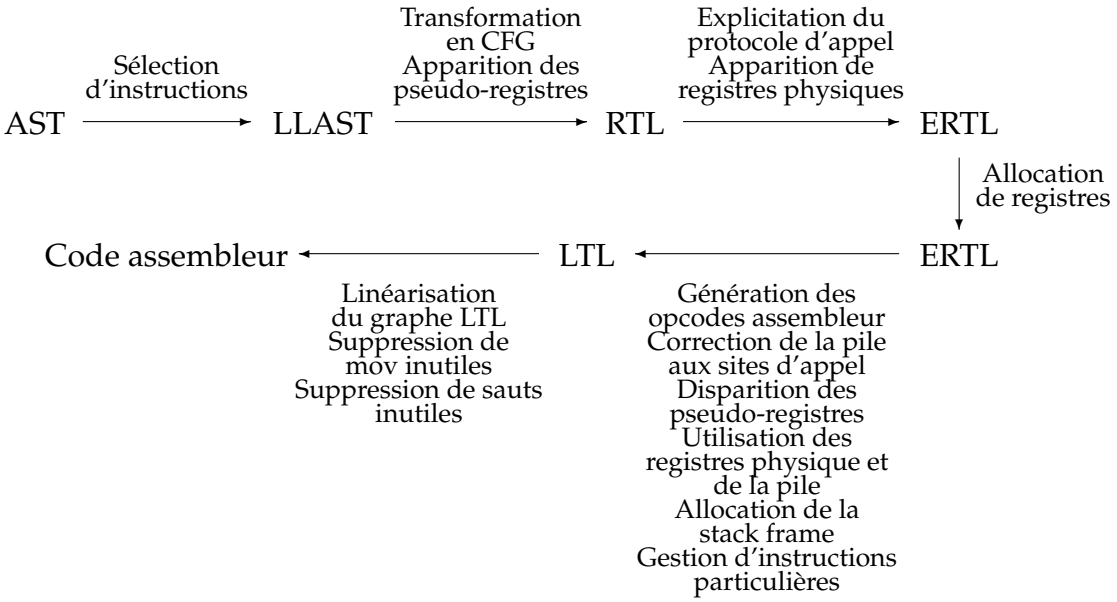


FIGURE 12.1 – Enchaînement des passes de compilation

1. http://compcert.inria.fr

12.3 Vers LLAST

La première passe va consister à traduire les opérateurs des expressions en pseudo-instructions assembleur. Cette passe est volontairement séparée de la transformation de l'AST en graphe pour permettre de réaliser quelques optimisations. La représentation obtenue reste un arbre très similaire à l'AST initial que nous appelons LLAST (pour *Low Level AST*).

Nous devons donc décider quelles futures instructions assembleur nous souhaitons utiliser. S'il vient immédiatement en tête les `mov`, `add` et autres opérations arithmétiques, d'autres s'avèrent souhaitables. En effet, lorsque nous devons compiler une instruction `i := i + 5`, nous ne voulons pas charger 5 dans un registre pour effectuer ensuite l'addition : `mov $5,reg1 ; add reg1,reg2`. Nous voulons à la place faire une addition avec une constante immédiate : `add $5,reg`.

12.3.1 Structure de LLAST

LLAST a donc la forme suivante dans laquelle l'on retrouve la structure des instructions et des expressions de l'AST initial, avec uniquement les opérateurs traduits dans le jeu de pseudo-instructions choisi. Notons qu'il n'y a pas d'instructions de division, de modulo, de test avec des constantes . . . par fainéantise ☺ (et pour raccourcir le code).

```
type unop_t = U_neg of expr_t  (* − e *)

and binop_t =
  | B_addi of (Int64.t * expr_t)  (* i + e *)     | B_add of (expr_t * expr_t)   (* e1 + e2 *)
  | B_subi of (Int64.t * expr_t)  (* e − i *)     | B_sub of (expr_t * expr_t)   (* e1 − e2 *)
  | B_multi of (Int64.t * expr_t) (* i * e *)     | B_mult of (expr_t * expr_t)  (* e1 * e2 *)
  | B_div of (expr_t * expr_t)    (* e1 / e2 *)   | B_mod of (expr_t * expr_t)   (* e1 % e2 *)
  | B_land of (expr_t * expr_t)   (* e1 && e2 *)  | B_lor of (expr_t * expr_t)   (* e1 || e2 *)
  | B_eq of (expr_t * expr_t)     (* e1 == e2 *)  | B_neq of (expr_t * expr_t)  (* e1 != e2 *)
  | B_l of (expr_t * expr_t)      (* e1 < e2 *)   | B_le of (expr_t * expr_t)   (* e1 ≤ e2 *)
  | B_g of (expr_t * expr_t)      (* e1 > e2 *)   | B_ge of (expr_t * expr_t)   (* e1 ≥ e2 *)

and expr_t =
  | E_int of Int64.t | E_ident of Ast.var_t | E_app of (Ast.var_t * expr_t list)
  | E_unop of unop_t | E_binop of binop_t

and instr_t =
  | I_while of (expr_t * instr_t) | I_if of (expr_t * instr_t * instr_t option)
  | I_assign of (Ast.var_t * expr_t) | I_seq of (instr_t * instr_t)
  | I_return of expr_t option | I_app of (Ast.var_t * expr_t list) | I_print of expr_t

type var_decl_t = (Ast.var_t * (expr_t option))

type fun_def_t = {
  fdef_name : Ast.var_t ;
  fdef_params : Ast.param_decl_t list ;
  fdef_vars : var_decl_t list ;
  fdef_body : instr_t
}
```

12.3.2 Traduction vers LLAST

La traduction d'`AST` vers `LLAST`, notée $[\![]\!]$ (pour en pas surcharger la notation, on utilise le même symbole pour la traduction des expressions et des instructions) est réalisée par un simple parcours de l'arbre. Dans les règles du tableau ci-dessous, implicitement les constructeurs des membres gauches des égalités appartiennent à `AST` et ceux de droite à `LLAST`. Pour les expressions arithmétiques, au lieu d'émettre directement l'instruction associée, on utilise des fonctions dédiées qui vont se charger de faire les optimisations portant sur les constantes. De telles fonctions sont appelées *smart constructors*. Ainsi, lorsque l'on rencontre une expression `x + 3 + 1` on souhaite générer `B_addi (4, (E_ident "x"))`. Nous définirons ainsi les fonctions $[\![e]\!]_{neg}$, $[\![e_1\ e_2]\!]_+$, $[\![e_1\ e_2]\!]_-$, etc. dédiées à ces optimisations. Ces fonctions se comportent de manière similaire avec une petite différence pour les opérateurs non symétriques. Par exemple, dans le cas de la soustraction, puisque `B_subi (i, e)` calcule $e-i$, si c'est la première opérande qui est une constante (calcul de $i-e$), aucune simplification n'est possible en utilisant `B_subi`.

$[\![$`E_int` $i]\!] =$ `E_int` i	$[\![$`E_ident` $x]\!] =$ `E_ident` x
$[\![$`E_app` $(f\ [e_1; \ldots; e_n])]\!] =$ `E_app` $(f, [[\![e_1]\!]; \ldots; [\![e_n]\!]])$	$[\![$`E_unop` $(" - ", e)]\!] = [\![e]\!]_{neg}$
$[\![$`E_binop` $(" + ", e_1, e_2)]\!] = [\![e_1\ e_2]\!]_+$	$[\![$`E_binop` $(" - ", e_1, e_2)]\!] = [\![e_1\ e_2]\!]_-$
...	
$[\![$`E_binop` $("\&\&", e_1, e_2)]\!] =$ `E_binop` $($`B_land` $([\![e_1]\!], [\![e_2]\!]))$	$[\![$`E_binop` $("\|\|", e_1, e_2)]\!] =$ `E_binop` $($`B_lor` $([\![e_1]\!], [\![e_2]\!]))$
$[\![$`E_binop` $(" ==", e_1, e_2)]\!] =$ `E_binop` $($`B_eq` $([\![e_1]\!], [\![e_2]\!]))$	$[\![$`E_binop` $("! =", e_1, e_2)]\!] =$ `E_binop` $($`B_neq` $([\![e_1]\!], [\![e_2]\!]))$
...	
$[\![$`I_while` $(e, i)]\!] =$ `I_while` $([\![e]\!], [\![i]\!])$	$[\![$`I_if` $(e, i,$ `None`$)]\!] =$ `I_if` $([\![e]\!], [\![i]\!],$ `None`$)$
...	

$[\![e]\!]_{neg} =$
`match` $[\![e]\!]$ `with`
`E_int` $i \rightarrow$ `E_int` $(-i)$
`E_unop` (`U_neg` e) $\rightarrow e$
$e' \rightarrow$ `E_unop` (`U_neg` e')

$[\![e_1\ e_2]\!]_+ =$
`match` $[\![e_1]\!], [\![e_2]\!]$ `with`
(`E_int` i_1), (`E_int` i_2) $\rightarrow$ `E_binop` (`E_int` $(i_1 + i_2)$)
(`E_int` i), $e'_2 \rightarrow$ `E_binop` (`B_addi` (i, e'_2))
e'_1, (`E_int` i) $\rightarrow$ `E_binop` (`B_addi` (i, e'_1))
$e'_1, e'_2 \rightarrow$ `E_binop` (`B_add` (e'_1, e'_2))

$[\![e_1\ e_2]\!]_- =$
`match` $[\![e_1]\!], [\![e_2]\!]$ `with`
(`E_int` i_1), (`E_int` i_2) $\rightarrow$ `E_binop` (`E_int` $(i_1 - i_2)$)
e'_1, (`E_int` i) $\rightarrow$ `E_binop` (`B_subi` (i, e'_1))
$e'_1, e'_2 \rightarrow$ `E_binop` (`B_sub` (e'_1, e'_2))

La traduction d'une définition de fonction ne présente aucune difficulté et se contente de traduire le corps pour reconstituer une définition `LLAST`.

12.3.3 Implantation en Ocaml

L'implémentation en Ocaml de cette traduction est très simple comme le montre l'extrait suivant, donnant les grandes lignes du code.

```
let rec do_add e1 e2 =
  let e1' = do_expr e1 in
  let e2' = do_expr e2 in
  match (e1', e2') with
  | ((LLAst.E_int i1), (LLAst.E_int i2)) → LLAst.E_int (Int64.add i1 i2)
```

```
  | ((LLAst.E_int i1), _) → LLAst.E_binop (LLAst.B_addi (i1, e2'))
  | (_, (LLAst.E_int i2)) → LLAst.E_binop (LLAst.B_addi (i2, e1'))
  | (_, _) → LLAst.E_binop (LLAst.B_add (e1', e2'))

and do_sub e1 e2 =
  let e1' = do_expr e1 in
  let e2' = do_expr e2 in
  match (e1', e2') with
  | ((LLAst.E_int i1), (LLAst.E_int i2)) → LLAst.E_int (Int64.sub i1 i2)
  | (_, (LLAst.E_int i2)) → LLAst.E_binop (LLAst.B_subi (i2, e1'))
  | (_, _) → LLAst.E_binop (LLAst.B_sub (e1', e2'))

and do_binop o_name e1 e2 =
  match o_name with
  | "+" → do_add e1 e2 | "-" → do_sub e1 e2 | "*" → do_mul e1 e2
  | "/" → do_div e1 e2 | "%" → do_mod e1 e2
  | "||" → LLAst.E_binop (LLAst.B_lor ((do_expr e1), (do_expr e2)))
  | "&&" → LLAst.E_binop (LLAst.B_land ((do_expr e1), (do_expr e2)))
   ...

and do_expr expr =
  match expr with
    | Ast.E_int i → LLAst.E_int i
    | Ast.E_ident var →LLAst.E_ident var
    | Ast.E_app (f_name, args) → LLAst.E_app (f_name, (List.map do_expr args))
    | Ast.E_unop (o_name, e) →do_unop o_name e
    | Ast.E_binop (o_name, e1, e2) → do_binop o_name e1 e2

let rec do_instr instr =
  let desc' =
    (match instr with
    | Ast.I_while (e, i) → LLAst.I_while ((do_expr e), (do_instr i))
    | Ast.I_if (e, i1, i2opt) →
        let i2opt' =
          (match i2opt with None →None | Some i → Some (do_instr i)) in
        LLAst.I_if ((do_expr e), (do_instr i1), i2opt')
    | Ast.I_assign (var, e) → LLAst.I_assign (var, (do_expr e))
     ...

let do_fun_def fun_def =
  { LLAst.fdef_name = fun_def.Ast.fdef_name ;
    LLAst.fdef_params = fun_def.Ast.fdef_params ;
    LLAst.fdef_vars = List.map do_var_decl fun_def.Ast.fdef_vars ;
    LLAst.fdef_body = do_instr fun_def.Ast.fdef_body }
```

12.4 Vers RTL

Une fois la sélection d'instructions effectuée, l'AST bas niveau va subir une transformation radicale pour obtenir une représentation intermédiaire sous forme d'un graphe de flot de contrôle : le *Register Transfer Langage* (RTL). Les nœuds de ce graphe représentent les instructions du microprocesseur et les arcs représentent les chemins d'exécution entre les instructions.

Ainsi, toutes les constructions de contrôle vont donner naissance à des arcs dans ce graphe : la conditionnelle, la boucle et la séquence. L'évaluation des expressions va également produire des arcs

puisque lorsque l'on calcule $e_1 + e_2$, il faut calculer e_1 puis calculer e_2 (ou dans l'ordre inverse, c'est un choix de compilation) puis calculer l'addition. Donc, dans le graphe résultat, le dernier nœud du code calculant e_1 aura pour enfant le premier de celui calculant e_2. Ainsi, durant la compilation vers RTL, on explicite l'ordre d'évaluation des expressions.

La compilation des conditionnelles et des boucles est l'une des parties les plus subtiles et s'inspirera de ce que nous avons fait dans le chapitre 11. Nous verrons également comment compiler les expressions logiques et relationnelles utilisées en dehors des conditions (par exemple `x = (y < 0) || (y == 7)`).

Les (pseudo) instructions utilisées ne seront pas forcément exactement celles de l'assembleur final, mais elles s'en rapprocheront énormément (par exemple, la division et le modulo devront être transformées dans une passe suivante). Ces instructions ne vont pas opérer sur les registres physiques du processeur mais sur des *pseudo-registres* disponibles en nombre illimité. Nous allons bien entendu possiblement obtenir du code terriblement inefficace, mais les passes suivantes se chargeront d'optimiser en allouant des registres réels, en supprimant des instructions inutiles ou redondantes.

12.4.1 Structure de graphe

Il existe plusieurs manières d'implanter une structure de graphe. La première consiste à représenter les nœuds par des enregistrements et à exploiter le partage physique pour garantir l'unicité d'un nœud représentant une instruction donnée. Chaque nœud comporte alors la liste de ses enfants (ou successeurs). Notons que dans une telle représentation il est nécessaire d'avoir un champ `id` permettant d'identifier de façon unique les nœuds pour les opérations de comparaisons nécessaires pour créer des `Map` ou des `Set` OCaml stockant des nœuds (sinon la comparaison structurelle va boucler si le graphe est cyclique).

```
type asm_instr_t = ...

type node_t = {
  n_id : int ;
  n_asm : asm_instr_t ;
  mutable n_children : node_t list
}
```

Nous n'allons pas utiliser cette structure qui implique des manipulations compliquées de « pointeurs » et tend à complexifier l'implantation. À la place nous représentons un graphe comme une référence sur une `Map` OCaml associant des « étiquettes » à des instructions (référence car le graphe est modifié en place au cours des ajouts de nœuds, or les `Map` OCaml ne sont pas mutables). Ces étiquettes ne seront pas présentes dans l'assembleur final généré : elles représentent juste des identificateurs de nœuds (un moyen pour désigner un nœud particulier). Les arcs du graphe seront représentés dans les instructions : chaque instruction contiendra l'étiquette de son ou ses successeurs.

On se définit donc un module `Graph` représentant les étiquettes par des entiers, fournissant une fonction `new_graph_label` permettant de générer une nouvelle étiquette et une fonction `add_node` permettant d'ajouter un nouveau nœud et retourner son étiquette. La fonction `add_node_with_label` permet d'ajouter un nouveau nœud mais avec une étiquette donnée et sera utile uniquement pour la compilation des boucles.

```
type graph_label_t = int

let new_graph_label =
```

```
  let count = ref 0 in
  fun () → incr count ; !count

let string_of_label l = string_of_int l

module GLabelMod = struct type t = graph_label_t let compare x y = compare x y end
module G = Map.Make(GLabelMod)

let add_node graph instr =
  let graph_label = new_graph_label () in
  graph := G.add graph_label instr !graph ;
  graph_label

let add_node_with_label graph graph_label instr = graph := G.add graph_label instr !graph
```

Puisque les `Map` sont polymorphes, nous obtenons une structure de graphe également polymorphe, des α `graph` en quelque sorte, où α représente le type des instructions. Cela est très intéressant puisque les passes successives de compilation vont produire des graphes avec des types d'instructions différents. Nous pourrons donc réutiliser cette structure de graphe. Mieux encore, les graphes obtenus lors de ces passes sont des modifications (des ajouts de nœuds) du graphe de la passe précédente. Il sera donc possible de conserver les étiquettes d'un graphe sur l'autre pour représenter les instructions correspondantes dans les deux graphes. Cela simplifie grandement l'implémentation par rapport à une représentation à base d'enregistrements où il aurait fallu conserver trace de quel nœud du graphe d'une passe n correspond à quel nœud de la passe $n-1$.

12.4.2 Structure de RTL

Le RTL est donc une structure de graphe dont les nœuds contiennent les (pseudo) instructions. On retrouve des instructions semblables à celles vues dans le chapitre 11.2.4. En particulier, conformément aux instructions réelles du X86-64, celles de RTL auront majoritairement 2 opérandes. Les instructions arithmétiques `XXX` s, d entre un registre source s et un registre destination d auront pour sémantique $d \leftarrow d$ `XXX` s.

- `Movi` / `Mov` : transfert d'une constante dans un pseudo-registre ou entre deux pseudo-registres.
- `Addi`, `Add` : addition d'une constante à un registre ou de deux registres.
- `Subi`, `Sub` : soustraction d'une constante à un registre ou de deux registres.
- `Muli`, `Mul` : multiplication d'une constante par un registre ou de deux registres.
- `Div`, `Mod` : division entière et modulo entre deux registres.
- `Neg` : moins unaire d'un registre.
- `Cmpi`, `Cmp` : comparaison d'un registre à une constante ou entre deux registres.
- `Jmp` : saut inconditionnel.
- `J`*cc* : saut conditionnel.
- `Set`*cc* : met 0 ou 1 dans le registre si la condition *cc* est fausse ou vraie.
- `Label` : pseudo-instruction permettant de générer une étiquette dans le code assembleur, avec un éventuel commentaire. Attention, nous prendrons soin de toujours faire en sorte qu'un saut aboutisse sur un nœud `Label`. Cela simplifiera la question de quelles étiquettes devront rester dans l'assembleur final.
- `Call` : pseudo-instruction permettant d'effectuer un appel de fonction. Elle sera transformée de manière importante dans les passes suivantes.
- `Print` : pseudo-instruction permettant d'afficher un entier (finira comme un appel *ad hoc* à `printf`). Elle contient une chaîne représentant le nom de la fonction `printf` (point discuté en fin de section).
- `Ret` : retour de fonction.

Les codes de condition *cc* sont les mêmes que dans le chapitre 11.2.4 : `E` (*equal*), `Ne` (*not equal*), `G` (*greater*), `Ge` (*greater or equal*), `L` (*lower*), `Le` (*lower or equal*).

Chaque paramètre formel d'une fonction sera associé avec un pseudo-registre qui représentera l'endroit où trouver la valeur de ce paramètre. De même, un pseudo-registre sera dédié au stockage de la valeur retournée par la fonction. Ainsi, une fonction sera représentée par :

- son nom,
- la liste des pseudo-registres contenant ses paramètres,
- son pseudo-registre de retour,
- la liste de ses déclarations de variables locales,
- son graphe (qui représente la traduction de son corps),
- l'étiquette du nœud du graphe contenant la première instruction de la fonction (point d'entrée),
- l'étiquette du nœud du graphe contenant la dernière instruction de la fonction (là où sauter lorsque l'on rencontre une instruction `return` du langage source).

L'implantation en OCaml est donc la suivante, dans laquelle chaque constructeur représentant une instruction comporte un argument de type `Graph.graph_label_t` représentant l'étiquette de l'instruction suivante.

```
type cc_t = E | Ne | G | Ge | L | Le

type asm_instr_t =
  | Movi of (Int64.t * register_t * Graph.graph_label_t)
  | Mov of (register_t * register_t * Graph.graph_label_t)
  | Addi of (Int64.t * register_t * Graph.graph_label_t)
  | Add of (register_t * register_t * Graph.graph_label_t)
  | Subi of (Int64.t * register_t * Graph.graph_label_t)
  | Sub of (register_t * register_t * Graph.graph_label_t)
  | Muli of (Int64.t * register_t * Graph.graph_label_t)
  | Mul of (register_t * register_t * Graph.graph_label_t)
  | Div of (register_t * register_t * Graph.graph_label_t)
  | Mod of (register_t * register_t * Graph.graph_label_t)
  | Neg of (register_t * Graph.graph_label_t)
  | Cmp of (register_t * register_t * Graph.graph_label_t)
  | Cmpi of (Int64.t * register_t * Graph.graph_label_t)
  | Jmp of  Graph.graph_label_t
  | Jcc of (cc_t * Graph.graph_label_t * Graph.graph_label_t) (* Étiquettes ok / ko. *)
  | Setcc of (cc_t * register_t * Graph.graph_label_t)
  | Label of (string * string * Graph.graph_label_t)
  | Call of (register_t * Ast.var_t * (register_t list) * Graph.graph_label_t)
  | Print of (Ast.var_t * register_t * Graph.graph_label_t)
  | Ret

type fun_def_t = {
  fdef_name : Ast.var_t ;
  fdef_params : register_t list ;
  fdef_ret_reg : register_t ;
  fdef_vars : LLAst.var_decl_t list ;
  fdef_graph : asm_instr_t Graph.G.t ;
  fdef_entry : Graph.graph_label_t ;
  fdef_postlude : Graph.graph_label_t
}
```

On remarque que l'instruction `Ret` n'a pas d'étiquette d'instruction suivante, ce qui est bien normal puisqu'elle provoque la fin du code de la fonction courante. À l'inverse `Jcc` en possède deux : celle

dans le cas où le test est vrai et celle dans le cas où il est faux. L'instruction `Call` enregistre le registre dans lequel stocker le résultat de l'appel, le nom de la fonction appelée, la liste des pseudo-registres contenant les arguments de l'appel et, bien entendu, l'étiquette de l'instruction suivant cet appel. La pseudo-instruction `Print` n'utilise qu'un registre qui contient la seule valeur à afficher. Si notre langage gérait les chaînes de caractères, nous aurions pu nous appuyer sur des appels normaux à la fonction `printf` fournie par la bibliothèque standard du système d'exploitation.

12.4.3 Construction du graphe RTL

La construction du graphe est réalisée par un parcours de l'arbre LLAST. Deux choix sont possibles pour construire le graphe. La première solution consiste à créer le nœud d'une instruction puis de créer ses successeurs. On construit ainsi le graphe en partant du début de la fonction. Cette manière de procéder tend à aboutir à une implantation verbeuse nécessitant parfois de se rappeler du premier nœud généré pour compiler une expression, parfois de se rappeler du dernier, parfois des deux. Nous allons choisir l'approche inverse : construire le graphe en commençant par la fin de la fonction. Ainsi, récursivement nous obtiendrons le premier nœud du sous-graphe représentant « la suite du code ».

Nous allons définir un ensemble de fonctions de compilation, mutuellement récursives, pour traiter les expressions, les instructions et les conditions. Toutes ces fonctions prennent implicitement le graphe courant en argument puisqu'elles vont y ajouter des nœuds. Pour ne pas surcharger plus que nécessaire la notation, cet argument sera implicite dans les règles qui vont suivre.

Traduction des expressions

La fonction ${}^{e}[\![e]\!]_{\rho,dr,nl}$ donne la compilation d'une expression e. Elle s'effectue dans un environnement ρ, qui associe les identificateurs (noms de paramètres formels et variables locales) à leurs pseudo-registres. Le pseudo-registre dr (pour *destination register*) est celui dans lequel le résultat de l'expression devra être transféré. L'étiquette nl (pour *next label*) indique l'instruction suivante dans le graphe (puisque nous construisons le graphe en partant de la fin de la fonction). La compilation utilise la fonction `add_node` du module `Graph` introduite dans la section 12.4.1 (dont on ometra le paramètre représentant le graphe). La fonction ${}^{e}[\![e]\!]_{\rho,dr,nl}$ est définie mutuellement récursivement avec ${}^{b}[\![b]\!]_{\rho,dr,nl}$ qui permet de compiler les opérateurs binaires, ainsi que quelques autres fonctions que nous détaillerons au fur et à mesure.

${}^{e}[\![\texttt{E\_int}\ i]\!]_{\rho,dr,nl} = \texttt{add\_node}\ (\texttt{Movi}\ (i, dr, nl))$
${}^{e}[\![\texttt{E\_ident}\ x]\!]_{\rho,dr,nl} = \texttt{add\_node}\ (\texttt{Mov}\ (\rho(x), dr, nl))$
${}^{e}[\![\texttt{E\_app}\ (f, [e_1; \ldots; e_n])]\!]_{\rho,dr,nl} =$ Soit $[r_1; \ldots; r_n]$ une liste de n nouveaux pseudo-registres, ${}^{a}[\![[e_1; \ldots; e_n], [r_1; \ldots; r_n]]\!]_{\rho,}\texttt{add\_node}\ (\texttt{Call}\ (dr, f, [r_1; \ldots; r_n], nl))$
${}^{e}[\![\texttt{E\_unop}\ (\texttt{U\_neg}\ e)]\!]_{\rho,dr,nl} = {}^{e}[\![e]\!]_{\rho,dr,}\texttt{add\_node}\ (\texttt{Neg}\ (dr, nl))$
${}^{e}[\![\texttt{E\_binop}\ b]\!]_{\rho,dr,nl} = {}^{b}[\![b]\!]_{\rho,dr,nl}$

FIGURE 12.2 – Traduction LLAST → RTL des expressions

Le cas des constantes ou des identificateurs permet d'exploiter directement le registre dr en évitant ainsi un `Mov` supplémentaire dans un registre intermédiaire. On remarque bien la construction du graphe depuis la fin de la fonction puisque l'étiquette du nœud « suivant », nl qui dénote « le code à éxécuter ensuite » vient bien comme successeur de `Mov`.

Le cas de l'appel de fonction commence par générer autant de pseudo-registres qu'il y a d'arguments effectifs. La fonction ${}^{a}[\![\ ]\!]$ ci-dessous se chargera d'effectuer les transferts des arguments (après la compilation de leurs expressions) dans ces registres, la suite du code à produire sera donc une instruction `Call`, ayant elle-même pour suite « le reste du programme » (nl).

Les opérateurs binaires sont gérés au moyen de la fonction ${}^{b}[\![\ ]\!]$ qui sera présentée un peu plus loin, traitant différemment les opérateurs arithmétiques et les opérareurs relationnels.

$$
\begin{array}{lcl}
{}^{a}[\![[\,], [\,]]\!]_{\rho,nl} & = & nl \\
{}^{a}[\![[e_1; e_2 \ldots], [dr_1; dr_2 \ldots]]\!]_{\rho,nl} & = & {}^{e}[\![e_1]\!]_{\rho,dr_1,{}^{a}[\![[e_2 \ldots], [dr_2 \ldots]]\!]_{\rho,nl}} \\
{}^{a}[\![[e_1; \ldots], [\,]]\!]_{\rho,nl} & = & \text{Impossible} \\
{}^{a}[\![[\,], [dr_1; \ldots]]\!]_{\rho,nl} & = & \text{Impossible}
\end{array}
$$

FIGURE 12.3 – Traduction LLAST → RTL des arguments d'appel de fonction

La traduction des arguments effectifs d'un appel compile itérativement leurs expressions en utilisant comme registre destination celui qui a été créé pour chaque argument dans ${}^{e}[\![\ ]\!]$. Dans cette forme de récursion, on compile les arguments en commençant par la fin, l'étiquette de graphe ainsi générée servant de *nl* au code généré pour l'argument précédent dans la liste. Structurellement, il est impossible que la liste des expressions arguments et celle des pseudo-registres destination ne soient pas de même longueur.

Traduction des opérateurs binaires

La compilation des opérateurs binaires se scinde en deux schémas différents : celle des opérateurs arithmétiques et celle des opérateurs relationnels (« comparaisons ») utilisés en dehors des conditions (comme dans `int x = (y != 0)` qui initialise `x` avec le booléen disant si « `y` est différent de 0 »).

Lorsque l'on compile une expression arithmétique, par exemple, $e_1 + e_2$, le code que l'on souhaite obtenir est de la forme :

$reg_1 \leftarrow [\![e_2]\!]$
$dest\_reg \leftarrow [\![e_1]\!]$
`add` $reg_1, dest\_reg$
$\ldots$

où « $\ldots$ » dénote le code produit pour la suite du programme compilé. Il est intéressant de noter que ce schéma de compilation fonctionne également avec opérateurs non symétriques. Par exemple, pour la soustraction, puisque `sub` *src*, *dest* a pour effet $dest \leftarrow dest - src$, il faut bien que e_1 arrive dans *dest* et e_2 dans *src*.

On remarque que ce schéma de compilation des opérandes n'utilise qu'un seul pseudo-registre temporaire et transfère directement le résultat de l'opération dans le registre destination `dest_reg`. Cela n'est correct, comme nous le verrons pour la compilation d'une instruction d'affectation `v = e`, que parce que cette dernière fournit toujours un nouveau registre temporaire pour le calcul de l'expression *e*. Si l'on ne prenait pas ce soin, on pourrait avoir la valeur de la variable `v` modifiée en cours de route si elle apparaît plusieurs fois dans l'expression *e*. Dans la figure 12.4 ne sont données que quelques règles, les autres opérateurs arithmétiques étant gérés identiquement, à la pseudo-instruction utilisée près.

${}^{b}[\![\texttt{B\_addi}\ (i, e)]\!]_{\rho,dr,nl} = {}^{e}[\![e]\!]_{\rho,dr,\texttt{add\_node}\ (\texttt{Addi}\ (i, dr, nl))}$
${}^{b}[\![\texttt{B\_add}\ (e_1, e_2)]\!]_{\rho,dr,nl} =$ Soit r_2 un nouveau pseudo-registre, ${}^{e}[\![e_1]\!]_{\rho,dr,{}^{e}[\![e_2]\!]_{\rho,r_2,(}\texttt{add\_node}\ (\texttt{Add}\ (\texttt{r}_2, \texttt{dr}, \texttt{nl}))))}$
${}^{b}[\![\texttt{B\_subi}\ (i, e)]\!]_{\rho,dr,nl} = {}^{e}[\![e]\!]_{\rho,dr,\texttt{add\_node}\ (\texttt{Subi}\ (i, dr, nl))}$
${}^{b}[\![\texttt{B\_sub}\ (e_1, e_2)]\!]_{\rho,dr,nl} =$ Soit r_2 un nouveau pseudo-registre, ${}^{e}[\![e_1]\!]_{\rho,dr,{}^{e}[\![e_2]\!]_{\rho,r_2,(}\texttt{add\_node}\ (\texttt{Sub}\ (\texttt{r}_2, \texttt{dr}, \texttt{nl}))))}$
$\ldots$

FIGURE 12.4 – Traduction LLAST → RTL des opérateurs binaires arithmétiques

Le schéma de compilation des opérateurs relationnels est différent. En effet, lorsqu'ils ne sont pas utilisés dans une condition de boucle ou de test, ils n'ont pas de raison de provoquer un déroutement du flot d'exécution. Autrement dit, lorsque l'on compile `x = (y != 10)`, nous souhaitons juste obtenir dans `x` la valeur booléenne disant si `y` est différent de 10. Un schéma de compilation qui produirait (pour `x` représenté par r_2 et `y` par r_1) :

```
    mov $1, r2      % vrai par défaut
    cmpi $10, r1
    jne end         % test ok, on ne change rien
    mov $0, r2      % sinon on met faux
end :   ...
```

serait inefficace car un saut est toujours une opération coûteuse. Fort heureusement, il existe une famille d'instructions du X86-64 qui permet de s'affranchir d'un tel saut. Ce sont les `set`*cc* introduits en section 12.4.2. Ainsi, pour compiler une expression relationnelle, par exemple $e_1 == e_2$, nous générerons le code suivant :

```
reg1 ← [[e1]]
reg2 ← [[e2]]
cmp reg2, reg1
sete dest_reg
...
```

On remarque que contrairement aux expressions arithmétiques, le registre destination n'est utilisé qu'en toute fin de calcul. En effet, ces instructions n'ont pas une sémantique de la forme $dest \leftarrow dest\ op\ src$. L'instruction `set`*cc* changera (sa partie *cc*) en fonction de l'opérateur compilé.

${}^b[\![\texttt{B\_eq}\ (e_1, e_2)]\!]_{\rho,dr,nl} =$ Soient r_1, r_2 de nouveaux pseudo-registres, ${}^e[\![e_1]\!]_{\rho,r_1,}{}^e[\![e_2]\!]_{\rho,r_2,}\texttt{add\_node}\ (\texttt{Cmp}\ (r_2, r_1, \texttt{add\_node}\ (\texttt{Setcc}\ (\texttt{E}, dr, nl))))$
${}^b[\![\texttt{B\_neq}\ (e_1, e_2)]\!]_{\rho,dr,nl} =$ Soient r_1, r_2 de nouveaux pseudo-registres, ${}^e[\![e_1]\!]_{\rho,r_1,}{}^e[\![e_2]\!]_{\rho,r_2,}\texttt{add\_node}\ (\texttt{Cmp}\ (r_2, r_1, \texttt{add\_node}\ (\texttt{Setcc}\ (\texttt{Ne}, dr, nl))))$
...

FIGURE 12.5 – Traduction LLAST → RTL des opérateurs binaires relationnels (hors expression de condition)

En fonction de l'opérateur compilé, seul le *cc* change : == (`B_eq`) → `E`, != (`B_neq`) → `Ne`, < (`B_l`) → `L`, <= (`B_le`) → `Le`, > (`B_g`) → `G`, >= (`B_ge`) → `Ge`. Rappelons que l'instruction `Cmp` (e_1, e_2) réalise $e_2 - e_1$ pour positionner les bits du registre de statut, d'où l'ordre « inversé » des opérandes dans le code généré.

Traduction des conditions

Avant de pouvoir compiler les instructions, et en particulier les tests et les boucles, il est nécessaire de compiler les expressions utilisées comme conditions. Elles feront intervenir des opérateurs relationnels comme ci-dessus, mais dans un contexte totalement différent puisqu'elles engendreront des ruptures de séquence (des sauts). Le modèle de compilation utilisé est en tout point similaire à celui que nous avons mis en œuvre dans le chapitre 11.3.2. La traduction est réalisée par la fonction fonction ${}^c[\![e]\!]_{\rho,t,lt,lf}$ où t est un booléen disant si l'on doit tester la condition ou sa négation (et sauter le cas échéant à l'étiquette

adéquate), lt (*label true*) est l'étiquette où sauter si la condition est satisfaite et lf (*label false*) où sauter si sa négation l'est.

Rappelons que, comme précisé dans la section 12.4.2, nous faisons en sorte que tout saut aboutisse à une (pseudo)-instruction de type `Label`. La traduction des conditions respectera donc cet invariant.

${}^c[\![\texttt{E\_binop (B\_land } (e_1, e_2))]\!]_{\rho,t,lt,lf} =$ Soit *lab* une nouvelle étiquette de `Label`, ${}^c[\![e_1]\!]_{\rho,false,\texttt{add\_node (Label } (lab, "and2", ({}^c[\![e_2]\!]_{\rho,false,lt,lf}))),lf}$
${}^c[\![\texttt{E\_binop (B\_lor } (e_1, e_2))]\!]_{\rho,t,lt,lf} =$ Soit *lab* une nouvelle étiquette de `Label`, ${}^c[\![e_1]\!]_{\rho,false,lt,\texttt{add\_node (Label } (lab, "or2", ({}^c[\![e_2]\!]_{\rho,true,lt,lf})))}$
${}^c[\![\texttt{E\_binop (B\_eq } (e_1, e_2))]\!]_{\rho,t,lt,lf} =$ Soit *jccl* = si t alors `add_node (Jcc (E,` lt, lf`))` sinon `add_node (Jcc (Ne,` lf, lt`))` Soient r_1, r_2 de nouveaux pseudo-registres, ${}^e[\![e_1]\!]_{\rho,r_1,{}^e[\![e_2]\!]_{\rho,r_2,\texttt{add\_node (Cmp } (r_2, r_1, jccl))}}$
${}^c[\![\texttt{E\_binop (B\_ge } (e_1, e_2))]\!]_{\rho,t,lt,lf} =$ Soit *jccl* = si t alors `add_node (Jcc (Ge,` lt, lf`))` sinon `add_node (Jcc (L,` lf, lt`))` Soient r_1, r_2 de nouveaux pseudo-registres, ${}^e[\![e_1]\!]_{\rho,r_1,{}^e[\![e_2]\!]_{\rho,r_2,\texttt{add\_node (Cmp } (r_2, r_1, jccl))}}$
...
${}^c[\![\texttt{E\_binop } (e)]\!]_{\rho,t,lt,lf} =$ Soit *jccl* = si t alors `add_node (Jcc (Ne ,` lt, lf`))` sinon `add_node (Jcc (E ,` lf, lt`))` Soit r un nouveau pseudo-registre, ${}^e[\![e]\!]_{\rho,r,\texttt{add\_node (Cmpi (Int64.zero,}\ r, jccl))}$

FIGURE 12.6 – Traduction LLAST → RTL des conditions

On remarque que le même schéma se répète pour les opérateurs d'ordre : si le booléen t est vrai on effectue un `j`*cc* avec le code de condition correspondant à l'opérateur, sinon avec le code « inverse » :

- pour `B_eq` (==) : soit `E` soit `Ne`,
- pour `B_neq` (!=) : soit `Ne` soit `E`,
- pour `B_g` (>) : soit `G` soit `Le`,
- pour `B_l` (<) : soit `L` soit `Ge`,
- pour `B_ge` (>=) : soit `Ge` soit `L`,
- pour `B_le` (<=) : soit `Le` soit `G`.

Le dernier cas des règles correspond à une expression quelconque qui n'est pas un opérateur. En accord avec la sémantique de nombreux langages, nous considérons que 0 correspond à « faux » et $\neq 0$ à « vrai ». Nous générons donc un code similaire aux autres autres cas, instancié avec avec une comparaison avec la valeur immédiate 0.

Traduction des instructions

Nous pouvons désormais présenter la compilation des instructions. Cette fonction est naturellement récursive tout comme l'est la structure des instructions. Elle s'appuie sur la compilation des expressions et des conditions.

${}^{i}[\![\texttt{I\_while}\ (e, i)]\!]_{\rho,pl,rr,nl} =$ Soit *jcl* une nouvelle étiquette de graphe, Soient lab_1, lab_2, lab_3 des nouvelles étiquettes de `Label`, Soit *cl* = `add_node` (`Label` (lab_1, *"lcond"*, $({}^{c}[\![e]\!]_{\rho,false,}$`add_node ( Label (`$lab_2$, *"lbody"*, ${}^{i}[\![i]\!]_{\rho,pl,rr,jcl}$`)),add_node (Label (`$lab_3$, *"lexit"*, nl`))`$)$`))` Ajouter (`Jmp` *cl*) à l'étiquette *jcl* dans le graphe *cl*
${}^{i}[\![\texttt{I\_if}\ (e, i_1, \texttt{Some}\ i_2)]\!]_{\rho,pl,rr,nl} =$ Soient lab_1, lab_2, lab_3 des nouvelles étiquettes de `Label`, Soit $eifl =$ `add_node (Label (`$lab1$, *"endif"*, nl`))` ${}^{c}[\![e]\!]_{\rho,false,}$`add_node (Label (`$lab2$, *"then"*, $({}^{i}[\![i_1]\!]_{\rho,pl,rr,eifl})$`)),add_node (Label (`$lab3$, *"else"*, $({}^{i}[\![i_2]\!]_{\rho,pl,rr,eifl})$`))`
${}^{i}[\![\texttt{I\_if}\ (e, i, \texttt{None})]\!]_{\rho,pl,rr,nl} =$ Soit *lab* une nouvelle étiquette de `Label`, Soit $eifl =$ `add_node (Label (`*lab*, *"endif"*, nl`))` ${}^{c}[\![e]\!]_{\rho,false,({}^{i}[\![i]\!]_{\rho,pl,rr,eifl}),eifl}$
${}^{i}[\![\texttt{I\_assign}\ (v, e)]\!]_{\rho,pl,rr,nl} = {}^{e}[\![e]\!]_{\rho,\rho(v),nl}$
${}^{i}[\![\texttt{I\_seq}\ (i_1, i_2)]\!]_{\rho,pl,rr,nl} = {}^{i}[\![i_1]\!]_{\rho,pl,rr,}{}^{i}[\![i_2]\!]_{\rho,pl,rr,nl}$
${}^{i}[\![\texttt{I\_return}\ e]\!]_{\rho,pl,rr,nl} =$ Soit r un nouveau pseudo-registre, ${}^{e}[\![e]\!]_{\rho,r,}$`add_node (Mov (`$r, rr,$ `(add_node (Jmp` pl`))))`
${}^{i}[\![\texttt{I\_return}]\!]_{\rho,pl,rr,nl} =$ `add_node (Jmp` pl`)`
${}^{i}[\![\texttt{I\_app}\ (f, [e_1; \ldots; e_n])]\!]_{\rho,pl,rr,nl} =$ Soit r_0 un nouveau pseudo-registre, Soit $[r_1; \ldots; r_n]$ une liste de n de nouveaux pseudo-registres, ${}^{a}[\![[e_1; \ldots; e_n], [r_1; \ldots; r_n]]\!]_{\rho,(}$`add_node (Call (`$r_0, f, [r_1; \ldots; r_n], nl$`))`$)$
${}^{i}[\![\texttt{I\_print}\ e]\!]_{\rho,pl,rr,nl} =$ Soit r un nouveau pseudo-registre, ${}^{e}[\![e]\!]_{\rho,r,}$`add_node (Print (`*"printf"*, r, nl`))`

FIGURE 12.7 – Traduction LLAST → RTL des instructions

La compilation d'une boucle est le seul point subtil. Comme nous l'avons précisé en début de cette section, nous construisons le graphe en commençant par la fin de la fonction. Ainsi, à la fin d'une boucle, il faut générer un saut vers le nœud effectuant l'évaluation de la condition de la boucle. Or, ce nœud n'est pas encore créé. Ainsi, nous générons une nouvelle étiquette de graphe *jcl* (*jump to condition label*), rattachée à aucune instruction. Lorsque nous compilons le corps de la boucle, nous spécifions que la prochaine étiquette d'instruction est *jcl*, qui nous renverra vers la condition. Une fois le code de la boucle généré, nous obtenons l'étiquette de la première instruction de la condition, *cl* (*condition loop*). Il ne nous reste plus qu'à insérer dans le graphe à l'étiquette *jcl*, une instruction de saut vers *cl* : `Jmp` *cl*. C'est le seul endroit dans la traduction LLAST → RTL où l'on insère une instruction dans le graphe sans passer par le mécanisme habituel utilisant `add_node`. La raison de cette particularité est qu'une boucle devant créer un cycle dans le graphe, il est impossible de construire ce dernier récursivement (sans boucler). En quelque sorte, nous retrouvons ici le point fixe que la boucle dénote.

La compilation de `I_return` utilise l'étiquette du nœud de postlude *pl* comme prochaine instruction à utiliser. En particulier, contrairement à ce que l'on pourrait attendre, `I_return` ne génère pas d'instruction `Ret`. La raison est que retourner d'une fonction n'est pas *que* exécuter un `Ret`. Beaucoup

« d'administratif » va devoir être fait avant ce `Ret` final (transfert du résultat là où l'appelant l'attend, nettoyage de la pile, etc.). Les instructions effectuant ces traitements seront insérées plus tard au niveau du nœud de postlude, (lors de la construction de l'ERTL puis du LTL).

Traduction des définitions de fonctions

Il ne reste plus qu'à mettre tout ça ensemble pour compiler une définition de fonction complète. Initialement le graphe est vide (rappelons qu'il y a un graphe par définition de fonction et non un graphe global à tout le programme, ce qui ne serait pas possible avec de la compilation séparée).

$[\![\{ f; [p_1; \ldots; p_n]; [(v_1, vi_1); \ldots; (v_n, vi_n)]; i \}]\!] =$
Soit G un graphe vide (implicitement le graphe courant),
Soient lab_1, lab_2 de nouvelles étiquettes de `Label`,
Soient $r_1, \ldots; r_n, r_{n+1} \ldots; r_{n+m}$ de nouveaux pseudo-registres,
Soit $\rho = [(p_1, r_1); \ldots; (p_n, r_n); (v_1, r_{n+1}); \ldots; (v_m, r_{n+m})]$,
Soit pl = `add_node (Label (`lab_1`, "postlude", add_node (Ret)))`
Soit rr = un nouveau pseudo-registre,
Soit $bl = {}^{i}[\![i]\!]_{\rho,pl,rr,pl}$
Soit $il = {}^{v}[\![[(v_1, vi_1); \ldots; (v_n, vi_n)]]\!]_{\rho,bl}$
Soit rl = `add_node (Label (`lab_2`, "body",` il`))`
$\{ f; [r_1, \ldots; r_n]; rr; G; rl; pl \}$

FIGURE 12.8 – Traduction LLAST → RTL d'une fonction

On commence par créer deux étiquette de pseudo-instructions `Label`. Cela nous permettra d'identifier le nœud de début de fonction ainsi que le nœud de postlude. On crée ensuite autant de pseudo-registres qu'il y a de paramètres et de variables locales. L'environnement ρ initial associe paramètres et variables locales à leurs pseudo-registres. Le corps de la fonction est traduit en utilisant l'étiquette de postlude *pl* (*postlude label*) créée, un nouveau pseudo-registre *rr* comme réceptacle de la valeur de retour et en indiquant que l'étiquette représentant « le reste du code » est le postlude. Il faut également compiler les potentielles initialisations de variables locales. Cela est réalisé en disant que « le reste du code » est l'étiquette de début du corps *bl* (*body label*). Finalement, la racine du graphe *rl* (*root label*) est un nœud `Label` dont le successeur est *il* (*inits label*) l'étiquette de début des initialisations de variables locales. Tout cela se retrouve agrégé dans une structure récapitulative représentant la traduction de la définition de fonction.

La traduction des initialisations de variables locales se contente de compiler les expressions d'initialisation lorsqu'il y en a, en chaînant les sous-graphes obtenus.

$$
\begin{array}{lcl}
{}^{v}[\![[]]\!]_{\rho,nl} & = & nl \\
{}^{v}[\![[(v_1, \texttt{None}); \ldots]]\!]_{\rho,nl} & = & {}^{v}[\![[\ldots]]\!]_{\rho,nl} \\
{}^{v}[\![[(v_1, \texttt{Some } e); \ldots]]\!]_{\rho,nl} & = & {}^{e}[\![e]\!]_{\rho,\rho(v_1),{}^{v}[\![[\ldots]]\!]_{\rho,nl}}
\end{array}
$$

FIGURE 12.9 – Traduction LLAST → RTL des initialisations de variables locales

12.4.4 Implantation en OCAML

L'implantation en OCAML est réalisée au travers d'autant de fonctions (certaines mutuellement récursives). Dans les codes qui suivent, l'indentation est volontairement « anormale » pour faire ressortir la structure du code généré plutôt que l'imbrication des appels de fonctions de l'implantation. Certains arguments des fonctions sont étiquetés (notation `~nom`) afin de les nommer explicitement et de

faciliter la relecture de code. Ainsi, on réduit les chances de se tromper dans l'ordre de passage d'arguments du même type. Ainsi, pour une fonction `f ~x ~y ~z` à 3 arguments, au lieu d'effectuer un appel par `f 10 20 30`, nous devrons écrire `f ~x: 10 ~y: 20 ~z: 30`, rendant bien visible quel paramètre reçoit quelle valeur et réduisant les risques d'inversion malencontreuse.

La fonction `find_register` permet de rechercher le pseudo-registre associé à un identificateur (paramètre formel ou variable locale). Puisqu'une passe de résolution de porté a déjà eu lieu, il est impossible de ne pas trouver un identificateur dans l'environnement (ou alors le compilateur est buggé). Viennent ensuite les fonctions traduisant les expressions :

- `do_relation_binop` qui implante $^{r}[\![\ ]\!]$,
- `do_binop` qui implante $^{b}[\![\ ]\!]$,
- `do_unop` qui implante le cas `E_Uno` de $^{e}[\![\ ]\!]$,
- `do_expr` qui implante $^{e}[\![\ ]\!]$,
- `do_cond_expr` et `do_cond_binop` qui implantent $^{c}[\![\ ]\!]$,
- `do_call_args` et `do_funcall` qui implantent le cas `E_app` de $^{e}[\![\ ]\!]$ et le cas `I_app` de $^{i}[\![\ ]\!]$.

```
let find_register var env = try List.assoc var env with Not_found →assert false

let rec do_relation_binop env graph ~dest_reg cc e1 e2 next =
  let reg_e1 = RTL.new_reg () in
  let reg_e2 = RTL.new_reg () in
  do_expr env ~graph ~dest_reg: reg_e1 e1
  (do_expr env ~graph ~dest_reg: reg_e2 e2
  (Graph.add_node graph (RTL.Cmp (reg_e2, reg_e1,
  Graph.add_node graph (RTL.Setcc (cc, dest_reg, next))))))

and do_binop env graph ~dest_reg binop next =
  match binop with
  | LLAst.B_addi (i, e) →
      do_expr env graph ~dest_reg e
      (Graph.add_node graph (RTL.Addi (i, dest_reg, next)))
  | LLAst.B_add (e1, e2) →
      let reg2 = RTL.new_reg () in
      do_expr env graph ~dest_reg e1
      (do_expr env graph ~dest_reg: reg2 e2
      (Graph.add_node graph (RTL.Add (reg2, dest_reg, next))))
  | LLAst.B_subi (i, e) →
      do_expr env graph ~dest_reg e
      (Graph.add_node graph (RTL.Subi (i, dest_reg, next)))
  | LLAst.B_sub (e1, e2) →
      let reg2 = RTL.new_reg () in
      do_expr env graph ~dest_reg e1
      (do_expr env graph ~dest_reg: reg2 e2
      (Graph.add_node graph (RTL.Sub (reg2, dest_reg, next))))
   ...
  | LLAst.B_eq (e1, e2) → do_relation_binop env graph ~dest_reg RTL.E e1 e2 next
  | LLAst.B_neq (e1, e2) → do_relation_binop env graph ~dest_reg RTL.Ne e1 e2 next
   ...

and do_unop env graph ~dest_reg unop next =
  match unop with
  | LLAst.U_neg e →
      do_expr env graph ~dest_reg e (Graph.add_node graph (RTL.Neg (dest_reg, next)))

and do_expr env graph ~dest_reg expr next =
```

```
  match expr with
  | LLAst.E_int i →Graph.add_node graph (RTL.Movi (i, dest_reg, next))
  | LLAst.E_ident var →
      let reg = find_register var env in
      Graph.add_node graph (RTL.Mov (reg, dest_reg, next))
  | LLAst.E_app (f_name, args) →do_funcall env graph ~dest_reg f_name args next
  | LLAst.E_unop unop →do_unop env graph ~dest_reg unop next
  | LLAst.E_binop binop →do_binop env graph ~dest_reg binop next

and do_cond_expr env graph ~test ~true_node ~false_node expr =
  match expr with
  | LLAst.E_binop (LLAst.B_land ( e1, e2)) →
      do_cond_expr env graph ~test: false
        ~true_node: (Graph.add_node graph (RTL.Label ((RTL.new_label ()), "and2",
                    (do_cond_expr env graph ~test: false ~true_node ~false_node e2))))
        ~false_node
        e1
  | LLAst.E_binop (LLAst.B_lor (e1, e2)) →
      do_cond_expr env graph ~test: false
        ~true_node
        ~false_node: (Graph.add_node graph (RTL.Label ((RTL.new_label ()), "or2",
                     (do_cond_expr env graph ~test: true ~true_node ~false_node e2))))
        e1
  | LLAst.E_binop (LLAst.B_eq (e1, e2)) →
      do_cond_binop env graph ~test ~true_node ~false_node ~true_cc: RTL.E ~false_cc: RTL.Ne e1 e2
  ...
  | _ →
      let jcc_graph_label =
        if test then Graph.add_node graph (RTL.Jcc (RTL.Ne, true_node, false_node))
        else Graph.add_node graph (RTL.Jcc (RTL.E, false_node, true_node)) in
      let e_reg = RTL.new_reg () in
      do_expr env graph ~dest_reg: e_reg expr
      (Graph.add_node graph (RTL.Cmpi (Int64.zero, e_reg, jcc_graph_label)))

and do_cond_binop env graph ~test ~true_node ~false_node ~true_cc ~false_cc e1 e2 =
  let jcc_graph_label =
    if test then Graph.add_node graph (RTL.Jcc (true_cc, true_node, false_node))
    else Graph.add_node graph (RTL.Jcc (false_cc, false_node, true_node)) in
  let reg1 = RTL.new_reg () in
  let reg2 = RTL.new_reg () in
  do_expr env graph ~dest_reg: reg1 e1
  (do_expr env graph ~dest_reg: reg2 e2
  (Graph.add_node graph (RTL.Cmp (reg2, reg1, jcc_graph_label))))

and do_call_args env graph args dest_regs next =
  let rec rec_do = function
    | ([], []) → next
    | ((h :: q), (dest_reg :: rem_dest_regs)) →do_expr env graph ~dest_reg h (rec_do (q, rem_dest_regs))
    | (_, _) → assert false in
  rec_do (args, dest_regs)

and do_funcall env graph ~dest_reg f_name args next =
  let f_name' = { f_name with Ast.v_name = asm_fun_name f_name.Ast.v_name } in
  let dest_regs = List.map (fun _ →RTL.new_reg ()) args in
  do_call_args env graph args dest_regs
```

```
    (Graph.add_node graph (RTL.Call (dest_reg, f_name', dest_regs, next)))

let rec do_instr env graph ~postlude_n ~ret_reg instr next =
  match instr with
  | LLAst.I_while (e, i) →
      let jump_to_cond_label = Graph.new_graph_label () in
      let cond_n =
        Graph.add_node graph (RTL.Label ((RTL.new_label ()), "lcond",
        (do_cond_expr env graph ~test: false
           ~true_node:
             (Graph.add_node graph (RTL.Label ((RTL.new_label ()), "lbody",
             (do_instr env graph ~postlude_n ~ret_reg i jump_to_cond_label))))
           ~false_node:
             (Graph.add_node graph (RTL.Label ((RTL.new_label ()), "lexit", next)))
           e))) in
      Graph.add_node_with_label graph jump_to_cond_label (RTL.Jmp cond_n) ;
      cond_n
  | LLAst.I_if (e, i1, (Some i2)) →
      let endif_n = Graph.add_node graph (RTL.Label ((RTL.new_label ()), "endif", next)) in
      do_cond_expr
        env graph ~test: false
        ~true_node:
          (Graph.add_node graph (RTL.Label ((RTL.new_label ()), "then",
          (do_instr env graph ~postlude_n ~ret_reg i1 endif_n))))
        ~false_node:
          (Graph.add_node graph (RTL.Label ((RTL.new_label ()), "else",
          (do_instr env graph ~postlude_n ~ret_reg i2 endif_n))))
        e
  | LLAst.I_if (e, i, None) →
      let endif_n = Graph.add_node graph (RTL.Label ((RTL.new_label ()), "endif", next)) in
      do_cond_expr env graph ~test: false
        ~true_node: (do_instr env graph ~postlude_n ~ret_reg i endif_n)
        ~false_node: endif_n
        e
  | LLAst.I_assign (var, e) →
      let var_reg = find_register var env in
      let dest_reg = RTL.new_reg () in
      do_expr env graph ~dest_reg e
      (Graph.add_node graph (RTL.Mov (dest_reg, var_reg, next)))
  | LLAst.I_seq (i1, i2) →
      do_instr env graph ~postlude_n ~ret_reg i1
      (do_instr env graph ~postlude_n ~ret_reg i2 next)
  | LLAst.I_return e_opt →
      (match e_opt with
      | None → Graph.add_node graph (RTL.Jmp postlude_n)
      | Some e →
          let reg_e = RTL.new_reg () in
          do_expr env graph ~dest_reg: reg_e e
          (Graph.add_node graph (RTL.Mov (reg_e, ret_reg,
          (Graph.add_node graph (RTL.Jmp postlude_n))))))
  | LLAst.I_app (f_name, args) →
      let dummy_reg = RTL.new_reg () in
      do_funcall env graph ~dest_reg: dummy_reg f_name args next
  | LLAst.I_print e →
      let printf_name = {
```

```
        Ast.v_name = asm_fun_name "printf" ;
        Ast.v_stamp = 0 ;
        Ast.v_scope = Ast.S_global } in
      let reg = RTL.new_reg () in
      do_expr env graph ~dest_reg: reg e
      (Graph.add_node graph (RTL.Print (printf_name, reg, next)))

let do_local_inits env graph next inits =
  let rec work = function
    | [ ] → next
    | { Ast.ast_desc = (var, _, init_opt) } :: q →
        (match init_opt with
        | None → work q
        | Some e →
            let dest_reg = find_register var env in
            do_expr env graph ~dest_reg e (work q)
        ) in
  work inits

let do_toplevel top =
  match top.Ast.ast_desc with
  | LLAst.TOP_fundef f_def →
      let graph = ref Graph.G.empty in
      let env_params =
        List.map (fun { Ast.ast_desc = (var, _) } → (var, RTL.new_reg ())) f_def.LLAst.fdef_params in
      let env =
        List.fold_left
          (fun accu { Ast.ast_desc = (var, _, _) } → (var, RTL.new_reg ()) :: accu)
          env_params f_def.LLAst.fdef_vars in
      let postlude_n =
        Graph.add_node graph (RTL.Label (RTL.new_label (), "fun postlude",
        (Graph.add_node graph RTL.Ret))) in
      let ret_reg = RTL.new_reg () in
      let start_body =
        do_instr env graph ~postlude_n ~ret_reg: ret_reg f_def.LLAst.fdef_body postlude_n in
      let start_inits = do_local_inits env graph start_body f_def.LLAst.fdef_vars in
      let root_n =
        Graph.add_node graph (RTL.Label (RTL.new_label (), (f_def.LLAst.fdef_name.Ast.v_name ^ "_body"),
        start_inits)) in
      let fun_name = {
        f_def.LLAst.fdef_name with Ast.v_name = asm_fun_name f_def.LLAst.fdef_name.Ast.v_name } in
      { RTL.fdef_name = fun_name ; RTL.fdef_params = List.map snd env_params ;
        RTL.fdef_ret_reg = ret_reg ; RTL.fdef_vars = f_def.LLAst.fdef_vars ;
        RTL.fdef_graph = !graph ; RTL.fdef_entry = root_n ;
        RTL.fdef_postlude = postlude_n }
```

Les lecteurs attentifs auront remarqué un appel à une fonction `asm_fun_name` dans la traduction d'une définition de fonction. Cette fonction permet de traduire le nom d'une fonction en accord avec la manière de laquelle l'assembleur (le logiciel d'assemblage, pas le langage) fait référence aux symboles globaux. Comme évoqué dans la section 11.3.4, selon les OS les noms de symboles peuvent être préfixés par un _. C'est un point de détail mais le choix est fait ici de prendre en compte cette subtilité dès la traduction vers RTL afin de ne plus s'en soucier dans la suite.

12.4.5 Résumé

Nous avons donc obtenu une nouvelle représentation du programme, le RTL, sous forme d'un graphe de flot de contrôle. Les nœuds du graphe contiennent des pseudo-instructions opérant sur des pseudo-registres en nombre illimité. Aucun registre physique n'apparaît. Les variables locales, les paramètres des fonctions et le résultat retourné à l'appelant sont représentés par des pseudo-registres. Les boucles et les conditionnelles ont donné lieu à des sauts explicites. Aucune convention d'appel n'a été explicitée et le postlude des fonctions est encore vide.

12.5 Vers ERTL

Maintenant que la structure du programme est celle d'un graphe, nous allons transformer ce dernier en lui injectant de nouveaux nœuds. Comme précisé dans la section 12.4.1, la représentation du graphe sous forme d'une `Map` d'étiquettes va nous permettre de conserver ces dernières d'un graphe sur l'autre.

Nous allons construire une nouvelle représentation intermédiaire du programme, le *Explicit Register Transfer Language* (ERTL). Dans ce graphe, les registres physiques commencent à faire leur apparition pour des instructions spécifiques (la division par exemple). Le protocole d'appel des fonctions devient explicite : paramètres passés dans les registres imposés par l'ABI, paramètres restants passés sur la pile, retour dans le registre `%eax`. Les registres à sauvegarder par l'appelé sont également pris en compte et des marqueurs d'allocation et de libération de *stack frame* sont insérés.

12.5.1 Représentation des registres

Puisque les registres peuvent désormais être soit des pseudo-registres soit des registres physiques, le module `Regs` doit donc fournir différentes définitions pour manipuler cette nouvelle représentation.

Les registres physiques seront représentés par des chaînes de caractères (éventuellement rendues abstraites par un fichier d'interface `.mli`). Les processeurs X86-64 disposent de nombreux registres, que l'on peut adresser sur 8, 16, 32 ou 64 bits. Nous n'allons pas tous les détailler. Sachant que notre langage ne gère que des entiers signés sur 64 bits, nous ne considérerons que les registres 64 bits. Ces registres sont : `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`, `%rsp` et `%rbp`. Le registre `%rsp` est le *pointeur de pile* et ne peut donc être utilisé comme registre de calcul arbitraire. De même, `%rbp` servira de *frame pointer* (comme dans vu en section 11.3, on accédera aux variables locales et aux paramètres via un décalage relatif à ce registre) et ne pourra être utilisé pour les calculs. Pour terminer, nous verrons ultérieurement que nous auront besoin d'un registre temporaire (pour nous simplifier la tâche) : ce sera le registre `%r15`.

```
type reg_t = string
type temp_t = R_pseudo of int | R_phys of reg_t
module RegMod = struct type t = temp_t let compare = compare end
module RegSet = Set.Make(RegMod)
module RegMap = Map.Make(RegMod)
let rax = "%rax"
let rbx = "%rbx"
let rcx = "%rcx"
...
```

Le pseudo-registres sont représentés comme des entiers. Les registres ERTL, appelés des *temporaires* (car ils seront ultimement remplacés par des registres physiques ou par des emplacements dans la pile). On se définit pour de multiples utilisations futures des ensembles et des `Map` de temporaires. Afin de ne pas débuter par une liste de définitions pour le moment injustifiées, le module `Regs` sera étendu plus tard lorsque nous en rencontrerons le besoin.

12.5.2 Structure du ERTL

La structure du graphe est la même que celle du RTL, seules les instructions changent. Ces instructions restent pour autant très similaires à celle du RTL, si ce n'est qu'elles manipulent la nouvelle représentation des registres. Dans la suite de cette section ainsi que l'allocation de registre, nous emploierons indifféremment les termes « registre » et « temporaires ». Lorsque nous aurons besoin de préciser, nous dirons explicitement « pseudo » ou « physique ».

Sept nouvelles instructions font leur apparition et l'instruction d'appel de fonction est modifiée :

- `Push` : permet d'empiler un registre au sommet de la pile.
- `Pop` : permet de dépiler le sommet de la pile et de le mettre dans un registre.
- `Frame_start` : pseudo-instruction indiquant où insérer ultérieurement l'allocation de la *stack frame*.
- `Frame_end` : pseudo-instruction indiquant où insérer ultérieurement la libération de la *stack frame*.
- `Stack_prm_read` : représente la lecture d'un paramètre qui a été transmis sur la pile. Cette pseudo-instruction comporte le temporaire destination ainsi que le déplacement (*offset*) relatif à `rbp` où trouver ce paramètre (cf. section 11.3 pour le mécanisme d'accès relatif).
- `Cqto` : une instruction similaire à `cdq` utilisée dans la section 11.3.1 pour compiler la division. Elle permet d'étendre la valeur de `%rax` dans `%rdx:%rax`.
- `Lea_rip` : instruction permettant de charger une adresse dans un registre, relativement au pointeur d'instruction. Nous utiliserons cette instruction pour compiler, toujours de manière *ad hoc*, les appels à `printf` pour l'instruction `print` de notre langage. Elle permettra de transmettre l'adresse du « format » `"%d\n"` à `print`. Cette instruction comporte une chaîne de caractère qui représentera l'étiquette, dans le code final, où sera définie la chaîne du format. Elle comporte également le temporaire destination dans lequel stocker l'adresse obtenue.
- `Call` : désormais, cette instruction mémorise le nombre d'arguments passés par registres et par la pile. La première information sera nécessaire dans la phase suivante pour déterminer la durée de vie des registres (sous-étape de l'allocation de registres). La seconde le sera pour générer la correction de la pile garantissant un alignement sur 16 octets aux sites d'appel. Cette correction ne peut pas être calculée au passage à ERTL car on ne connaît pas la taille de la *stack frame*. En effet, l'allocation de registres, qui sera faite ultérieurement, va possiblement causer l'utilisation de la pile.

De manière similaire au RTL, une définition de fonction sera représentée par un enregistrement regroupant les informations ERTL qui la concerne : son nom, son graphe ERTL, l'étiquette du nœud point d'entrée et un champ mutable représentant la taille de sa *stack frame*. Ce dernier champ sera initialisé avec une valeur nulle et sera mis à jour par effet de bords après l'allocation de registres. Cette astuce permet de conserver la même structure de donnée en sortie d'allocation de registres, évitant de multiplier inutilement les représentations intermédiaires (on en est déjà à la quatrième – AST, LLAST, RTL, ERTL – et il nous en reste encore une !).

L'implantation en OCaml est la suivante, dans laquelle les registres sont désormais des `Regs.temp_t`. Seules quelques instructions isomorphes à celle de RTL et celles faisant leur apparition sont données.

```
type asm_instr_t =
  | Movi of (Int64.t * Regs.temp_t * Graph.graph_label_t)
  | Mov of (Regs.temp_t * Regs.temp_t * Graph.graph_label_t)
  | Addi of (Int64.t * Regs.temp_t * Graph.graph_label_t)
  | Add of (Regs.temp_t * Regs.temp_t * Graph.graph_label_t)
...
  | Call of (Ast.var_t * int * int * Graph.graph_label_t)
  | Cqto of Graph.graph_label_t
  | Push of (Regs.temp_t * Graph.graph_label_t)
  | Pop of (Regs.temp_t * Graph.graph_label_t)
  | Frame_start of Graph.graph_label_t
```

```
  | Frame_end of Graph.graph_label_t
  | Stack_prm_read of (Regs.temp_t * int * Graph.graph_label_t)
  | Lea_rip of (string * Regs.temp_t * Graph.graph_label_t)

type fun_def_t = {
  fdef_name : Ast.var_t ;
  fdef_graph : asm_instr_t Graph.G.t ;
  fdef_entry : Graph.graph_label_t ;
  mutable fdef_frame_size : int
}
```

12.5.3 Construction du graphe ERTL

Comme présenté en introduction, ce graphe va être une extension du RTL dans laquelle on va venir greffer de nouveaux nœuds et en remplacer certains. Le fait de partager les étiquettes de graphe entre les graphes permet une implémentation très simple de la construction. Il n'est pas nécessaire d'effectuer un parcours récursif du graphe, seule une itération sur la `Map` représentant le graphe suffit, en traduisant « localement » (séparément) chaque instruction. Cela nous affranchit des usuelles mémorisations de nœuds déjà rencontrés pour éviter de boucler dans les parcours de graphes. Cela nous évite également de devoir se souvenir quel nœud ERTL a été déjà généré pour un nœud RTL que l'on rencontrerait une nouvelle fois : ils auront la même étiquette puisqu'elles sont partagées.

De nombreuses petites fonctions auxiliaires vont être nécessaires pour gérer l'explicitation des conventions d'appel, alors que la fonction de traduction des instructions sera, dans la majorité des cas, très simple. Nous définirons ces fonctions dans un second temps, lorsque nous en aurons constaté le besoin. Nous notons $[\![i]\!]$ la traduction de l'instruction RTL i en ERTL. Il est intéressant de noter que cette fonction n'a pas besoin d'environnement puisque tous les identificateurs ont été remplacés par des pseudo-registres dans le RTL. Dans les règles du tableau 12.5.3, implicitement les constructeurs de gauche appartiennent à RTL et ceux de droite à ERTL. Contrairement à la traduction vers RTL où il y avait de multiples fonctions mutuellement récursives, certaines traitant les instructions, d'autre les expressions, d'autres les expressions conditionnelles, nous utiliserons un unique nom de fonction de traduction (pour alléger la présentation).

$$
\begin{array}{l}
[\![\texttt{Movi}\ (i, r, nl)]\!] = \texttt{Movi}\ (i, (\texttt{R\_pseudo}\ r), nl) \\
[\![\texttt{Mov}\ (r_1, r_2, nl)]\!] = \texttt{Mov}\ ((\texttt{R\_pseudo}\ r_1), (\texttt{R\_pseudo}\ r_2), nl) \\
[\![\texttt{Addi}\ (i, r, nl)]\!] = \texttt{Addi}\ (i, (\texttt{R\_pseudo}\ r), nl) \\
[\![\texttt{Add}\ (r_1, r_2, nl)]\!] = \texttt{Add}\ ((\texttt{R\_pseudo}\ r_1), (\texttt{R\_pseudo}\ r_2), nl) \\
[\![\texttt{Subi}\ (i, r, nl)]\!] = \texttt{Subi}\ (i, (\texttt{R\_pseudo}\ r), nl) \\
[\![\texttt{Sub}\ (r_1, r_2, nl)]\!] = \texttt{Sub}\ ((\texttt{R\_pseudo}\ r_1), (\texttt{R\_pseudo}\ r_2), nl) \\
[\![\texttt{Muli}\ (i, r, nl)]\!] = \texttt{Muli}\ (i, (\texttt{R\_pseudo}\ r), nl) \\
[\![\texttt{Mul}\ (r_1, r_2, nl)]\!] = \texttt{Mul}\ ((\texttt{R\_pseudo}\ r_1), (\texttt{R\_pseudo}\ r_2), nl) \\
[\![\texttt{Div}\ (r_1, r_2, nl)]\!] = \\
\quad \texttt{EMov}\ ((\texttt{R\_pseudo}\ r_2), (\texttt{R\_phys}\ \texttt{rax}), \\
\quad\quad \texttt{add\_node}\ (\texttt{ECqto} \\
\quad\quad\quad (\texttt{add\_node}\ (\texttt{EDiv}\ ((\texttt{R\_pseudo}\ r_1), \\
\quad\quad\quad\quad \texttt{add\_node}\ (\texttt{EMov}\ ((\texttt{R\_phys}\ \texttt{rax}), (\texttt{R\_pseudo}\ r_2), nl))))))) \\
[\![\texttt{Mod}\ (r_1, r_2, nl)]\!] \\
\quad \texttt{EMov}\ ((\texttt{R\_pseudo}\ r_2), (\texttt{R\_phys}\ \texttt{rax}), \\
\quad\quad \texttt{add\_node}\ (\texttt{ECqto} \\
\quad\quad\quad (\texttt{add\_node}\ (\texttt{EDiv}\ ((\texttt{R\_pseudo}\ r_1), \\
\quad\quad\quad\quad \texttt{add\_node}\ (\texttt{EMov}\ ((\texttt{R\_phys}\ \texttt{rdx}), (\texttt{R\_pseudo}\ r_2), nl))))))) \\
[\![\texttt{Cmp}\ (r_1, r_2, nl)]\!] = \texttt{Cmp}\ ((\texttt{R\_pseudo}\ r_1), (\texttt{R\_pseudo}\ r_2), nl)
\end{array}
$$

$[\![\texttt{Cmpi}\ (i, r, nl)]\!] -> \texttt{Cmpi}\ (i, (\texttt{R\_pseudo}\ r), nl)$
$[\![\texttt{Neg}\ (r, nl)]\!] -> \texttt{Neg}\ ((\texttt{R\_pseudo}\ r), nl)$
$[\![\texttt{Jmp}\ nl]\!] -> \texttt{Jmp}\ nl$
$[\![\texttt{Jcc}\ (cc, l_{ok}, l_{ko})]\!] = \texttt{Jcc}\ (cc, l_{ok}, l_{ko})$
$[\![\texttt{Label}\ (s, comment, nl)]\!] = \texttt{Label}\ (s, comment, nl)$
$[\![\texttt{Ret}]\!] = \texttt{Ret}$
$[\![\texttt{Setcc}\ (cc, r, nl)]\!] = \texttt{Set}\ (cc, (\texttt{R\_pseudo}\ r), nl)$
$[\![\texttt{Call}]\!]$ et $[\![\texttt{Print}]\!]\ldots$ voir tableau suivant.

FIGURE 12.10 – Traduction RTL → ERTL des instructions (début)

Le cas de la division et du modulo sont quasiment identiques. En effet, l'instruction réelle assembleur réalisant ces deux opérations est la même. Seul diffère le registre dans lequel aller chercher le résultat : `%rax` pour la division et `%rdx` pour le modulo. Nous voyons ainsi apparaître des registres physiques. Bien entendu, dans l'implantation nous gérerons ces deux cas en même temps. Ces deux cas nécessitent l'extension de `%rax` dans `%dax`.

À l'exception de nouvelles instructions insérées, les instructions ERTL et RTL partagent bien les mêmes étiquettes de graphe (nl) et étiquettes de sauts (`Label`).

Le cas de l'appel de fonction est plus compliqué et nécessite quelques explications préalables. En premier lieu, il convient de gérer le passage des arguments à l'appelé. L'ABI System V impose un certain nombre d'obligations lors d'un appel de fonction. Respecter ces consignes est important pour pouvoir appeler du code compilé par un autre compilateur, en particulier les fonctions de la bibliothèque standard de l'OS.

1. En premier lieu, les 6 premiers arguments sont passés dans les registres `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` et `%r9`, et dans cet ordre. Ainsi, il faut commencer par générer les transferts des pseudo-registres contenant les 6 premiers arguments vers les registres physiques. On conssidère que la liste ci-dessus des registres dédiés est connue du compilateur sous le nom $\mathcal{R}_{params}$.
2. S'il reste des arguments, ces derniers doivent alors être passés sur la pile, empilés de droite à gauche par des instructions `Push`.
3. L'instruction d'appel peut ensuite avoir lieu.
4. Le protocole d'appel indique également que la valeur de retour de l'appelé doit être placée dans le registre `%rax`. Comme l'instruction RTL `Call` comporte le pseudo-registre où stocker le résultat de l'appel, il convient de générer un transfert de `%rax` dans ce pseudo-registre.
5. Pour terminer, dans le cas où l'on a passé des arguments sur la pile, le pointeur de pile `%rsp` aura été décrémenté par les instructions `Push` successives. Il convient alors de le restaurer après l'appel. On pourrait générer autant de `Pop` qu'il y a eu de `Push`, mais ce serait inefficace : autant directement ajouter à `%rsp` le déplacement induit par les arguments empilés. Comme notre langage ne travaille que sur des entiers que nous avons choisis sur 64 bits, la correction de pile sera $n \times 8$ octets s'il y a eu n arguments passés par la pile. Bien entendu, dans le cas où aucun argument n'aurait été passé par la pile, on ne veut pas émettre un `add $0,%rsp`.

Tout ceci nous donne la forme du code à produire mais il reste une subtilité. Comme nous l'avions vu dans la section 11.3, la pile doit être alignée sur 16 octets aux sites d'appel (au moment de l'instruction `call`). Or nous n'avons pas pris cette contrainte en compte pour le moment ! Mais, le pouvons-nous ? La réponse est bien entendu non car nous ignorons l'état réel de la pile : il nous manque la connaissance de quels pseudo-registres seront placées en pile par manque de registres physiques ! Et cette information ne sera connue qu'à l'issue de l'allocation de registres (passe ultérieure). Qu'à ce que cela ne tienne, nous allons débuter notre sous-graphe représentant l'appel par une instruction vide, `Nop` que nous remplacerons ultérieurement par une décrémentation de `%rsp` si besoin. Ce nœud sera facile à trouver : c'est le premier du code généré pour l'appel de fonction. Et dans le cas où aucune correction serait nécessaire, ce nœud restera un `Nop` et sera éliminé lors de la passe finale de compilation (ainsi, pas d'instruction inutile).

Pour illustrer ces propos examinons deux exemples d'appels de fonction et comparons les codes à générer. Dans le premier cas, moins de 6 arguments sont fournis : ils sont tous passés par des registres. Dans le second, il y a 8 arguments.

```
main () : int
begin
  int a = 8 ;
  int b = 6 ;
  int c = 4 ;
  return min3 (a, b, c) ;
end

...
34: nop >>> 62
62: mov #19,%rdi >>> 61
61: mov #20,%rsi >>> 60
60: mov #21,%rdx >>> 59
59: call _min3 >>> 58
58: mov %rax,#18 >>> 33
...
```

```
main () : int
begin
  int a = 8 ;
  int b = 6 ;
  int c = 4 ;
  print (min8 (a, b, c, 20, 21, 22, 23, 24)) ;
end

...
33: nop >>> 81
81: mov #24,%rdi >>> 80
80: mov #25,%rsi >>> 79
79: mov #26,%rdx >>> 78
78: mov #27,%rcx >>> 77
77: mov #28,%r8 >>> 76
76: mov #29,%r9 >>> 75
75: push #30 >>> 74
74: push #31 >>> 73
73: call _min8 >>> 72
72: mov %rax,#23 >>> 71
71: addi $16,%rsp >>> 32
...
```

Dans les deux cas, on remarque le nœud initial `Nop` qui servira d'éventuelle correction de pile pour l'alignement sur 16 octets. À gauche, les pseudo-registres correspondant à `a`, `b` et `c` sont bien transférés dans les registres physiques, immédiatement suivis de l'appel. Comme aucun argument n'a été transmis via la pile, aucune correction de cette dernière n'est nécessaire. À droite, 2 instructions `Push` viennent empiler les 2 arguments restants. Ces deux `Push` sont bien annulés en fin d'appel par un addition de $2 \times 8 = 16$ octets à `%rsp`. Dans les deux cas, `%rax` est transféré dans le pseudo-registre destination de l'expression d'appel de fonction.

Nous pouvons désormais donner la règle de traduction d'un appel de fonction. La traduction de `Print` sera un encodage manuel du cas général d'un appel avec une petite subtilité. En effet, comme `printf` est une fonction à nombre variable d'arguments, il faudra indiquer dans `%rax` le nombre d'arguments transmis hors registres. Comme nous n'affichons qu'une seule valeur, ce sera toujours 0. De plus, comme indiqué en section 12.5.2, l'adresse du format sera calculée relativement au pointeur d'instruction (mécanisme que nous ne détaillerons pas et qu'il faut accepter tel quel ici). Notons que, comme nous nommons les nœuds intermédiaires générés, pour retrouver la structure du code généré il convient de lire les règles de « bas en haut ». Nous notons $\| l \|$ la longueur de la liste l.

$[\![$`Call` $(rr, f, args, nl)]\!] =$
  Soient a_{regs} les 6 premiers arguments de $args$,
  Soient a_{stack} les éventuels restants,
  Soit $nb_{stack} = \| a_{stack} \|$,
  Soit $nb_{regs} = \| a_{regs} \|$,
  Soit $popl =$
    Si $nb_{stack} > 0$ alors `add_node` (`Addi` $((nb_{stack} \times 8), ($`R_phys rsp`$), nl)$)
    Sinon nl
  Soit $ml =$ `add_node` (`Mov` $(($`R_phys rax`$), ($`R_pseudo` $rr), popl)$)
  Soit $cl =$ `add_node` (`Call` $(f, nb_{regs}, nb_{stack}, ml)$)
  Soit $pushl =$ `push_args` (a_{stack}, cl)
  Soit $toregsl =$ `args_to_regs` $(\mathcal{R}_{params}, a_{regs}, pushl)$
  `Nop` $toregsl$

$[\![$`Print` $(f, r, nl)]\!] =$
Soit lab une nouvelle étiquette de chaîne associée à `"%ld\n"`,
  Soit $cl =$
    `add_node` (`Mov` $(($`R_pseudo` $reg), ($`R_phys rsi`$),$
      (`add_node` (`Lea_rip` $(lab, ($`R_phys rdi`$),$
        (`add_node` (`Movi` $(0, ($`R_phys rax`$), ($`add_node` (`Call` $(f, 3, 0, nl))))))))))$
  `Nop` cl

FIGURE 12.11 – Traduction RTL → ERTL des instructions (fin)

Pour rappel, le nom de la fonction `printf` embarqué dans l'instruction `Print` a été construit pendant la traduction en RTL afin de s'ajuster aux conventions de nommage des symboles globaux en fonction de l'OS hôte. Le fait de l'avoir mémorisé dans l'instruction permet de ne plus se soucier de ce détail technique.

Il reste à donner les deux fonctions auxiliaires `push_args` et `args_to_regs` qui permettent, respectivement d'empiler les arguments excédentaires (par rapport au nombre de registres physiques dédiés au passage d'arguments) et de transférer les 6 premiers éventuels arguments dans les registres physiques dédiés au passage d'arguments. Chacune retourne l'étiquette de graphe du premier nœud de ces transferts.

`push_args` $([], nl)$	$=$	nl
`push_args` $([r_1; \ldots], nl)$	$=$	`add_node` (`Push` $(($`R_pseudo` $r_1), ($`push_args` $([\ldots], nl))))$

FIGURE 12.12 – Empilement des arguments excédentaires

Notons que la fonction `push_args` est la seule à générer des instructions `Push`. Toutes les autres manipulations au travers de la pile seront faits par des accès relatifs à `%rbp`.

`args_to_regs` $([], \_, nl)$	$=$	nl
`args_to_regs` $(\_, [], nl)$	$=$	nl
`args_to_regs` $([\phi_1; \ldots], [r_1; \ldots], nl)$	$=$	`add_node` (`Mov` $(($`R_pseudo` $r_1), \phi_1, ($`args_to_regs` $(\ldots, \ldots, nl))))$

FIGURE 12.13 – Transfert des arguments dans les registres dédiés

La fonction `args_to_regs` prend en premier argument la liste des registres physiques (ϕ pour « physique ») dédiés au passage d'arguments et en second argument la liste des pseudo-registres contenant les arguments effectifs. À l'issue de l'exécution de cette fonction, si la première liste est vide mais pas la seconde, alors il reste des arguments à passer par la pile. Si à l'inverse la seconde est vide alors tous les arguments tiennent dans les registres physiques.

La traduction des instructions RTL vers ERTL est désormais terminée, il reste à traduire les définitions de fonction. Bien évidemment, outre la traduction des instructions qui composent le corps d'une fonction, toute la mécanique de passage des arguments lors d'un appel va trouver un écho du côté de l'entrée dans une fonction.

1. Il faut commencer par insérer une pseudo-instruction d'allocation de *stack frame*. Cette pseudo-instruction sera donc la toute première instruction de la fonction. Elle sera remplacée dans la prochaine passe par une allocation réelle sur pile une fois que nous connaîtrons la taille nécessaire.
2. Selon l'ABI System V, un certain nombre de registres doivent être sauvegardés par l'appelé en début de fonction, puis restaurés en fin. Il s'agit des registres (appelés *callee saved*) `%rbx`, `%r12`, `%r13`, `%r14` et `%r15`. Cette liste est connue du compilateur sous le nom $\mathcal{R}_{calleeS}$. Le registre `%rbp` doit aussi être sauvegardé par l'appelé. Néanmoins nous ne le mettrons pas dans la liste $\mathcal{R}_{calleeS}$ car il va être géré spécifiquement. En effet, comme décrit dans le chapitre 11.3 ce dernier sert à accéder aux paramètres et aux variables locales. Il doit être sauvegardé en tout début de fonction, ce qui sera inséré dans le prélude lors de la prochaine passe (production du LTL).
3. Ensuite, il faut copier les arguments transmis par l'appelant dans les pseudo-registres servant à représenter les paramètres. Les 6 premiers (au maximum) proviennent des registres physiques, le reste est à récupérer dans la pile. La suite du code à exécuter sera la traduction de la première instruction RTL de la fonction. Nous verrons d'ici peu comment retrouver son étiquette (sans trahir un secret, du fait du partage des étiquettes entre les graphes, ce sera trivial).

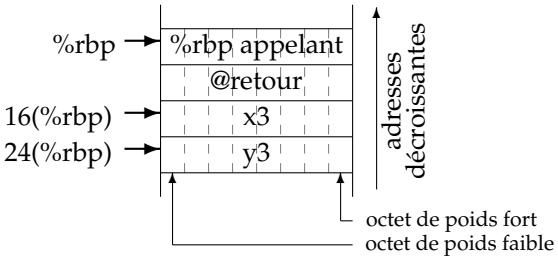
La sauvegarde des registres à la charge de l'appelé peut être allégée. D'une part, dans le cas de la fonction `main`, comme il n'y a pas réellement d'appelant, on peut s'épargner cette sauvegarde. D'autre part, les compilateurs intelligents sont capables d'effectuer des analyses inter-procédurales pour tenter de vérifier quels registres *callee saved* sont effectivement utilisés par la fonction et celles qu'elle appelle et de ne sauvegarder / restaurer que ceux-ci (quand l'analyse peut-être faite). Pour simplifier la présentation, nous sauvegarderons tous les registres sauf dans le cas de la fonction `main`.

Pour illustrer la mise en place du prélude d'une fonction, nous reprenons les exemples des fonctions `min3` et `min8` utilisés précédemment, mais vus sous l'angle des fonctions et non de leur appel.

```
min3 (int x, int y, int z) : int
begin
  if x < y && x < z then
    return x ;
  else
    if y < z then
      return y ;
    endif
  endif
    return z ;
end

64: frame start >>> 63
63: mov %rbx,#34 >>> 62
62: mov %r12,#35 >>> 61
61: mov %r13,#36 >>> 60
60: mov %r14,#37 >>> 59
59: mov %r15,#38 >>> 58
58: mov %rdi,#1 >>> 57
57: mov %rsi,#2 >>> 56
56: mov %rdx,#3 >>> 29
29: lbl_7:   # min3_body >>> 28
...
```

```
min3 (int x1, int y1, int z1, int x2, int y2,
      int z2, int x3, int y3) : int
begin
  ...
end


59: frame start >>> 58
58: mov %rbx,#32 >>> 57
57: mov %r12,#33 >>> 56
56: mov %r13,#34 >>> 55
55: mov %r14,#35 >>> 54
54: mov %r15,#36 >>> 53
53: mov %rdi,#1 >>> 52
52: mov %rsi,#2 >>> 51
51: mov %rdx,#3 >>> 50
50: mov %rcx,#4 >>> 49
49: mov %r8,#5 >>> 48
48: mov %r9,#6 >>> 47
47: mov 16(%rbp),#7 >>> 46
46: mov 24(%rbp),#8 >>> 29
29: lbl_7:   # min8_body >>> 28
...
```

Dans les deux cas le graphe débute par une pseudo-instruction d'allocation de *stack frame* suivie de la sauvegarde des 5 registres `%rbx` à `%r15` dans des pseudo-registres. Ensuite, la fonction de gauche récupère ses 3 arguments dans `%rdi`, `%rsi` et `%rdx` pour les stocker dans 3 pseudo-registres. Du côté de la fonction de droite, la récupération des 6 premiers arguments se fait dans les 6 registres dédiés, suivie de deux accès à la pile avec un décalage de 16 octets par rapport à `%rbp` pour accéder à `x3` et un décalage de 24 pour `y3`.



Si nous reprenons le second schéma de la pile de la section 11.3, avec `%rbp` sauvegardé tel qu'il le sera après la transformation en LTL, les paramètres sont bien accessibles par un décalage positif, débutant à 16 puisque `%rbp` aura été sauvegardé (8 octets) et l'adresse de retour aura été empilée par l'instruction d'appel (encore 8 octets). De plus, puisque notre langage ne manipule que des entiers sur 64 bits, chaque paramètre est décalé du précédent de 8 octets (16 + 8 = 24 pour `y3`). Comme nous avons indiqué précédemment, l'ABI System V impose que les arguments soient empilés de droite à gauche, donc `y3` est empilé avant `x3` et se trouve donc « plus bas » (graphiquement parlant) dans la pile. Notons que la récupération des arguments sur la pile ne doit pas être faite par des instructions `Pop` mais par de simples `Mov`. En effet, `Pop` aurait pour effet de dépiler le sommet de la pile, ce qui n'est absolument pas où se trouvent les arguments. De plus, cela aurait pour effet de modifier le pointeur de pile `%rsp`, détruisant la *stack frame*.

Après l'émission du prélude de la fonction sont émises les instructions composant son corps. Puisque nous partageons les étiquettes de graphe entre RTL et ERTL, l'étiquette de la première instruction **du corps** (pas du prélude) de la fonction ERTL reste celle de la première instruction de la fonction RTL. Le prélude décrit ci-dessus s'insère donc avant cette étiquette et son premier nœud deviendra le point d'entrée de la fonction ERTL.

Il faut finalement générer le postlude de la fonction. Celui-ci va présenter une certaine symétrie par rapport au prélude puisqu'il devra remettre la pile et les registres sauvegardés « en état ».

1. La valeur de retour doit être copiée dans le registre `%rax`.
2. Les registres physiques sauvegardés doivent ensuite être restaurés depuis les pseudo-registres.
3. Une pseudo-instruction de désallocation de *stack frame* doit être insérée.

Puisque nous avons pris soin de toujours générer une pseudo-instruction `Label` à chaque destination de saut, dans le RTL, à l'étiquette du nœud de postlude (qui sert de destination des sauts induits par des `Ret`) se trouvera forcément une telle pseudo-instruction (suivie du `Ret` final). Les instructions ERTL à rajouter le seront après cette étiquette. Mais puisque nous avons déjà traduit toutes les instructions RTL en ERTL, forcément le `Label` et le `Ret` finaux existeront déjà dans le graphe ERTL. Il faudra donc faire en sorte que le nœud `Label` n'ait plus pour successeur le `Ret` directement, mais le premier nœud du postlude ERTL. Nous écraserons donc son ancien arc après-coup, après avoir généré le postlude et nous ferons en sorte que la dernière instruction du postlude soit justement le nœud ERTL contenant le `Ret`.

Reprenons l'exemple de la fonction `min3` ci-dessus pour illustrer le postlude généré. Notons que ce prélude sera le même pour la fonction `min8` (au niveau ERTL car il pourra changer par la suite après transformation finale en LTL).

```
2: lbl_1:   # fun postlude >>> 71
71: mov #4,%rax >>> 70
70: mov #34,%rbx >>> 69
69: mov #35,%r12 >>> 68
```

```
68: mov #36,%r13 >>> 67
67: mov #37,%r14 >>> 66
66: mov #38,%r15 >>> 65
65: frame end >>> 1
1: ret >>>
```

Le pseudo-registre dédié à la valeur de retour de l'appel (mémorisé dans l'instruction RTL `Call`) est transféré dans `%rax`. Puis on retrouve la restauration des 5 registres physiques à la charge de l'appelé, qui va bien chercher les valeurs dans les pseudo-registres utilisés dans le prélude (`#34` dans `%rbx`, `#35` dans `%r12`, etc). Cela implique bien entendu qu'il faille mémoriser dans quel pseudo-registre on sauvegarde un registre physique. Cette mémorisation sera une liste d'association (que l'on nommera ρ) retournée par la fonction chargée de la sauvegarde. Finalement la pseudo-instruction de désallocation de *stack frame* apparaît suivie du `Ret` final.

Nous pouvons désormais donner la fonction de traduction d'une définition de fonction vers ERTL, qui part d'un graphe ERTL initialement vide. Elle prend en entrée la représentation RTL d'une fonction f telle que décrite en 12.4.3 : $\{ f; [r_1, \ldots; r_n]; rr; G; rl; pl \}$ où $r_1, \ldots; r_n$ sont les pseudo-registres représentant les paramètres, rr (*return register*) est celui dédié à la valeur retournée, rl (*root label*) est l'étiquette du premier nœud de la fonction et pl (*postlude label*) est celle de son postlude. Comme pour la traduction d'une fonction vers RTL, nous nommons les sous-graphes intermédiaires (sinon on n'aurait qu'un seul immense et illisible enchaînement d'appels de fonctions), donc l'ordre de lecture est inversé au sein du prélude et du postlude.

$[\![\{ f; [r_1, \ldots; r_n]; G; rr; rl; pl \}]\!] =$
  Soit G' un graphe vide (implicitement le graphe courant,)
  Pour tous les couples (lab, i) du graphe RTL G,
    Ajouter $(lab, [\![i]\!])$ à G'
  Soient r_{regs} les 6 premiers pseudo-registres de $[r_1, \ldots; r_n]$,
  Soient r_{stack} les éventuels restants,
  Soit $gprmsl$ = `regs_to_args` $(r_{regs}, \mathcal{R}_{params},$ `stack_to_args` $(r_{stack}, rl))$
  Soit $(\rho, sregsl)$ =
    Si $f \neq$ "*main*" alors `save_regs` $(\mathcal{R}_{calleeS}, gprmsl)$
    Sinon $([], gprmsl)$
  Soit $frallocl$ = `add_node` (`Frame_start` $sregsl$)
  Soit $retl$ = étiquette du successeur de pl dans G,
  Soit $frfreel$ = `add_node` (`Frame_end` $retl$)
  Soit $rregsl$ = `restore_regs` $(\rho, frfreel)$
  Soit $resl$ = `add_node` (`Mov` ((`R_pseudo` rr), (`R_phys rax`), $rregsl$))
  Rechercher le nœud ERTL d'étiquette pl
    et y mettre $resl$ comme successeur
  $\{ f; G'; frallocl; 0 \}$

Cette fonction s'appuie sur 4 fonctions auxiliaires pour d'une part gérer la récupération des paramètres et leur transfert vers les pseudo-registres adéquats de la fonction :

- `regs_to_args` qui permet de transférer les arguments des registres physiques vers les pseudo-registres,
- `stack_to_args` qui permet de transférer les arguments de la pile vers les pseudo-registres,

et d'autre part gérer la sauvegarde / restauration des registres physiques à la charge de l'appelé :

- `save_regs` qui permet de sauvegarder les registres physiques à la charge de l'appelé dans des pseudo-registres,

- `restore_regs` qui restaure les registres physiques sauvegardés.

```
regs_to_args ([], _, nl)                 =  nl
regs_to_args (_, [], nl)                 =  nl
regs_to_args ([φ1; ...], [r1; ...], nl)  =  add_node (Mov (φ1, (R_pseudo r1), (regs_to_args (..., ..., nl))))
```

```
Soit o le décalage dans la pile, initialement égal à 16,
stack_to_args ([], nl) = nl
stack_to_args ([r1; ...], nl) =
  Soit o' = o
  o ← o + 8
  add_node (Stack_prm_read ((R_pseudo r1), o', (stack_to_args (..., nl))))
```

La fonction `stack_to_args` est écrite de manière un peu cavalière, avec un effet de bord. On se pardonnera cette liberté. Le décalage permettant d'accéder aux paramètres empilés est initialement de +16 puisque, comme expliqué sur le schéma de la pile, l'adresse de retour et la sauvegarde de `%rbp` sont déjà sur la pile une fois que l'on est entré dans la fonction. Pour accéder au paramètre suivant, on augmente ce décalage de 8 octets, la taille des entiers que notre langage permet de manipuler. Nous voyons que nous générons à chaque fois une pseudo-instruction `Stack_prm_read` qui sera ultimement traduite en un accès relatif à `%rbp` avec un décalage idoine.

```
save_regs ([], nl) = nl
save_regs ([φ1; ...], nl) =
  Soit r un nouveau pseudo-registre,
  Soit (ρ, nl') = save_regs (..., nl)
  Soit l = add_node (Mov (φ1, (R_pseudo r), nl))
  (((φ1, r) :: ρ), l)
```

La fonction `save_regs` construit et retourne la liste d'association ρ permettant de savoir dans quel pseudo-registre chaque registre physique a été sauvegardé ainsi que l'étiquette de graphe correspondant à la première instruction de sauvegarde des registres.

```
restore_regs ([], nl) = nl
restore_regs ([(φ1, r1); ...], nl) = add_node (Mov ((R_pseudo r1), φ1, (restore_regs (..., nl))))
```

La fonction `restore_regs` effectue des `Mov` symétriques à `save_regs`. Elle se contente de parcourir la liste d'association ρ construite par `save_regs` pour retrouver les registres à restaurer.

12.5.4 Implantation en OCaml

Dans l'implantation OCaml, il est nécessaire de gérer plus explicitement la création d'étiquettes de chaînes de caractères pour le format de `printf` utilisé pour compiler l'instruction `Print`. Puisque l'on affiche toujours un seul entier, ce format est invariablement `"%d\n"`. Nous pourrions le générer en dur à la fin de tout fichier assembleur. Pour être moins brouillon, le choix est fait de se définir une fonction qui mémorise toutes les chaînes rencontrées et leur associe une étiquette fraîche. À la fin de la compilation, on ira récupérer cet ensemble et on générera des définitions de chaînes de caractères assembleur. Le couple de fonction `new_format_string` et `get_format_strings` sert à cet effet.

```
let (new_format_string, get_format_strings) =
  let strings_map = ref [ ] in
  let count = ref 0 in
  (fun format →
```

```
    try List.assoc format !strings_map
    with Not_found →
      incr count ;
      let label = "lstr" ^ (string_of_int !count) in
      strings_map := (format, label) :: !strings_map ;
      label),
  (fun () → !strings_map)
```

Afin d'isoler les 6 premiers éléments d'une liste et les restants (utilisé pour la gestion du passage d'arguments dans les registres et dans la pile), on se définit une fonction utilitaire sur les listes permettant de récupérer la liste des *nb* premiers éléments d'une liste *l* et celle des éléments restants.

```
let rec split_list l nb =
  if nb < 0 then failwith "split_list"
  else if nb = 0 then ([ ], l)
  else
    match l with
    | [ ] → ([ ], [ ])
    | (h :: q) →
        let (left, right) = split_list q (nb − 1) in
        ((h :: left), right)
```

Maintenant que nous avons écrit ces deux fonction auxiliaires, nous pouvons désormais donner l'implémentation des fonctions de traduction proprement dites.

```
let push_args graph args next =
  let rec rec_push = function
    | [ ] → next
    | h :: q → Graph.add_node graph (ERTL.Push ((Regs.R_pseudo h), (rec_push q))) in
  rec_push args

let pop_args graph rem_params next =
  let offset = ref (RTL.word_size ∗ 2) in
  let rec rec_pop = function
    | [ ] → next
    | h :: q →
        let curr_offset = !offset in
        offset := !offset + RTL.word_size ;
        let next = rec_pop q in
        Graph.add_node graph (ERTL.Stack_prm_read ((Regs.R_pseudo h), curr_offset, next)) in
  rec_pop rem_params

let args_to_regs graph phys_regs args_regs next =
  let rec rec_do phys args =
    match (phys, args) with
    | ([ ], _) | (_, [ ]) → next
    | ((reg :: rem_regs), arg :: rem_args) →
        Graph.add_node graph (ERTL.Mov ((Regs.R_pseudo arg), reg, (rec_do rem_regs rem_args))) in
  rec_do phys_regs args_regs
```

```
let regs_to_args graph init_phys_regs init_pseud_regs next =
  let rec rec_do phys_regs pseud_regs =
    match (phys_regs, pseud_regs) with
    | ([], _) | (_, []) → next
    | ((reg :: rem_regs), pseud :: rem_pseudos) →
        let next = rec_do rem_regs rem_pseudos in
        Graph.add_node graph (ERTL.Mov (reg, (Regs.R_pseudo pseud), next)) in
  rec_do init_phys_regs init_pseud_regs

let do_instr graph asm =
  match asm with
  | RTL.Movi (i, reg, next) → ERTL.Movi (i, (Regs.R_pseudo reg), next)
  | RTL.Mov (reg1, reg2, next) → ERTL.Mov ((Regs.R_pseudo reg1), (Regs.R_pseudo reg2), next)
  | RTL.Addi (i, reg, next) → ERTL.Addi (i, (Regs.R_pseudo reg), next)
  | RTL.Jmp next → ERTL.Jmp next
   ...
  | RTL.Div (reg1, reg2, next) | RTL.Mod (reg1, reg2, next) →
      ERTL.Mov ((Regs.R_pseudo reg2), (Regs.R_phys Regs.rax),
      Graph.add_node graph (ERTL.Cqto
      (Graph.add_node graph (ERTL.Div ((Regs.R_pseudo reg1),
      match asm with
      | RTL.Div _ → Graph.add_node graph (ERTL.Mov ((Regs.R_phys Regs.rax), (Regs.R_pseudo reg2), next))
      | _ → Graph.add_node graph (ERTL.Mov ((Regs.R_phys Regs.rdx), (Regs.R_pseudo reg2), next)))))))
  | RTL.Call (ret_reg, var, args, next) →
      let (args_for_regs, args_for_stack) = split_list args Regs.nb_for_params in
      let nb_args_stacked = List.length args_for_stack in
      let pop_node =
        if nb_args_stacked > 0 then (
          Graph.add_node graph
            (ERTL.Addi ((Int64.of_int (nb_args_stacked * RTL.word_size)), (Regs.R_phys Regs.rsp), next))
        )
        else next in
      let res_mov_nod =
        Graph.add_node graph (ERTL.Mov (Regs.R_phys Regs.rax, (Regs.R_pseudo ret_reg), pop_node)) in
      let call_node = Graph.add_node graph (ERTL.Call (var, nb_args_stacked, res_mov_nod)) in
      let start_push = push_args graph args_for_stack call_node in
      let start_args_to_regs_node = args_to_regs graph Regs.for_params args_for_regs start_push in
      ERTL.Nop start_args_to_regs_node
  | RTL.Print (printf_name, reg, next) →
      let format_string_lbl = new_format_string "%ld\n" in
      let call_node =
        Graph.add_node graph (ERTL.Mov ((Regs.R_pseudo reg), (Regs.R_phys Regs.rsi),
        (Graph.add_node graph (ERTL.Lea_rip (format_string_lbl, (Regs.R_phys Regs.rdi),
        (Graph.add_node graph (ERTL.Movi (Int64.zero, (Regs.R_phys Regs.rax),
        (Graph.add_node graph (ERTL.Call (printf_name, 3, 0, next ))))))))))  in
      ERTL.Nop call_node
```

Pour terminer vient enfin l'implantation de la traduction d'une définition de fonction, accompagnée des fonctions auxiliaires s'occupant de la gestion des registres *calle saved* et de la récupération des paramètres effectifs pour les placer dans les pseudo-registres définis lors du passage en RTL.

```
let save_callee_registers graph regs next =
  let rec rec_work = function
```

```
    | [ ] → ([ ], next)
    | h :: q →
        let reg = RTL.new_reg () in
        let (rem_env, next) = rec_work q in
        let new_node_lab = Graph.add_node graph (ERTL.Mov (h, (Regs.R_pseudo reg), next)) in
        (((h, reg) :: rem_env), new_node_lab) in
  rec_work regs

let restore_callee_registers graph sav_map next =
  let rec rec_work = function
    | [ ] → next
    | (phys_reg, saved_in) :: q →
        let next = rec_work q in
        Graph.add_node graph (ERTL.Mov ((Regs.R_pseudo saved_in), phys_reg, next)) in
  rec_work sav_map

let retrieve_params graph fdef_params next =
  let (args_from_regs, args_from_stack) = split_list fdef_params Regs.nb_for_params in
  regs_to_args graph Regs.for_params args_from_regs (pop_args graph args_from_stack next)

let do_fundef f_def =
  let graph = ref (Graph.G.empty) in
  Graph.G.iter
    (fun rtl_label asm →
      let asm' = do_instr graph asm in
      graph := Graph.G.add rtl_label asm' !graph)
    f_def.RTL.fdef_graph ;
  let get_params_n = retrieve_params graph f_def.RTL.fdef_params f_def.RTL.fdef_entry in
  let (saved_map, regs_save_n) =
    if f_def.RTL.fdef_name.Ast.v_name ≠ToRTL.main_name then (
      save_callee_registers graph Regs.callee_saved get_params_n)
    else ([ ], (Graph.add_node graph (ERTL.Nop get_params_n))) in
  let fr_alloc_n = Graph.add_node graph (ERTL.Frame_start regs_save_n) in
  let ret_node_label =
    (match Graph.G.find f_def.RTL.fdef_postlude f_def.RTL.fdef_graph with
    | RTL.Label (_, _, next) → next
    | _ → assert false) in
  let dealloc_frame_n = Graph.add_node graph (ERTL.Frame_end ret_node_label) in
  let regs_rest_n = restore_callee_registers graph saved_map dealloc_frame_n in
  let mov_ax_n =
    Graph.add_node graph
      (ERTL.Mov ((Regs.R_pseudo f_def.RTL.fdef_ret_reg), (Regs.R_phys Regs.rax), regs_rest_n)) in
  (match Graph.G.find f_def.RTL.fdef_postlude !graph with
  | ERTL.Label (lab, comment, _) →
      graph := Graph.G.add f_def.RTL.fdef_postlude (ERTL.Label (lab, comment, mov_ax_n)) !graph
  | _ → assert false) ;
  { ERTL.fdef_name = f_def.RTL.fdef_name ; ERTL.fdef_graph = !graph ;
    ERTL.fdef_entry = fr_alloc_n ; ERTL.fdef_frame_size = 0 }
```

12.5.5 Résumé

Lors du passage de RTL vers ERTL, nous avons obtenu une nouvelle représentation intermédiaire sous forme de graphe par insertion de nouvelles instructions entre celles présentes dans le RTL. Des registres physiques ont commencé à faire leur apparition, côtoyant des pseudo-registres toujours pré-

sents en grand nombre. Les conventions d'appel de fonction imposées par l'ABI System V ont été explicitées : on passe certains paramètres dans des registres physiques, le reste par la pile, dans un ordre bien défini. La valeur retournée par la fonction est explicitement transférée dans le registre `%rax`. Des pseudo-instructions d'allocation et de désallocation de (future) *stack frame* on été insérées en début et en fin de fonction. Pour terminer, la gestion de la correction de pile pour garantir un alignement sur 16 octets aux sites d'appel reste en suspens, mais une instruction « vide » a été insérée à cet effet pour être éventuellement remplacée lors de la traduction vers LTL.

12.6 Allocation de registres

Notre RTL est désormais assez proche d'intructions assembleur réelles. Il reste néanmoins un problème de taille : les pseudo-registres ! En effet, nos instructions opèrent encore majoritairement sur des registres qui n'existent pas dans la machine cible, et qui sont infiniment plus nombreux que ceux réellement disponibles dans le processeur. La tâche de l'allocation de registres est de tenter de remplacer les pseudo-registres par les registres physiques.

Les pseudo-registres que nous avons générés aveuglement ne sont pas tous utilisés durant toute l'exécution de la fonction. Il doit bien être possible de partager un même registre physique pour représenter plusieurs pseudo-registres. . . Considérons le petit programme ci-dessous qui échange 2 variables et le ERTL généré.

```
main () : int                ...
begin                        13: loadi $6,#1 >>> 12
  int x = 6 ;                12: loadi $7,#2 >>> 11
  int y = 7 ;                11: mov #1,#8 >>> 10
  int tmp ;                  10: mov #8,#3 >>> 9
  tmp = x ;                  9: mov #2,#7 >>> 8
  x = y ;                    8: mov #7,#1 >>> 7
  y = tmp ;                  7: mov #3,#6 >>> 6
  return 0 ;                 6: mov #6,#2 >>> 5
end                          5: loadi $0,#5 >>> 4
                             4: mov #5,#4 >>> 18
                             18: mov #4,%rax >>> 17
                             ...
```

Nous utilisons 8 pseudo-registres, mais on se rend bien compte qu'il est possible de se contenter de 3 registres, par exemple avec :

```
loadi $6,%rbx
loadi $7,%rcx
mov %rbx, %rax
mov %rcx, %rbx
mov %rax, %rcx
mov $0, %rax
```

Sur cet exemple simpliste, l'allocation est facile et permet de transformer tous les pseudo-registres en registres physiques. On dit parfois que l'on « réalise » un pseudo-registre par un registre. Malheureusement, il arrive que le nombre de registres physiques ne soit pas suffisant. Il suffit par exemple d'avoir une expression utilisant plus de variables qu'il n'existe de registres disponibles dans le processeur (par exemple un simple `x11 = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10`). Dans ce cas,

il faudra traduire des pseudo-registres en emplacements mémoire sur la pile. En réalité, il n'y a même pas besoin d'un programme avec autant de variables car il ne faut pas oublier que certains registres (les *callee saved*) doit être sauvegardés (ou non modifiés, ce qui ne nécessite pas de les sauvegarder mais interdit leur utilisation).

Le problème de recycler un registre physique pour représenter plusieurs pseudo-registres se pose si à un point de programme, ces pseudo-registres contiennent tous des valeurs nécessaires pour la suite de l'exécution. En effet, toutes ces valeurs étant requises elles ne peuvent coexister dans un même registre physique (elles s'écraseraient les unes les autres). On dit que les pseudo-registres sont *vivants* à ce point de programme. Une autre manière de le dire est qu'un registre (ce qui se dit aussi d'une variable) est vivant à un point de programme si sa valeur va être lue plus tard avant d'être écrasée (écrite). La *durée de vie* est alors l'ensemble des points de programme où un registre est vivant.

L'allocation de registre va devoir calculer la *durée de vie* de chaque temporaire (registre pseudo ou physique) sur le ERTL. Ensuite, sur la base de ces durées de vie, elle va établir un *graphe d'interférence* décrivant quel registre est vivant en même temps que quels autres. Des registres vivants en même temps (on dit qu'ils *interfèrent*) ne pourront alors pas être représentés par un même registre physique.

Une fois ce graphe établi, il va falloir trouver un moyen d'assigner à chaque pseudo-registre un registre physique différent de ceux utilisés par les registres (pseudo ou physique) avec lesquels il interfère. Nous verrons que ce problème peut se traiter comme un coloriage de graphe. Nous verrons que le coloriage peut être impossible avec le nombre de registres physiques disponibles. Comme précisé précédemment, on utilisera alors des emplacements mémoire pour certains pseudo-registres.

Une fois le coloriage terminé, chaque pseudo-registre se sera vu assigné un registre physique ou un emplacement en pile et il sera possible de traduire le ERTL vers la représentation intermédiaire finale, LTL pour ensuite émettre le code assembleur.

12.6.1 Calcul de durée de vie

Ce calcul se déroule en deux étapes. Il faut tout d'abord déterminer pour chaque instruction quels sont les temporaires *lus* et quels sont les temporaires *écrits*. Nous donc parlons ici de registres physiques ou de pseudo-registres. On appelle cette analyse *calcul des def and use*. On note $def\,(l)$ l'ensemble des registres écrits par l'instruction d'étiquette l et $use\,(l)$ celui des registres lus.

Cette analyse ne présente pas de difficultés pour la majorité des instructions. Par exemple, dans `add #3,#5` dont l'effet est #5 ← #5 + #3, les registres `#3` et `#5` sont lus alors que seul `#5` est écrit. Deux cas sont plus subtiles, celui de `Call` et celui de `Ret` que nous détaillons après la figure 12.6.1.

$def\,$(`Cqto _`) $= \{$`%rdx`$\}$	$use\,$(`Cqto _`) $= \{$`%rax`$\}$
$def\,$(`Movi/Lea_rip` $(\_, r, \_)) = \{r\}$	$use\,$(`Movi/Lea_rip` $(\_, r, \_)) = \emptyset$
$def\,$(`Mov` $(r_1, r_2, \_)) = \{r_2\}$	$use\,$(`Mov` $(r_1, r_2, \_)) = \{r_1\}$
$def\,$(`Addi/Subi/Muli` $(\_, r, \_)) = \{r\}$	$use\,$(`Addi/Subi/Muli` $(\_, r, \_)) = \{r\}$
$def\,$(`Add/Sub/Mul` $(r_1, r_2, \_)) = \{r_2\}$	$use\,$(`Add/Sub/Mul` $(r_1, r_2, \_)) = \{r_1, r_2\}$
$def\,$(`Div` $(r, \_)) = \{$`%rax, %rdx`$\}$	$use\,$(`Div` $(r, \_)) = \{r,$ `%rax, %rdx`$\}$
$def\,$(`Cmp` $(r_1, r_2, \_)) = \emptyset$	$use\,$(`Cmp` $(r_1, r_2, \_)) = \{r_1, r_2\}$
$def\,$(`Cmpi` $(\_, r, \_)) = \emptyset$	$use\,$(`Cmpi` $(\_, r, \_)) = \{r\}$
$def\,$(`Neg` $(r, \_)) = \{r\}$	$use\,$(`Neg` $(r, \_)) = \{r\}$
$def\,$(`Call` $(\_, n, \_, \_)) = \mathcal{R}_{calleeK}$	$use\,$(`Call` $(\_, n, \_, \_))$ = n premiers registres de $\mathcal{R}_{params}$
$def\,$(`Jmp _`) $= \emptyset$	$use\,$(`Jmp _`) $= \emptyset$
$def\,$(`Jcc` $(\_, \_, \_)) = \emptyset$	$use\,$(`Jcc` $(\_, \_, \_)) = \emptyset$
$def\,$(`Label` $(\_, \_, \_)) = \emptyset$	$use\,$(`Label` $(\_, \_, \_)) = \emptyset$
$def\,$(`Ret`) $= \emptyset$	$use\,$(`Ret`) $= \emptyset$

def (`Set` (_, r, _)) = { r }	use (`Set` (_, r, _)) = ∅
def (`Pop` (r, _)) = { r }	use (`Pop` (r, _)) = ∅
def (`Push` (r, _)) = ∅	use (`Push` (r, _)) = { r }
def (`Nop`/`Frame_start`/`Frame_end` _) = ∅	use (`Nop`/`Frame_start`/`Frame_end` _) = ∅
def (`Stack_prm_read` (r, _, _)) = { r }	use (`Stack_prm_read` (r, _, _)) = ∅

FIGURE 12.14 – Calcul de *def and use*

Dans le cas de `Call`, il faut considérer que les registres physiques effectivement utilisés pour passer les arguments sont lus. Dans les faits c'est bien le cas, sauf situation pathologique de l'appel à une fonction qui n'utiliserait pas ses arguments (le déterminer nécessiterait une analyse dite *inter-procédurale*). Pour ce qui est des registres écrits, ce seront tous ceux qui ne sont pas préservés par l'appelé selon l'ABI considérée. Cette liste $\mathcal{R}_{calleeK}$, connue du compilateur, est `%rax`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10` et `%r11`. Notons que `%rbp` n'est pas dedans, même s'il est modifié par un appel, car il est géré à part en tant que *frame pointer*.

Dans le cas de `Ret`, `%rax` est utilisé implicitement par l'appelant (qui récupérera la valeur retournée) ainsi que tous les registres que l'on a sauvegardés de notre côté appelé. En effet, ils servent possiblement à l'appelant qui avait pu y stocker des informations (c'est bien pour cela que l'on les avait sauvegardés puis restaurés).

Pour `Div`, `%rax` et `$rdx` sont écrits puisque l'instruction X86-64`idivq reg` qui sera ultimement utilisée a pour effet de mettre le quotient de (%`rdx` : %`rax`)/*reg* dans `%rax` et le reste dans `%rdx`. De ce fait, `%rdx` et `%rax` sont lus, ainsi que *reg* bien entendu. Dernier cas « non régulier » (également utilisé pour la division et le modulo), `Cqto` qui étend `%rax` dans `%rdx`, donc qui lit `%rax` et écrit `%rdx`.

Une fois que l'on dispose des informations de *def* et *use* il est possible de calculer les variables vivantes à chaque point de programme. On distingue les variables vivantes à *l'entrée* d'un nœud du graphe (donc avant l'exécution de l'instruction) de celles vivantes à la *sortie* du nœud (donc après l'exécution de l'instruction). Ainsi, si l est une étiquette de nœud du graphe ERTL, alors :

- in (l) est l'ensemble des registres vivants sur les arcs aboutissant à l.
- out (l) est l'ensemble des registres vivants sur les arcs sortant de l.

Le calcul de in (l) et out (l) est le plus petit point fixe des deux équations suivantes, mutuellement récursives. Ce point fixe existe car, d'une part le nombre de nœuds du graphe est fini, d'autre part les fonctions in (l) et out (l) sont croissantes (les ensembles de registres ne peuvent que grandir), et sur un treillis complet (ensemble des parties de l'ensemble des registres, muni de l'inclusion).

$$\begin{cases} in\,(l) = use\,(l) \cup (out\,(l) \setminus def\,(l)) \\ out\,(l) = \bigcup_{s \in Succ\,(l)} in\,(l) \end{cases}$$

Le calcul de la solution de ces équations est réalisé par itération : on calcule in (l) et out (l) pour chaque étiquette l du graphe et l'on s'arrête dès que l'on obtient le même résultat qu'à l'itération précédente. Il convient donc de mémoriser les *in* et les *out* (initialement les ensembles vides) de chaque nœud. Dans la pratique, nous les mémoriserons dans une `Map` (ou autre structure associative) associant une étiquette l avec des informations déjà calculées de *def*, *use*, *in*, *out*. On obtient donc l'algorithme naïf suivant :

```
inout (G) =
  change ← vrai
  While change = vrai
    change ← faux
    For chaque étiquette l de G
      in_{n+1} (l) = use_n (l) ∪ (out_n(l) \ def (l))
      out_{n+1} (l) = ⋃_{s∈ Succ (l)} in_n (s)
      If in_{n+1}(l) ≠ in_n(l) ou out_{n+1} (l) ≠ out_n (l)
        change ← vrai
```

Même s'il fonctionne, cet algorithme n'est pas très efficace : il a une complexité en $O(n^4)$ avec n le nombre de sommets du graphe. Il est possible l'améliorer de plusieurs façons. La première consiste à calculer en parcourant le graphe de flot en sens inverse. En effet, on remarque que *out* est une union des *in* de successeurs. Donc si l'on a déjà calculé les informations de *in* et *out* pour ces successeurs, on n'a pas besoin de point fixe. La nécessité du point fixe est introduite par les boucles qui introduisent des arcs qui « remontent » vers des nœuds prédécesseurs.

La seconde est de ne plus considérer les sommets individuellement mais de travailler sur des *blocs de base* (*basic block* en anglais). Un tel bloc est une séquence d'instructions consécutives tel qu'il ne soit possible d'y entrer que par une seule instruction (le *point d'entrée*) et d'en sortir que par une seule instruction également (le *point de sortie*). Autrement dit, si l'une des instructions du bloc est exécutée, alors toutes les autres le seront forcément. Lors de la création des blocs de base, on ne s'intéresse pas à la sémantique des instructions arithmétiques, tout ce qui importe est de détecter les sauts et les étiquettes (`Label` destinations de sauts). L'intérêt de travailler avec des blocs de base est que le graphe de flot devient nettement plus petit, donc le calcul des *in* et *out* est plus rapide. Une fois ces informations obtenues pour les blocs, il est possible de les recalculer au niveau des instructions, en les traitant en sens inverse, à partir du *in* calculé pour le bloc.

Une troisième solution est d'appliquer le principe de l'algorithme de Kildall qui est un mécanisme général d'analyses de flot de données. Basé sur une *working-list* qui représente les instructions dont *in* et *out* n'ont pas encore convergé, il permet à chaque itération de ne recalculer ces informations que sur les instructions dont *in* et *out* sont susceptibles de changer. Lors du traitement d'une telle instruction, si les nouvelles valeurs de *in* et *out* diffèrent de celles de l'étape précédente, alors tous les successeurs de l'instruction sont rajoutés à la *working-list*. Ainsi, on ne re-parcourt pas l'intégralité du graphe de flot de contrôle à chaque itération mais juste les nœuds présents dans la *working-list*.

Pour illustrer la convergence progressive mais lente de l'algorithme naïf, le tableau suivant représente le code ERTL généré pour le programme trivial ci-dessous avec les itérations successives du calcul de *in* et *out*. Volontairement les registres *callee saved* ne figurent afin de ne pas rallonger le code ni le nombre d'étapes nécessaires à l'obtention du point fixe.

```
main () : int
begin
  int x = 10 ;
  while x != 0 do
    x = x - 1 ;
  done
  return x ;
end
```

Les *in* et *out* initiaux ne sont pas représentés puisqu'ils sont tous égaux à l'ensemble vide. De plus, la pour se rendre compte que la dernière colonne représente l'atteinte du point fixe, il a fallu faire une itération de plus pour se rendre compte qu'aucun *in* ni *out* n'avait changé.

Instruction	*use*	*def*	in_1	out_1	in_2	out_2	in_3	out_3	in_4	out_4
(20) `frame start`										
(19) `nop`										
(18) `lbl_5 :`										
(17) `loadi $10,#1`		#1				#1		#1		#1
(16) `lbl_4 :`				#1	#1	#1	#1	#1	#1	#1
(15) `mov #1,#5`	#1	#5	#1		#1	#5	#1	#5	#1	#5
(14) `loadi $0,#6`		#6		#5;#6	#5	#5;#6	#5	#5;#6	#5	#1;#5;#6
(13) `cmp #6,#5`	#5;#6		#5;#6		#5;#6		#5;#6	#1	#1;#5;#6	#1
(12) `je 7`						#1	#1	#1	#1	#1
(11) `lbl_3 :`				#1	#1	#1	#1	#1	#1	#1
(10) `mov #1,#4`	#1	#4	#1	#4	#1	#4	#1	#4	#1	#4
(9) `subi $1,#4`	#4	#4	#4	#4	#4	#4	#4	#4	#4	#4
(8) `mov #4,#1`	#4	#1	#4		#4		#4		#4	#1
(6) `jmp 16`								#1	#1	#1
(7) `lbl_2 :`				#1	#1	#1	#1	#1	#1	#1
(5) `mov #1,#3`	#1	#3	#1	#3	#1	#3	#1	#3	#1	#3
(4) `mov #3,#2`	#3	#2	#3		#3		#3		#3	#2
(3) `jmp 2`								#2	#2	#2
(2) `lbl_1 :`						#2	#2	#2	#2	#2
(22) `mov #2,%rax`	#2	*%rax*	#2		#2	*%rax*	#2	*%rax*	#2	*%rax*
(21) `frame end`				*%rax*	*%rax*	*%rax*	*%rax*	*%rax*	*%rax*	*%rax*
(1) `ret`	*%rax*		*%rax*		*%rax*		*%rax*		*%rax*	

L'implémentation en OCaml du calcul de *def and use* est un simple filtrage sur les instructions. Afin de ne pas multiplier les structures de données, le choix est fait de regrouper au sein d'un même enregistrement les calculs de *def*, *use*, *in* et *out*. Ces deux dernières seront initialisées avec l'ensemble vide lors du calcul de *def and use* et modifiées par effet de bord lors de celui de *in out*. On mémorisera les informations calculées dans `Map` indexée par les étiquettes de nœuds du graphe ERTL.

```
module DUMap = Map.Make(Graph.GLabelMod)

type def_use_t = {
  def : Regs.RegSet.t ; use : Regs.RegSet.t ;
  mutable live_in : Regs.RegSet.t ; mutable live_out : Regs.RegSet.t }

let def_use_asm = function
  | ERTL.Cqto _ →
      { def = Regs.RegSet.singleton (Regs.R_phys Regs.rdx) ;
        use = Regs.RegSet.singleton (Regs.R_phys Regs.rax) ;
        live_in = Regs.RegSet.empty ; live_out = Regs.RegSet.empty }
  | ERTL.Movi (_, reg, _) | ERTL.Lea_rip (_, reg, _) →
      { def = Regs.RegSet.singleton reg ; use = Regs.RegSet.empty ;
        live_in = Regs.RegSet.empty ; live_out = Regs.RegSet.empty }
  | ERTL.Addi (_, reg, _) | ERTL.Subi (_, reg, _) | ERTL.Muli (_, reg, _) →
      { def = Regs.RegSet.singleton reg ; use = Regs.RegSet.singleton reg ;
        live_in = Regs.RegSet.empty ; live_out = Regs.RegSet.empty }
...
  | ERTL.Div (reg, _) →
      { def = Regs.RegSet.of_list [Regs.R_phys Regs.rdx ; Regs.R_phys Regs.rax] ;
        use = Regs.RegSet.of_list [reg ; Regs.R_phys Regs.rdx ; Regs.R_phys Regs.rax] ;
```

```
        live_in = Regs.RegSet.empty ; live_out = Regs.RegSet.empty }
...
  | ERTL.Call (_, nb_args_in_regs, _, _) →
      { def = Regs.RegSet.of_list Regs.call_not_preserved ;
        use = Regs.RegSet.of_list (list_n_elems nb_args_in_regs Regs.for_params) ;
        live_in = Regs.RegSet.empty ; live_out = Regs.RegSet.empty }
...
```

Afin de conserver une relative simplicité de présentation, nous nous contentons ici d'implémenter l'algorithme naïf de calcul de *in out*. On a besoin d'une fonction auxiliaire `get_children` permettant d'obtenir la liste des étiquettes des successeurs d'un nœud. C'est un simple filtrage sur les instructions ERTL et l'on gagnera un peu de place en ne donnant pas son code.

```
let compute_in_out du_map ertl_graph =
  let changed = ref true in
  let do_one_node n =
    let n_du = DUMap.find n du_map in
    (* In (n) = use (n) U (out (n) \ def (n)). *)
    let live_in = Regs.RegSet.union n_du.use (Regs.RegSet.diff n_du.live_out n_du.def) in
    (* Out (n) = U_children in ( child ). *)
    let live_out =
      List.fold_left
        (fun accu child_n →
          Regs.RegSet.union (DUMap.find child_n du_map).live_in accu)
        Regs.RegSet.empty (get_children ertl_graph n) in
    if not ((Regs.RegSet.equal live_in n_du.live_in) && (Regs.RegSet.equal live_out n_du.live_out)) then (
      n_du.live_in ←live_in ;
      n_du.live_out ←live_out ;
      changed := true) in
  let do_graph () = Graph.G.iter (fun graph_label _ →do_one_node graph_label) ertl_graph in
  while !changed do changed := false ; do_graph () done
```

Cette fonction utilise `List.fold_left` de la bibliothèque standard d'OCaml qui permet de calculer $f(\ldots\,(f\,(f\,a\,e_1)\,e_2)\,\ldots)\,e_n$ pour une fonction f, une liste $[e_1;\ldots;e_n]$ avec un accumulateur initial a. Elle est parfaitement adaptée pour effectuer l'union des *out* de la liste des successeurs d'un nœud.

Comme pour les précédentes passes, on regroupe toutes les informations relatives à une fonction dans une seule structure après avoir d'abord calculé les *def and use* de chaque instruction du graphe, puis le point fixe des *in out*. Notons que pour une raison que nous éclaircirons plus tard dans l'allocation de registres, nous calculons et mémorisons dans une autre `Map` (`count_usages_map`) le nombre d'utilisations de chaque registre (physique ou pseudo).

```
type fun_def_t = {
  fdef_name : Ast.var_t ;
  fdef_graph : ERTL.asm_instr_t Graph.G.t ;
  fdef_entry : Graph.graph_label_t ;
  mutable fdef_frame_size : int ;
  fdef_du_map : def_use_t DUMap.t ;
  fdef_usage_map : int Regs.RegMap.t
}
```

```
let update_usages_cout du_info count_usages_map =
  let process_set set =
    Regs.RegSet.iter
      (fun reg →
        let old_count = (try Regs.RegMap.find reg !count_usages_map with Not_found →0) in
        count_usages_map := Regs.RegMap.add reg (old_count + 1) !count_usages_map)
      set in
  process_set du_info.def ;
  process_set du_info.use

let do_fundef f_def =
  let count_usages_map = ref Regs.RegMap.empty in
  let du_map =
    Graph.G.fold
      (fun ertl_label asm accu →
        let du_info = def_use_asm asm in
        update_usages_cout du_info count_usages_map ;
        DUMap.add ertl_label du_info accu)
      f_def.ERTL.fdef_graph DUMap.empty in
  compute_in_out du_map f_def.ERTL.fdef_graph ;
  { fdef_name = f_def.ERTL.fdef_name ; fdef_graph = f_def.ERTL.fdef_graph ;
    fdef_entry = f_def.ERTL.fdef_entry ; fdef_frame_size = f_def.ERTL.fdef_frame_size ;
    fdef_du_map = du_map ; fdef_usage_map = !count_usages_map }
```

12.6.2 Graphe d'interférence

Comme présenté en introduction de ce chapitre, l'information de *in out* permet de représenter pour chaque instruction quels sont les registres vivants (dont le contenu est nécessaire à la suite du calcul), donc qui ne peuvent pas être représentés ultimement par un même registre physique. Les sommets du graphe d'interférence sont les temporaires. Les arcs de ce graphe non orienté sont définis de la manière suivante.

- Si l'instruction du sommet n n'est pas un `Mov`, alors les registres de $def(n)$ interfèrent avec les registres $out(n) \setminus def(n)$. En d'autres mots, les registres écrits interfèrent avec ceux qui sont vivants en sortie de l'instruction, hormis bien sûr eux-mêmes puisqu'ils sont écrits par cette instruction.
- Si l'instruction du sommet n est un `Mov` (r_s, r_d), alors r_d interfère avec tous les registres $out(n) \setminus \{r_d; r_s\}$. En effet, à l'issue de cette instruction, les deux registres contiennent la même valeur, donc il n'interfèrent pas, bien au contraire. On aurait même envie de les matérialiser par le même registre physique.

Le cas particulier de l'instruction `Mov` est très important car il suggère que l'on devrait se souvenir que deux temporaires sont équivalents. Si l'on peut trouver un registre physique pour matérialiser l'un, autant utiliser ce registre pour matérialiser l'autre. Le gain serait double : on utilise un seul registre physique et l'on s'épargne un `Mov` inutile. Ainsi, le graphe d'interférence va comporter deux types d'arcs : les arcs d'interférence et les arcs "mov" (parfois appelés arcs de *préférence*). Dans le cas où deux arcs existent entre deux nœuds, l'arc d'interférence est prioritaire (il remplace l'arc "mov"). En effet, s'il existe une interférence entre deux registres c'est qu'ils ne peuvent pas être matérialisés par le même registre physique, quand bien même on préférerait ! Ainsi, les arcs du graphes sont tels que :

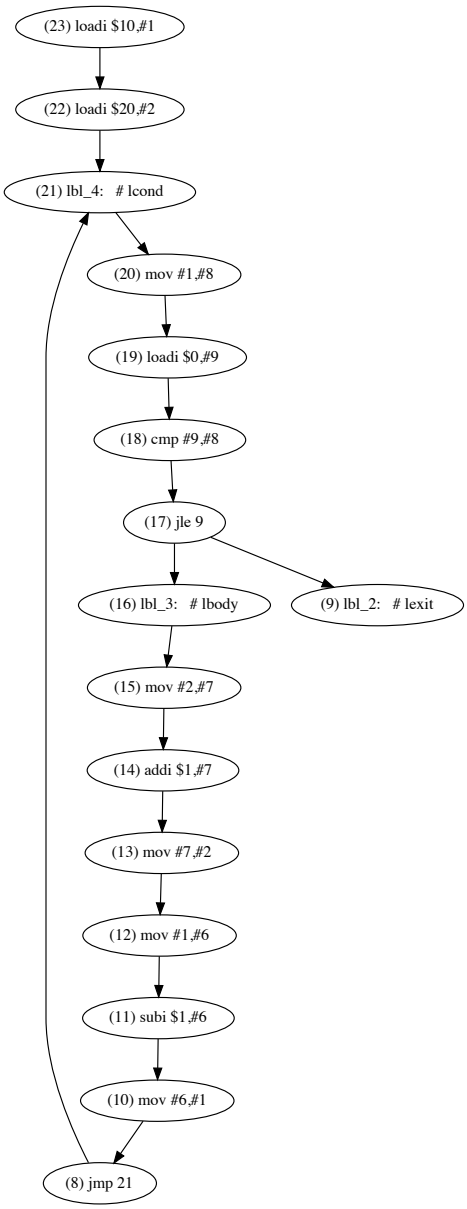
- si l'instruction du sommet n n'est pas un `Mov`, alors on ajout des arcs d'interférence entre $def(n)$ et $out(n) \setminus def(n)$,
- si l'instruction du sommet n est un `Mov` (r_s, r_d), alors on ajout un arc d'interférence entre r_d et $out(n) \setminus \{r_d; r_s\}$ et un arc "mov" entre r_s et r_d.

Pour illustrer la construction d'un tel graphe, considérons l'extrait de programme de la figure suivante dans lequel on considère que `x` et `y` sont utilisés dans la suite (omise) de cet extrait. On remarque que le compilateur a logé `x` dans le pseudo-registre `#1` et `y` dans `#2`. Les *def*, *use*, *in* et *out* ont donnés pour chaque instruction en partie gauche de la figure alors que la partie droite représente le graphe de flot de contrôle. Une fois de plus, nous n'avons pas fait figurer les registres physiques sauvegardés en prélude de fonction conformément aux conventions d'appel

de l'ABI. Ceci est une omission volontaire pour ne se concentrer que sur les pseudo-registres et obtenir un graphe d'interférence plus simple et plus lisible.

```
int x = 10 ;
int y = 20 ;
while (x > 0) do
  y = y + 1 ;
  x = x - 1 ;
done
... /* Utilise x et y. */
```

(23) `loadi $10,#1` ⇒ 22
$U = \emptyset$ $D = \{\#1\}$ $I = \emptyset$ $O = \{\#1\}$
(22) `loadi $20,#2` ⇒ 21
$U = \emptyset$ $D = \{\#2\}$ $I = \{\#1\}$ $O = \{\#1;\#2\}$
(21) `lbl_4:` ⇒ 20
$U = \emptyset$ $D = \emptyset$ $I = \#1;\#2$ $O = \{\#1\#2\}$
(20) `mov #1,#8` ⇒ 19
$U = \{\#1\}$ $D = \{\#8\}$ $I = \{\#1;\#2\}$ $O = \{\#1;\#2;\#8\}$
(19) `loadi $0,#9` ⇒ 18
$U = \emptyset$ $D = \{\#9\}$ $I = \{\#1;\#2;\#8\}$ $O = \{\#1;\#2;\#8;\#9\}$
(18) `cmp #9,#8` ⇒ 17
$U = \{\#8\#9\}$ $D = \emptyset$ $I = \{\#1;\#2;\#8;\#9\}$ $O = \{\#1;\#2\}$
(17) `jle 9` ⇒ 16
$U = \emptyset$ $D = \emptyset$ $I = \{\#1;\#2\}$ $O = \{\#1;\#2\}$
(16) `lbl_3:` ⇒ 15
$U = \emptyset$ $D = \emptyset$ $I = \{\#1;\#2\}$ $O = \{\#1;\#2\}$
(15) `mov #2,#7` ⇒ 14
$U = \{\#2\}$ $D = \{\#7\}$ $I = \{\#1;\#2\}$ $O = \{\#1;\#7\}$
(14) `addi $1,#7` ⇒ 13
$U = \{\#7\}$ $D = \{\#7\}$ $I = \{\#1;\#7\}$ $O = \{\#1;\#7\}$
(13) `mov #7,#2` ⇒ 12
$U = \{\#7\}$ $D = \{\#2\}$ $I = \{\#1;\#7\}$ $O = \{\#1;\#2\}$
(12) `mov #1,#6` ⇒ 11
$U = \{\#1\}$ $D = \{\#6\}$ $I = \{\#1;\#2\}$ $O = \{\#2;\#6\}$
(11) `subi $1,#6` ⇒ 10
$U = \{\#6\}$ $D = \{\#6\}$ $I = \{\#2;\#6\}$ $O = \{\#2;\#6\}$
(10) `mov #6,#1` ⇒ 8
$U = \{\#6\}$ $D = \{\#1\}$ $I = \{\#2;\#6\}$ $O = \{\#1;\#2\}$
(8) `jmp 21` ⇒ 21
$U = \emptyset$ $D = \emptyset$ $I = \{\#1;\#2\}$ $O = \{\#1;\#2\}$
(9) `lbl_2:` ⇒ 7
$U = \emptyset$ $D = \emptyset$ $I = \{\#1;\#2\}$ $O = \{\#1;\#2\}$
...



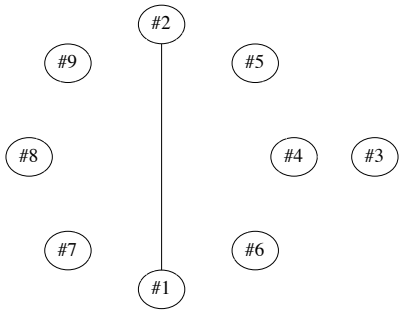
Si l'on considère l'instruction `cmp #9,#8` d'étiquette 20, on constate qu'elle lit bien `#1` et écrit bien `#8`. Les registres vivants en entrée sont $\{\#1;\#2\}$ qui sont ceux vivants en sortie de l'instruction précédente. Ceux vivants en sortie contiennent en plus `#8` qui a bien été écrit et se trouve utilisé plus tard dans l'instruction `cmp #9,#8` d'étiquette 20.

Si l'on regarde justement maintenant cette instruction `cmp #9,#8`, on se rend compte que les registres vivants en entrée sont $\{\#1;\#2;\#8;\#9\}$ alors qu'en sortie ils sont nettement moins nombreux : $\{\#1;\#2\}$. En effet, `#8` et `#9` qui étaient vivants en entrée ont été utilisés par la comparaison et ne servent plus dans la suite du programme (du moins, avant qu'ils n'aient été écrasés). Par contre, `#1` et `#2` sont bien encore utiles, par exemple dans les instructions d'étiquette 15 et 14 (ils ne sont pas écrasés entre temps).

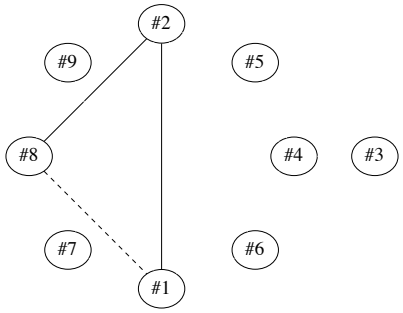
La construction du graphe d'interférence est faite en traitant chacune des instructions (de manière isolée) et en ajoutant les arcs d'interférence ou "mov" en accord avec les informations de *in out*.

Pour la première instruction, `loadi` $10, #1, *def* et *out* étant identiques, $out(n) \setminus def(n)$ est vide et aucun arc n'est ajouté.

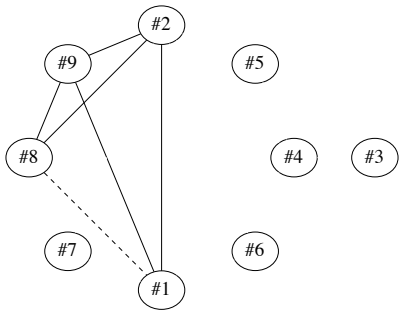
Pour la seconde, `loadi` $20, #2, on a $def = \{\#2\}$ et $out = \{\#1; \#2\}$ donc il faut rajouter un arc d'interférence entre `#2` et `#1`.



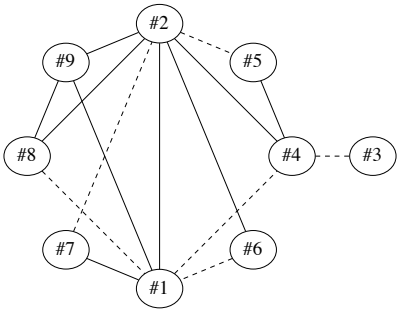
Pour la troisième, qui est un `Label`, aucun arc n'est nécessaire. La quatrième est un déplacement, `mov` #1, #8 avec $def = \{\#8\}$ et $out = \{\#1; \#2; \#8\}$. Il faut donc rajouter un arc d'interférence entre `#8` et `#2` et un arc "mov" entre `#1` et `#8`.



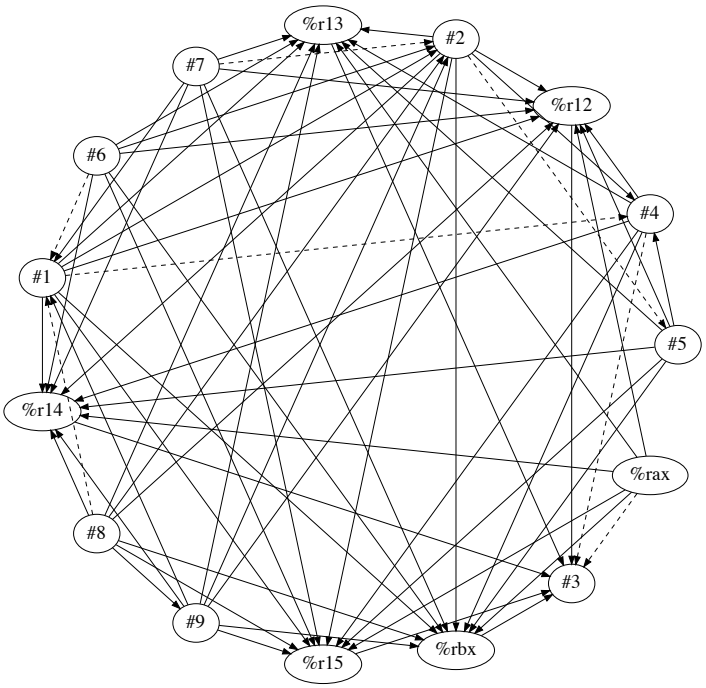
L'instruction d'après, `loadi` $0, #9 introduit des arcs d'interférence entre `#9` et $\{\#1; \#2; \#8\}$.



Le processus continue avec les autres instructions pour aboutir au graphe final.



Si l'on avait réellement inclus les registres physiques dont la sauvegarde est imposée par l'ABI et l'utilisation de `%rax` pour simplement retourner la valeur `x + y`, on aurait obtenu le graphe ci-dessous . . . et ce pour un programme tout petit !



Nous pouvons implanter la construction du graphe en OCaml. Pour cela, il nous faut une structure représentant le graphe et ses deux types d'arcs. Contrairement aux graphes RTL et ERTL (et LTL que nous verrons) plus tard, la représentation choisie n'est pas basée sur une `Map` et utilise des enregistrements avec partage physique des nœuds. Le type somme `edge_kind_t` représente les deux types d'arcs tandis que `edge_t` représente effectivement un arc entre deux nœuds avec son type. Un nœud est représenté par une structure contenant le registre qu'il représente, son degré (le nombre d'arcs arrivant sur le nœud), une « couleur » à ignorer pour le moment (qui sera mise à jour par effet de bord lors de l'allocation effective des registres physiques) et la liste des ses arcs.

```
type edge_kind_t = EK_interf | EK_mov
type edge_t = { e_child : node_t ; mutable e_kind : edge_kind_t }
type color_t = C_reg of Regs.reg_t | C_spill of int | C_none
type edge_t = { e_child : node_t ; mutable e_kind : edge_kind_t }
and node_t = {
  n_reg : Regs.temp_t ;
```

```
    mutable n_degree : int ;
    mutable n_color : color_t ;
    mutable n_children : edge_t list
  }

  let empty_graph () = { g_nodes = [ ] }
```

Dans une telle représentation, un nœud représentant un temporaire doit exister de manière unique et tout ajout d'arc « à un temporaire » doit opérer sur l'unique nœud représentant ce temporaire.

L'ajout d'un arc entre 2 nœuds est fait par deux fonctions utilitaires : `add_edge_between_nodes` qui ajoute effectivement une structure d'arc entre 2 nœuds et `add_edge` qui commence par rechercher les 2 nœuds représentant les registres reçus en paramètres, les crée s'ils n'existent pas encore, puis appelle `add_edge` pour effectivement insérer l'arc.

Le graphe d'interférence étant non orienté, l'ajout d'un arc entre r_1 et r_2 provoque l'ajout d'un arc entre r_2 et r_1. Comme expliqué précédemment, si un arc d'interférence et un arc "mov" doivent être établis entre 2 mêmes registres, alors l'arc d'interférence est prioritaire.

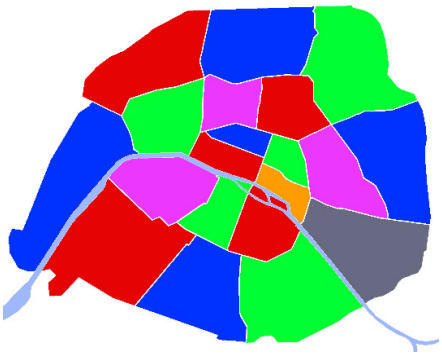
Deux dernières fonctions utilitaires mineures permettent de retrouver le nœud représentant un registre dans le graphe (`find_node`) et trouver un arc entre 2 nœuds (`find_edge_to`) s'il en existe.

```
  let rec find_edge_to dst = function
    | [ ] → raise Not_found
    | h :: q → if dst == h.e_child then h else find_edge_to dst q

  let find_node reg nodes = List.find (fun n →n.n_reg = reg) nodes

  let add_edge_between_nodes node1 node2 kind =
    try
      let edge12 = find_edge_to node2 node1.n_children in
      let edge21 = find_edge_to node1 node2.n_children in
      if kind = EK_interf then (
        edge12.e_kind ←EK_interf ;
        edge21.e_kind ←EK_interf
      )
    with Not_found →
      node1.n_children ←{ e_child = node2 ; e_kind = kind } :: node1.n_children ;
      node2.n_children ←{ e_child = node1 ; e_kind = kind } :: node2.n_children ;
      node1.n_degree ←node1.n_degree + 1 ;
      node2.n_degree ←node2.n_degree + 1

  let add_edge ~graph reg1 reg2 kind =
    let nod1 =
      (try find_node reg1 graph.g_nodes with
        Not_found →
          let n = { n_reg = reg1 ; n_children = [ ] ; n_degree = 0 ; n_color = C_none } in
          graph.g_nodes ←n :: graph.g_nodes ;
          n) in
    let nod2 =
      (try find_node reg2 graph.g_nodes with
        Not_found →
          let n = { n_reg = reg2 ; n_children = [ ] ; n_degree = 0 ; n_color = C_none } in
          graph.g_nodes ←n :: graph.g_nodes ;
          n) in
    add_edge_between_nodes nod1 nod2 kind
```

12.6.3 Coloriage de graphe

Munis du graphe d'interférence, le problème est maintenant d'assigner à chaque pseudo-registre un registre physique. Si deux pseudo-registres interfèrent (donc sont reliés par un arc d'interférence, pour le moment nous ignorons les arcs "mov") alors ils ne peuvent pas être représentés par le même registre physique. Ce problème est le même que le *coloriage de graphe*, dans lequel on a un nombre de « couleurs » (i.e. ici de registres physiques) fixé et chaque nœud doit être colorié avec une couleur différente de ses voisins.

Ce problème est connu depuis longtemps et remonte à l'époque où l'on cherchait à colorier des cartes géographiques en faisant en sorte que deux pays (ou régions) partageant une frontière soient de couleurs différentes (pour des questions de meilleure visualisation).



On appelle *k-coloriage*, pour un k fixé et un graphe de sommets S et d'arcs A, une assignation d'un entier $1 \leq c \leq k$ à chaque sommet s de S tel que pour tout s_1, s_2 de A, $c(s_1) \neq c(s_2)$. Si un tel coloriage existe, le graphe est dit *k-coloriable*.

Malheureusement pour un graphe quelconque et un nombre de couleurs $k > 2$, ce problème est NP-complet. Ainsi il faut abandonner l'idée de trouver une solution optimale dans un temps raisonnable, et l'on devra se contenter d'heuristiques. De plus, lorsqu'un graphe n'est pas coloriable (i.e. pas assez de registres physiques disponibles) il faudra trouver une solution de contournement car le compilateur ne peut décemment dire qu'il ne peut pas compiler le programme. Dans ce cas, il faudra choisir de matérialiser certains pseudo-registres par des emplacements dans la pile. On dira que l'on *spill* ces pseudo-registres.

Il est important de remarquer que nos graphes comportent des sommets déjà coloriés qui représentent les registres physiques. Nous verrons ultérieurement comment intégrer ces conditions initiales dans l'algorithme de coloriage.

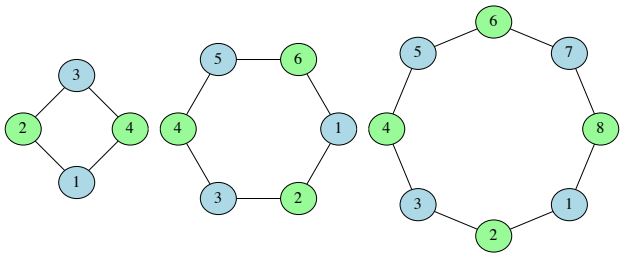
Un algorithme assez simple de coloriage a été proposé par Gregory Chaitin en 1981-1982. On appelle *trivialement coloriable* un sommet dont le degré est inférieur strictement à k (le nombre de couleurs). En effet, s'il a strictement moins de voisins que de couleurs disponibles, alors on peut assigner une couleur à chaque voisin et il en restera toujours une après pour le sommet (pour rappel, pour le moment nous ignorons les arcs "mov"). Ainsi, l'algorithme consiste à choisir un sommet trivialement coloriable s du graphe G puis à récursivement colorier $G \setminus \{s\}$. C'est ce que l'on appelle une étape de *simplification*. Au retour du coloriage de $G \setminus \{s\}$ on assigne une des couleurs disponibles à s. De cette façon, le coloriage effectif est fait à la « remontée » de la récursion. Quand on ne trouve pas de sommet trivialement coloriable, on en choisit un à *spiller*. L'algorithme en pseudo-code est donc le suivant. Lors de la recherche du nœud à supprimer, on prendra celui de degré minimal.

```
colorize (G) =
  s'il existe s de degré  < k (minimal)
    colorize (G \ {s})
    assigner à s une couleur différente de celle de ses voisins
  sinon s'il reste des sommets
    choisir s à spiller
    colorize (G \ {s})
    spiller s
  sinon fin
```

Spiller un sommet signifie matérialiser son pseudo-registre par un emplacement dans la pile. Cet emplacement se trouvera « au-dessus » (donc à des adresse plus faibles) des paramètres ayant été passés par la pile. On y accédera, comme aux paramètres, par un décalage relatif à `%rbp`. Cela implique, à l'exécution, un accès à la mémoire à chaque utilisation de la donnée contenue dans ce pseudo-registre. L'accès en mémoire étant beaucoup plus lent qu'en registre, il faut choisir les sommets à *spiller* judicieusement. En particulier il faut éviter de *spiller* les pseudo-registres souvent utilisés. Mais pour se donner plus de chances de colorier le reste du graphe, il est intéressant de *spiller* des nœuds de fort degré (ça « libèrera » plus de couleurs).

On associe donc une *fonction de coût* au *spill* de pseudo-registres et on choisit celui de coût le plus faible. Une fonction simple et intuitive est $\frac{\text{nombre d'utilisations}}{\text{degré}}$. C'est la raison pour laquelle dans l'implantation OCAML donnée en 12.6.1, nous avions mémorisé, dans les informations relatives à chaque fonction, une `Map` conservant le nombre d'utilisation de chaque pseudo-registre.

Cet algorithme est assez pessimiste : dès qu'il ne trouve pas de nœud de degré $< k$, donc qu'il n'est pas possible de *simplifier* le graphe, il *spill* un pseudo-registre. Ainsi, il échoue à colorier sans *spiller* certains graphes alors qu'ils le sont. Par exemple, c'est le cas pour $k = 2$ et n'importe quel graphe ayant un nombre de sommets pair, tous de degré 2, et formant un « cycle ».

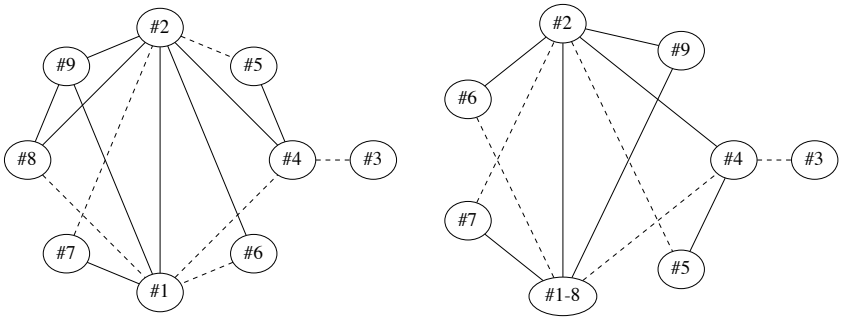


Il est possible de modifier cet algorithme légèrement pour le rendre plus optimiste. Au lieu de *spiller* aveuglement lorsqu'il n'y a pas de nœud trivialement coloriable, donc que l'on ne peut pas *simplifier* le graphe, on choisit un candidat *potentiel* au *spilling* s, on colorie $G \setminus \{s\}$ et au retour de ce coloriage, on essaye de colorier s avec une couleur possible. C'est alors seulement s'il n'y a pas de couleur disponible que l'on *spille* réellement s. En effet, il est possible que le coloriage de $G \setminus \{s\}$ ait en fait laissé des couleurs disponibles. Le pseudo-code précédent devient donc le suivant.

```
colorize (G) =
  s'il existe s de degré < k (minimal)
    colorize (G \ {s})
    assigner à s une couleur différente de celle de ses voisins
  sinon s'il reste des sommets
    choisir s candidat au spilling
    colorize (G \ {s})
    s'il reste une couleur c possible pour s
      alors attribuer c à s
    sinon spiller s
  sinon fin
```

Cette nouvelle version de l'algorithme permet maintenant de colorier les graphes cycliques donnés en exemple ci-dessus.

Jusqu'à présent seules les arêtes d'interférence ont été utilisées, les arêtes "mov" ayant été ignorées. Les arêtes "mov" indiquent qu'au contraire, on souhaiterait que deux sommets soient de la même couleur, autrement dit soient « fusionnés ». Cette technique de *coalescence* consiste à remplacer 2 sommets du graphe reliés par un arc "mov" par un nouveau sommet qui comporte tous les arcs de sommets fusionnés. En reprenant le graphe d'interférence obtenu sur l'exemple illustrant le calcul de *in out*, nous souhaiterions fusionner les nœuds `#1` et `#8` puisqu'ils sont liés par un arc "mov".



Lors d'une fusion, si un arc d'interférence et un arc "mov" apparaissent entre 2 mêmes nœuds, comme précédemment, l'arc d'interférence est privilégié et celui "mov" disparaît.

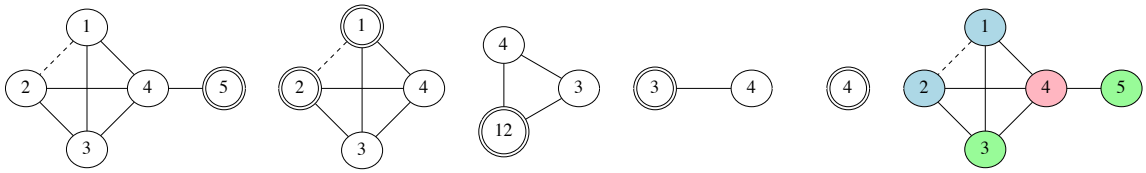
Fusionner deux sommets peut créer un sommet de très fort degré et rendre le graphe non k-coloriable alors qu'il l'était avant la fusion. Pour cette raison, il est nécessaire de déterminer un critère de « fusion raisonnable » qui permette de conserver la k-colorabilité, c'est à dire qui ne modifie pas la possibilité de k-colorier le graphe obtenu suite à la fusion (s'il 'était k-coloriable, il le reste). Deux critères différents sont communément présentés.

- Le critère de *(Preston) Briggs* : deux nœuds n_1 et n_2 peuvent être fusionnés en n_{12} si le nœud n_{12} résultat a strictement moins de k voisins non trivialement coloriables.

  Une telle fusion n'altère pas la k-colorabilité du graphe car la simplification va éliminer tous les nœuds trivialement coloriables voisins de n_{12} (comme elle l'aurait fait dans le graphe initial), ensuite pour n_{12} ne resteront que les voisins non trivialement coloriables, en nombre $< k$. Donc la simplification pourra éliminer n_{12} et il restera le problème de coloriage des voisins non trivialement coloriables, ce qui existait dans le graphe avant fusion.
- Le critère de *(Lal) George* : deux nœuds n_1 et n_2 peuvent être fusionnés si tout voisin non trivialement coloriable de n_1 est aussi voisin n_2. Dit autrement, deux nœuds n_1 et n_2 peuvent être fusionnés si pour tout voisin n' de n_1, n' interfère déjà avec n_2 ou n' est trivialement coloriable.

  Une telle fusion n'augmente en fait le degré d'aucun voisin non trivialement coloriable puisqu'il interférait déjà avec les deux nœuds. Et pour les voisins trivialement coloriables, ils perdent l'un des nœuds fusionnés comme voisin et gagnent le nouveau nœud, donc pas d'augmentation de degré non plus.

Comme le montre le schéma ci-dessous (pour $k = 3$), quand on supprime un nœud trivialement coloriable (5), le graphe se trouve simplifié et deux nœuds peuvent devenir fusionnables. De même, quand on fusionne deux nœuds (1 et 2), le graphe peut devenir k-coloriable alors qu'il ne l'était pas avant. Il faut donc que ces deux étapes soient appliquées alternativement.



Toutefois la simplification (suppression d'un nœud trivialement coloriable) doit éviter de supprimer un nœud qui pourrait être fusionné avec un autre : cela ferait perdre une occasion d'utiliser le même registre physique pour plusieurs pseudo-registres. Ainsi, la recherche d'un nœud à simplifier doit satisfaire 2 critères :

- le nœud doit être trivialement coloriable (donc de degré $< k$)
- et il ne doit pas comporter d'arc "mov".

Remarquons que la combinaison ces deux critères fait que l'on n'a pas besoin de se demander si les arcs "mov" comptent dans le degré d'un nœud. En effet, si un nœud possède un arc "mov", alors il est exclu des candidats à simplification, donc son degré ne nous importe pas.

Lorsque les alternances de simplifications / fusions aboutissent à un graphe sans sommet trivialement coloriable et sans sommet fusionnable, on pourrait envisager de *spiller* un pseudo-registre. Néanmoins, avant de se replier sur cette solution par dépit, on va se donner une dernière chance en supprimant les arcs "mov" d'un nœud (on dit que l'on *gèle* l'arc) puis re-tenter de colorier le graphe. On va donc rechercher un nœud trivialement coloriable (si l'on

en trouve un alors il a forcément des arcs "mov" puisqu'on l'a ignoré lors de la recherche de candidat à la fusion juste au-dessus). Si l'on trouve un tel nœud on supprime ses arcs "mov" et on recommence le processus global de coloriage, sinon il ne reste plus qu'à *spiller* un pseudo-registre en accord avec la fonction de coût.

Il en découle un algorithme tel que proposé Lal George et Andrew W. Appel, découpé en 5 fonctions mutuellement récursives :

- `colorize` : point d'entrée du coloriage qui ne fait qu'appeler `simplify` (on pourrait s'en passer en utilisant `simplify` à la place mais l'algorithme est plus lisible et plus élégant quand on identifie clairement le processus récursif global de « coloriage » en le différenciant des différentes étapes qui le composent).
- `simplify` : recherche un nœud trivialement coloriable, sans arcs "mov", de degré minimal, pour le supprimer du graphe et colorier le graphe restant. Si aucun nœud n'est disponible, la fusion va être lancée.
- `merge` : recherche deux nœuds à fusionner, s'il en existe, les fusionne et colorie le graphe obtenu. Si aucun nœud ne peut être fusionné, le gel va être considéré.
- `freeze` : recherche un candidat au gel, s'il en existe un, supprime ses arcs "mov" et colorie le graphe obtenu. Si aucun candidat n'est disponible, le *spilling* reste la dernière solution.
- `spill` : s'il ne reste plus de nœud (ou que des nœuds représentant des registres physiques), le coloriage est en fait terminé. Sinon, recherche le meilleur candidat au *spilling*, le supprime du graphe et colorie le graphe restant. À l'issue du coloriage, si (par bonheur) une couleur reste disponible pour le candidat, on lui attribue cette couleur, sinon on lui attribue une couleur « en pile ».

colorize (G) = *simplify* (G)

simplify (G) =
  s'il existe n de degré $< k$ (minimal) sans arc "mov"
    colorize $(G \setminus \{n\})$
    assigner à n une couleur différente de celle de ses voisins
  sinon *merge* (G)

merge (G) =
  s'il existe n_1, n_2 avec arc "mov" et satisfaisant le critère de George
    fusionner n_1 et n_2 en n_{12}
    simplify $(G \setminus \{n_1; n_2\} \cup \{n_{12}\})$
    assigner à n_1 et n_2 la couleur assignée à n_{12}
  sinon *freeze*(G)

freeze (G) =
  s'il existe n de degré $< k$ (minimal)
    supprimer dans G les arcs "mov" de n
    simplify (G)
  sinon *spill* (G)

spill (G) =
  s'il ne reste plus de nœud dans G ou que des nœuds de registres physiques alors fin
  sinon
    trouver n de coût minimal
    colorize $(G - \{n\})$
    s'il reste une couleur c possible pour n
      alors attribuer c à n
    sinon attribuer à n un emplacement dans la pile

Lorsque l'on supprime un nœud du graphe, on supprime également les arcs dont il dispose. Le graphe étant non orienté, un arc entre n_1 et n_2 étant, dans l'implantation donnée, représenté par en fait 2 arcs (un dans la structure de n_1 avec pour destination n_2 et un dans la structure de n_2 avec pour destination n_1), il faudra penser à supprimer les deux arcs à chaque fois.

Lorsque l'on fusionne deux nœuds, un nouveau nœud apparaît à la place : il faudra donc penser à mettre à jour la `Map` mémorisant le nombre d'utilisations des registres en y rajoutant une entrée pour ce nouveau nœud (son

nombre d'utilisations sera la somme des utilisations des deux nœuds).

Lorsque l'on *spill* un pseudo-registre, on lui assigne un emplacement en mémoire, et plus particulièrement dans la pile. Pourtant, plusieurs pseudo-registres *spillés* peuvent ne pas interférer et l'on souhaiterait les matérialiser par le même emplacement de pile pour réduire la mémoire utilisée.

Au lieu de directement assigner un emplacement en pile, on peut noter quels pseudo-registres doivent être *spillés* et on se retrouve à nouveau avec un problème de coloriage de graphe pour l'allocation « de spill ». Cette fois-ci le nombre de couleurs est illimité (représentant les futurs réels emplacements en pile). Bien entendu, dans ce nouveau coloriage, il n'est pas question de *spiller* les *spill*, ce qui n'aurait aucun sens car reviendrait à sauver sur la pile quelque chose qui a été sauvé sur la pile. L'algorithme de coloriage se simplifie donc en :

1. Fusionner tous les nœuds ayant un arc "mov" entre eux. En effet, on n'a plus besoin de se soucier d'une fusion qui pourrait rendre le graphe non k-coloriable : k est infini maintenant. D'autre part, on maximise ainsi le partage d'un même futur emplacement de pile entre les pseudo-registres.
2. Colorier le graphe en utilisant `simplify` qui choisira (par heuristique) le nœud de degré le plus faible. Lorsque l'on devra attribuer une couleur, on prendra la première libre dans la pile (dans les faits, il suffira d'avoir un compteur incrémenté à chaque assignation de couleur).

Dans l'algorithme de coloriage présenté ci-dessus, il n'a quasiment jamais été fait mention des registres physiques présents dans le graphe d'interférence. En effet, lors du passage en ERTL ces derniers ont commencé à faire leur apparition au niveau de certaines instructions (`Div` par exemple), dans le prélude et le postlude des fonctions, ainsi qu'aux sites d'appel. Ces registres sont en fait pré-coloriés et ne doivent pas être re-coloriés. Afin de ne pas risquer de les re-colorier, on pourrait être tenté de les supprimer des nœuds du graphe, tout en laissant les arcs dont ils sont sources ou destination. Ceci ne fonctionne pas à cause de la fusion qui peut les faire réapparaître. La solution est d'exclure ces registres des recherches de candidats à la simplification, au gel et au *spill*. Pour ce qui est de la fusion, on peut fusionner un registre physique et un pseudo-registre, mais certainement pas deux registres physiques. Lorsque l'on a trouvé une paire de registres à fusionner, si l'un est un registre physique (donc déjà colorié), il donne directement sa couleur au nœud résultant de la fusion, donc avant même de tenter de colorier le graphe résultant. Ce nœud issu de la fusion devient alors directement coloré et sera ignoré des recherches précitées.

Nous allons illustrer l'algorithme de coloriage du graphe d'interférence en Ocaml, en faisant le choix simplificateur de ne pas chercher à minimiser les *spills* par un second coloriage. Ainsi, nous implantons l'algorithme tel qu'il a été décrit avec ses 5 fonctions et lorsque nous devons *spiller* un pseudo-registre, nous lui attribuons directement une nouvelle location dans la pile.

Le choix est volontairement fait de commencer par présenter le cœur du coloriage (les 5 fonctions), puis les fonctions auxiliaires servant à rechercher les nœuds candidats aux différentes opérations (simplification, fusion, gel et *spill*). Cela permet d'appréhender la structure générale de l'algorithme.

Le nombre de couleurs est le nombre de registres physiques du processeur que l'on souhaite pouvoir allouer. Ces registres sont définis dans le module `Regs` par la liste `allocatable_phys`. Seront possibles d'être alloués les registres `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13` et `%r14`. Les autres registres sont exclus pour plusieurs raisons. Nous avons vu que `%rbp` était utilisé dans le prélude de la fonction pour servir de pointeur de *stack frame* : il doit donc ne pas être modifié durant l'exécution de la fonction sinon il ne serait plus possible d'accéder aux valeurs *spillées* et aux paramètres passés sur la pile. Le registre `%rsp` est le pointeur de pile : il ne peut clairement pas servir à autre chose. Quant à `%r15`, nous faisons le choix de le garder de côté pour des usages occasionnels qui seront détaillés lors du passage en LTL (cf. section 12.7.2).

```
let allocatable_phys = [ rax ; rbx ; rcx ; rdx ; rsi ; rdi ; r8 ; r9 ; r10 ; r11 ; r12 ;  r13 ; r14 ]
let nb_allocatable_phys = List.length allocatable_phys
```

Souvenons-nous que lorsque nous avions implanté la structure du graphe d'interférence, nous avions introduit le type des « couleurs » (**type** `color_t = C_reg` **of** `Regs.reg_t | C_spill` **of** `int | C_none`). Un nœud peut ainsi colorié par un registre physique, par une location de *spill* ou bien ne pas avoir encore de couleur.

L'implémentation du coloriage qui suit n'est pas des plus optimisée. En particulier, elle n'utilise pas de structures de données malignes pour accélérer les recherches de nœuds. C'est un choix volontaire pour simplifier la présentation et coller à l'algorithme tel que décrit ci-dessus. Les fonctions prennent en arguments le graphe d'interférence ainsi que la `Map` donnant pour chaque registre (physique ou pseudo) le nombre d'utilisations qu'il en est fait.

```
let rec colorize igraph usage_map = simplify igraph usage_map

and simplify igraph usage_map =
  (* Trouver un noeud n de degré minimal < nb_colors, non colorié (pas un registre physique déjà précolorié)
     donc trivialement coloriable sans arc Mov. *)
  match find_node_min_deg_lt_nb_colors_no_mov_edge igraph with
  | Some found_node →
      let igraph' = remove_node_from_graph found_node igraph in
      colorize igraph' usage_map ;
      let color = (match find_unused_color found_node with None →assert false | Some c →c) in
      found_node.Interfer.n_color ←Interfer.C_reg color
  | None → merge igraph usage_map

and merge igraph usage_map =
  (* Rechercher n1 et n2 reliés par un arc Mov et satisfaisant le critère de fusion conservative (George). *)
  match find_nodes_for_merge igraph with
  | Some (found_n1, found_n2) →
      (* Créer un noeud n1n2 en privilégiant les arc normaux par rapport aux arcs Mov. *)
      let (igraph', n1n2_node) = merge_two_nodes igraph usage_map found_n1 found_n2 in
      (* Retirer n1 et n2 du graphe (après avoir fusionné les noeuds sinon ils disparaissent ainsi que
         leurs arcs). *)
      let igraph' = remove_node_from_graph found_n1 igraph' in
      let igraph' = remove_node_from_graph found_n2 igraph' in
      simplify igraph' usage_map ;
      found_n1.Interfer.n_color ←Interfer.C_reg color ;
      found_n2.Interfer.n_color ←Interfer.C_reg color
  | None → freeze igraph usage_map

and freeze igraph usage_map =
  (* Trouver un noeud n de degré minimal < nb_colors, donc trivialement coloriable qui n'est pas un registre
     physique. Comme on sait que l'on n'en a pas trouvé sans arc Mov avant, on relâche ce critère et on accepte
     un noeud avec des arcs Mov. *)
  match find_node_min_deg_lt_nb_colors igraph with
  | Some found_node →
      remove_mov_edges found_node ;
      simplify igraph usage_map
  | None → spill igraph usage_map

and spill igraph usage_map =
  (* On prend un noeud n par défaut, pas un registre physique, et de coût minimal. *)
  match find_candidate_for_spilling igraph usage_map with
  | None → () (* Terminé. *)
  | Some found_node →
      let igraph' = remove_node_from_graph found_node igraph in
      colorize igraph' usage_map ;
      (* On regarde si, bien que n n'était pas trivialement coloriable, on peut quand même le colorier. *)
      let color =
        (match find_unused_color found_node with
        | None → Interfer.C_spill (new_spill ())
        | Some c → Interfer.C_reg c) in
      found_node.Interfer.n_color ←color
```

On remarque que l'on colorie les nœuds en modifiant directement en place leur champ de couleur. On peut légitimement se demander si l'on ne serait pas en train de réécrire une couleur déjà en place. De par sa structure, l'algorithme garantit que si l'on trouve un nœud trivialement coloriable alors il n'est pas déjà colorié et il sera

forcément coloriable (même dans l'algorithme simple de Chaitin, donné initialement). Et si l'on n'en trouve pas, alors au pire on choisira ultimement un nœud à *spiller*, qui n'était pas déjà colorié et qui sera donc forcément colorié à l'issue de sa sélection. Comme dans tous les cas on supprime du graphe le nœud sélectionné, il ne pourra être colorié par une itération suivante (quand on colorie un nœud, c'est toujours juste après avoir colorié récursivement le graphe privé de ce nœud).

Bien entendu, lorsque l'on implémente un tel algorithme, surtout en phase de mise au point, il est plus sage d'effectuer l'affectation de couleur au moyen d'une fonction auxiliaire qui commence par s'assurer que le nœud n'a pas déjà une couleur et échoue en émettant un message d'erreur dans le cas contraire.

La structure générale étant donnée, nous pouvons donner l'implantation des différentes fonctions auxiliaires, de recherche de candidats ou servant à écrire ces dernières.

La fonction `find_node_min_deg_lt_nb_col` permet de rechercher un nœud de degré minimal < *nb_colors*. Ce sera donc un nœud trivialement coloriable. Elle accepte de retourner un noeud ayant un arc "mov". Elle s'appuie sur le fait que nous aurons au préalable trié la liste des nœuds du graphe par degré croissant (ce qu'il faudra faire initialement et lorsque l'on rajoutera un nœud issu d'une fusion).

```
let find_node_min_deg_lt_nb_colors graph =
  let rec rec_search = function
    | [ ] → None
    | node :: rem →
        if node.Interfer.n_degree ≥nb_colors then None
        else (
          if node.Interfer.n_color ≠Interfer.C_none then rec_search rem
          else Some node) in
  rec_search graph.Interfer.g_nodes
```

La fonction `find_node_min_deg_lt_nb_colors_no_mov_edge` recherche un noeud de degré minimal < *nb_colors* et sans arc "mov". Ce sera donc un nœud trivialement coloriable. Elle s'appuie aussi sur le tri en ordre croissant de degré de la liste des nœuds du graphe.

```
let find_node_min_deg_lt_nb_colors_no_mov_edge graph =
  let rec rec_search = function
    | [ ] → None
    | node :: rem →
        if node.Interfer.n_degree ≥nb_colors then None
        else (
          if node.Interfer.n_color ≠Interfer.C_none
             ||
             List.exists
                 (fun edge → edge.Interfer.e_kind = Interfer.EK_mov)
                 node.Interfer.n_children then
            rec_search rem
          else Some node
         ) in
  rec_search graph.Interfer.g_nodes
```

Nous allons maintenant donne l'implantation des fonctions permettant de trouver deux nœuds satisfaisant le critère de George, recherche préalable à la fusion. Pour rappel, deux candidats n_1 et n_2 satisfont ce critère si tous les voisins non trivialement coloriables de n_1 sont des voisins n_2. Bien entendu, les deux nœuds doivent être différents. De plus, les deux ne peuvent pas représenter des registres physiques en même temps. Cette fonction prend donc en arguments un nœud n_1 et la liste $emov_1$ de ses arcs de type "mov". Elle devra rechercher un éventuel nœud n_2. Elle récupère tous les voisins non trivialement coloriables de n_1. Ensuite elle parcoure chaque nœud n_2 destination des

arcs de $emov_1$ et vérifie s'il est différent de n_1 et si la liste des voisins non trivialement coloriables de n_1 est incluse dans les voisins de n_2. Tant qu'elle n'a pas trouvé un nœud n_2 satisfaisant ces critères, elle continue à parcourir la liste $emov_1$.

Ainsi, nous avons besoin de 2 fonctions auxiliaires : `get_non_triv_colorable_neighbs` qui retourne la liste des voisins non trivialement coloriables d'un nœud et `list_is_included_in` qui vérifie que les éléments de sa première liste sont tous présents dans la seconde liste (de nœuds). Comme les nœuds sont des enregistrements et n'existent qu'en une seule occurrence, nous pouvons utiliser un test d'égalité physique (`List.memq` et non `List.mem`) pour tirer parti du partage des nœuds entre les arcs.

```
let list_is_included_in l incl_in = List.for_all (fun n →List.memq n incl_in) l

let get_non_triv_colorable_neighbs node =
  let rec rec_filter = function
    | [ ] → [ ]
    | edge :: rem →
        if edge.Interfer.e_child.Interfer.n_degree > nb_colors then
          edge.Interfer.e_child :: (rec_filter rem)
        else rec_filter rem in
  rec_filter node.Interfer.n_children
```

Nous pouvons donner désormais l'implantation de la fonction recherchant, pour un nœud, un second nœud tel que la paire satisfasse le critère de George. Comme indiqué ci-dessus, cette fonction prend en argument un nœud et la liste de ses arcs de type "mov".

```
let find_nodes_george_criterion node edges =
  let neighbs_node = get_non_triv_colorable_neighbs node in
  let rec rec_find = function
    | [ ] → None
    | edge :: rem →
        let node' = edge.Interfer.e_child in
        if node != node' then (
          if node.Interfer.n_color ≠Interfer.C_none &&
            node'.Interfer.n_color ≠Interfer.C_none then
            rec_find rem
          else (
            let neighbs_node' = List.map (fun edge →edge.Interfer.e_child) node'.Interfer.n_children in
            if list_is_included_in ntriv_neighbs_node neighbs_node' then Some node'
            else rec_find rem
          )
        )
        else rec_find rem in
  rec_find edges
```

La fonction `find_nodes_for_merge` est en charge d'utiliser `find_nodes_george_criterion` pour trouver 2 nœuds candidats à la fusion.

```
let find_nodes_for_merge igraph =
  let rec rec_find = function
    | [ ] → None
    | node :: rem →
```

```
      let edges_with_mov =
        List.filter
          (fun e → e.Interfer.e_kind = Interfer.EK_mov)
          node.Interfer.n_children in
      match find_nodes_george_criterion node edges_with_mov with
      | Some node' → Some (node, node')
      | None → rec_find rem in
  rec_find igraph.Interfer.g_nodes
```

Pour terminer le code concernant la fusion, il ne reste qu'à donner le code des deux fonctions `merge_two_nodes` et `remove_node_from_graph`. Une fonction auxiliaire `insert_in_sorted` est nécessaire pour insérer un nœud issu d'une fusion au bon endroit dans la liste des nœuds du graphe, que nous maintenons triée par degrés croissants.

```
let remove_node_from_graph rem_node igraph =
  let nodes' =
    List.filter (fun node →node != rem_node) igraph.Interfer.g_nodes in
  List.iter
    (fun child_edge →
      let child_node = child_edge.Interfer.e_child in
      child_node.Interfer.n_children ←
        List.filter
          (fun back_edge →
            if back_edge.Interfer.e_child == rem_node then (
              child_node.Interfer.n_degree ←child_node.Interfer.n_degree − 1 ;
              false
             )
            else true)
          child_node.Interfer.n_children)
    rem_node.Interfer.n_children ;
  { Interfer.g_nodes = nodes' }

let rec insert_in_sorted e = function
  | [ ] → [e]
  | h :: q →
      if h.Interfer.n_degree > e.Interfer.n_degree then e :: h :: q else h :: (insert_in_sorted e q)

let merge_two_nodes igraph usage_map n1 n2 =
  let new_color =
    (match n1.Interfer.n_color, n2.Interfer.n_color with
    | (c, Interfer.C_none) | (Interfer.C_none, c) → c
    | (_, _) → assert false) in
  let reg1reg2 = Regs.R_pseudo (RTL.new_reg ()) in
  let n1n2 = {
    Interfer.n_reg = reg1reg2 ; Interfer.n_children = [ ];
    Interfer.n_degree = 0 ; Interfer.n_color = new_color } in
  let usages =
    (Regs.RegMap.find n1.Interfer.n_reg !usage_map) + (Regs.RegMap.find n2.Interfer.n_reg !usage_map) in
  usage_map := Regs.RegMap.add reg1reg2 usages ! usage_map ;
  let add_edges from_node =
    List.iter
      (fun edge → Interfer.add_edge_between_nodes n1n2 edge.Interfer.e_child edge.Interfer.e_kind)
      from_node.Interfer.n_children in
  add_edges n1 ;
```

```
    add_edges n2 ;
    let igraph' = { Interfer.g_nodes = insert_in_sorted n1n2 igraph.Interfer.g_nodes } in
    (igraph', n1n2)
```

Il est important de noter que lors de la fusion, le nouveau nœud `n1n2` est ajouté dans la liste des nœuds du graphe après que ses arcs lui aient été rajoutés. Cela est nécessaire pour que son degré soit mis à jour et l'insertion dans la liste triée par degrés croissants.

D'autre part, la fonction locale `add_edges` n'utilise pas `Interfer.add_edge` car nul besoin de créer les nœuds correspondants aux registres à fusionner : on sait qu'ils existent déjà (on les a sous la main).

Il nous reste quelques fonctions à donner, pour trouver un candidat au *spilling*, retirer les arcs "mov" d'un nœud, trouver une couleur non utilisée par les voisins d'un nœud et fabriquer un nouvel emplacement de *spill* sur la pile.

```
let find_candidate_for_spilling igraph usage_map =
  let rec rec_find best_accu = function
    | [ ] → best_accu
    | h :: q →
        if h.Interfer.n_color ≠Interfer.C_none then rec_find best_accu q
        else (
          let cost =
            (float_of_int (Regs.RegMap.find h.Interfer.n_reg !usage_map))
              /.
            (float_of_int h.Interfer.n_degree) in
          let new_best_accu =
            (match best_accu with
            | None → Some (cost, h)
            | Some (old_best_cost, _) → if cost < old_best_cost then Some (cost, h) else best_accu) in
          rec_find new_best_accu q
         ) in
  match rec_find None igraph.Interfer.g_nodes with
  | None → None
  | Some (_, best_reg) →Some best_reg

let remove_mov_edges node =
  let edges' =
    List.filter
      (fun e →
        match e.Interfer.e_kind with
        | Interfer.EK_interf →true
        | Interfer.EK_mov →
            let dest_node = e.Interfer.e_child in
            dest_node.Interfer.n_children ←
              List.filter (fun e → e.Interfer.e_child != node) dest_node.Interfer.n_children ;
            false)
      node.Interfer.n_children in
  node.Interfer.n_children ←edges'

let find_unused_color node =
  let all_children_colors =
    List.fold_left
      (fun accu edge →
        let child_color = edge.Interfer.e_child.Interfer.n_color in
        match child_color with
```

```
        | Interfer.C_reg color_reg →if not (List.mem color_reg accu) then color_reg :: accu else accu
        | Interfer.C_none | Interfer.C_spill _ →accu)
      [ ] node.Interfer.n_children in
  List.find_opt (fun col →not (List.mem col all_children_colors)) Regs.allocatable_phys

let (new_spill, frame_size, clear_spills) =
  let fr_size = ref 0 in
  (fun () → fr_size := !fr_size − RTL.word_size ; !fr_size),
  (fun () → − (! fr_size)),
  (fun () → fr_size := 0)
```

La fonction `find_candidate_for_spilling` de recherche d'un candidat au *spilling* se base sur la fonction de coût introduite précédemment. La fonction `remove_mov_edges` filtre les arcs en ne conservant que ceux d'interférence. Lorsqu'elle rencontre un arc "mov", elle l'oublie et retire également l'arc « miroir » qui se trouve dans le nœud voisin. La gestion des emplacements de *spill* dans la pile est faite au moyen d'un compteur qui représente le déplacement (négatif) par rapport à `%rbp` (compteur qui sera « augmenté » à chaque allocation d'un nouvel emplacement). La taille de la *stack frame* finale sera donc l'opposé du déplacement max qui aura été alloué pour cette fonction. Le compteur étant rémanent, il conviendra d'appeler `clear_spills` à chaque début de traitement de fonction afin de le réinitialiser à 0 (c'est un peu du bricolage, mais bon ☺).

Les fonctions de l'allocation de registres par coloriage du graphe étant données, il ne reste qu'à appeler cet algorithme de coloriage pour chaque définition de fonction. Le coloriage va colorier le graphe d'interférence par effet de bord, et mettre à jour la taille de la *stack frame* aussi par référence dans la structure représentant la fonction (et obtenue lors du passage en ERTL).

```
let do_fundef f_def =
  List.iter
    (fun node →
      match node.Interfer.n_reg with
      | Regs.R_phys rname →node.Interfer.n_color ←Interfer.C_reg rname
      | Regs.R_pseudo _ → ())
   f_def.Interfer.fdef_igraph.Interfer.g_nodes ;
  clear_spills () ;
  colorize f_def.Interfer.fdef_igraph (ref f_def.Interfer.fdef_usage_map) ;
  f_def.Interfer.fdef_frame_size ←frame_size ()
```

Cette fonction commence par pré-colorier les registres physiques, puis elle remet à zéro le décalage des *spills*, effectue le coloriage du graphe et termine par la mise à jour de la taille de la *stack frame*. Notons qu'elle crée une référence sur la `Map` de compte d'utilisations des registres puisque cette dernière doit pouvoir être modifiée lorsque l'on crée un nouveau pseudo-registre suite à une fusion.

12.6.4 Compléments

Notons que d'autres techniques d'allocation existent également (par exemple l'allocation par parcours linéaire d'intervales de vie).

L'article original de Briggs, Cooper et Torczon est *"Improvements to graph coloring register allocation"* dans ACM Transactions on Programming Languages and Systems de mai 1994 (DOI 10.1145/177492.177575).

Celui de George et Appel est *"Iterated register coalescing"* dans ACM Transactions on Programming Languages and Systems de mai 1996 (DOI 10.1145/229542.229546).

Pour l'allocation par parcours linéaire, l'article de Poletto et Sarkar est : *"Linear scan register allocation"* dans ACM Transactions on Programming Languages and Systems Septembre de 1999 (DOI 10.1145/330249.330250).

12.7 Vers LTL

Le programme compilé doit subir une dernière transformation avant la production du code assembleur final. Cette dernière représentation intermédiaire, toujours sous forme d'un graphe, est le LTL (pour *Location Transfer Language*). Dans cette représentation les instructions réelles assembleur sont utilisées. La correction de la pile aux sites d'appel pour garantir l'alignement sur 16 octets est insérée. Tous les pseudo-registres disparaissent au profit des registres physiques et des emplacements de *spill* dans la pile. La *stack frame* est réellement allouée (et libérée). Les arguments passés par la pile sont finalement transférés dans des registres (voire dans un emplacement de *spill* même si ce n'est pas efficace). Pour terminer, quelques instructions assembleur peuvent se rajouter pour gérer des cas particuliers.

12.7.1 Structure du LTL

L'activité principale de cette transformation va donc être d'itérer sur les instructions du graphe ERTL, tout comme nous l'avons fait sur celles du RTL pour obtenir le ERTL. Dans la majorité des cas, nous allons nous contenter de remplacer les pseudo-registres de l'instuction ERTL par les registres physiques ou les *spill* assignés par le coloriage. La structure du RTL est donc très proche du ERTL. Les pseudo-instructions `Frame_start`, `Frame_end` et `Stack_prm_read` disparaissent au profit de manipulations explicites de la pile par rapport à `%rbp`. Les opérandes de ces instructions sont désormais le résultat du coloriage, donc des `Interfer.color_t`.

```
type asm_instr_t =
  | Cqto of Graph.graph_label_t
  | Mov of (Interfer.color_t * Interfer.color_t * Graph.graph_label_t)
  | Movi of (Int64.t * Interfer.color_t * Graph.graph_label_t)
  | Add of (Interfer.color_t * Interfer.color_t * Graph.graph_label_t)
  | Addi of (Int64.t * Interfer.color_t * Graph.graph_label_t)
  | Subi of (Int64.t * Interfer.color_t * Graph.graph_label_t)
  | Sub of (Interfer.color_t * Interfer.color_t * Graph.graph_label_t)
  | Muli of (Int64.t * Interfer.color_t * Interfer.color_t * Graph.graph_label_t)
  | Mul of (Interfer.color_t * Interfer.color_t * Graph.graph_label_t)
  | Div of (Interfer.color_t * Graph.graph_label_t)
  | Cmp of (Interfer.color_t * Interfer.color_t * Graph.graph_label_t)
  | Cmpi of (Int64.t * Interfer.color_t * Graph.graph_label_t)
  | Neg of (Interfer.color_t * Graph.graph_label_t)
  | Call of (Ast.var_t * Graph.graph_label_t)
  | Jmp of Graph.graph_label_t
  | Jcc of (RTL.cc_t * Graph.graph_label_t * Graph.graph_label_t)
  | Label of (string * string * Graph.graph_label_t)
  | Ret
  | Set of (RTL.cc_t * Interfer.color_t * Graph.graph_label_t)
  | Push of (Interfer.color_t * Graph.graph_label_t)
  | Pop of (Interfer.color_t * Graph.graph_label_t)
  | Nop of Graph.graph_label_t
  | Mov_rel_rbp of (int * Interfer.color_t * Graph.graph_label_t)
  | Lea_rip of (string * Interfer.color_t * Graph.graph_label_t)
  | Andi of (Int64.t * Interfer.color_t * Graph.graph_label_t)

type fun_def_t = {
  fdef_name : Ast.var_t ;
  fdef_graph : asm_instr_t Graph.G.t ;
  fdef_entry : Graph.graph_label_t
}
```

Deux instructions font leur apparition et une diffère légèrement de sa version ERTL (`Muli`) :

- `Mov_rel_rbp` : Cette instruction permet de charger un registre avec le contenu d'une zone mémoire dont l'adresse est le contenu de `%rbp` plus un déplacement (en octets). Ainsi, `Mov_rel_rbp` (o, r) a pour effet $r \leftarrow [\%\texttt{rbp} + o]$.
- `Andi` : Cette instruction permet de faire un « et bit-à-bit » entre une valeur immédiate et un registre. Elle nous servira pour gérer l'instruction `Set` utilisée pour compiler les expressions relationnelles utilisées en dehors des conditions. Ainsi, `Andi` (i, r) a pour effet $r \leftarrow i \,\&\, r$.
- `Muli` : L'instruction de multiplication par une constante du X86-64, ne peut pas prendre seulement la constante et un registre. Il lui faut forcément 3 opérandes : la constante, le registre source et le registre destination. Ainsi, `Muli` (i, r_s, r_d) a pour effet $r_d \leftarrow i \times r_s$.

Comme pour les représentations intermédiaires précédentes, toutes les informations relatives à une fonction sont regroupées dans un enregistrement. Pour cette dernière représentation, seuls sont nécessaires le nom de la fonction, son graphe LTL et l'étiquette de graphe de son point d'entrée.

12.7.2 Construction du graphe ERTL

Dans beaucoup de cas aucun changement profond n'est opéré (par exemple pour les sauts, la majorité des opérations avec une constante, etc.). D'autres instructions nécessitent des traitements particuliers. Tout comme ce fut le cas pour la construction du graphe ERTL, nous allons greffer des instructions pour obtenir le LTL. Il n'est pas non plus nécessaire d'effectuer un parcours récursif du graphe, seule une itération sur la `Map` représentant le graphe suffit. Le graphe LTL partage de nouveau ses étiquettes avec le ERTL.

Le traitement du `Mov` permet d'éliminer les déplacements inutiles entre un même registre. En effet, lors des passes précédentes, nous avons généré sans retenue des pseudo-registres. Grâce aux arcs "mov" du graphe d'interférence, nous avons réussi à matérialiser plusieurs pseudo-registres par un même registre physique. De ce fait, il va apparaître des instructions de la forme `Mov` (r, r) que l'on va remplacer par un `Nop` (qui sera éliminé lors de la production du texte assembleur). Le traitement de cette instruction fait apparaître un problème général : la majorité des instructions du x86-64 n'acceptent au plus qu'une seule opérande mémoire. Si l'on doit traduire un `Mov` entre deux emplacements *spillés*, il faut un registre intermédiaire. C'est la raison pour laquelle, nous avons gardé de coté (rendu non allouable) le registre `%r15` lors de l'allocation de registres : il nous servira à cet effet. Ainsi, un `Mov` entre deux registres physiques ou bien un registre physique et un emplacement en mémoire seront traduits tels quels. Un `Mov` (sp_s, sp_d) entre deux zones *spillées* sera traduit par :

`mov` sp_s, `%r15`
`mov` `%r15`, sp_d

Pour le `Muli`, lorsque la seconde opérande est un registre physique, on l'utilise à la fois comme source et destination du calcul. Si c'est une zone de *spill*, on utilise le registre temporaire.

L'instruction `Mul` nécessite que la destination soit un registre physique. La source quant à elle peut être un emplacement mémoire Cette instruction est donc moins « souple » que `Add` ou `Sub`.

La gestion du `Call` est plus subtile et on y retrouve la gestion du décalage courant de la pile par rapport à l'alignement sur 16 octets que nous avions vue dans la section 11.3. À la différence de ce précédent chapitre, nous n'avons pas besoin de maintenir le décompte des octets de pile utilisés instruction après instruction : nous connaissons désormais (à l'issue de l'allocation de registres) la taille de la *stack frame* et dans l'instruction `Call` nous avons mémorisé combien d'arguments étaient passés sur la pile. À un site d'appel, le décalage courant est : $\textit{taille stack frame} + (\textit{nb arguments sur la pile} + 2) \times \textit{taille d'un entier}$. Le +2 provient du fait que dans la fonction courante, depuis son site d'appel, l'adresse de retour a été mise sur la pile, ainsi que le contenu initial de `%rbp`. En fonction du besoin d'une orrection on va insérer avant le `Call` une soustraction à `%rbp` et une addition après. Ainsi, un `Call` ERTL va possiblement générer 3 instructions RTL.

La pseudo-instruction ERTL `Frame_start` marque la première instruction d'une fonction. C'est à cet endroit qu'il faut insérer l'allocation de la *stack frame* où seront mis les *spills*. Mais, comme nous l'avons vu en 11.3, nous devons en premier lieu sauvegarder le contenu de `%rbp` sur la pile. La gestion de `Frame_end` est totalement symétrique : on défait l'allocation de *stack frame* puis on restaure `%rbp` depuis la pile.

`Stack_prm_read`, la pseudo-instruction ERTL qui permet de représenter l'accès en lecture à un paramètre passé par la pile revient simplement à un `Mov` relativement à `%rbp`. Si le pseudo-registre destination a été *spillé*, on se retrouve dans le cas de deux opérandes mémoire. Notons que l'on se retrouve donc à transférer dans un *spill*, donc sur la pile, quelque chose qui est ... déjà sur la pile. Ce n'est clairement pas efficace. La solution à ce problème

est une allocation de registres plus maligne, en plusieurs passes, prenant en compte ces arguments et tentant de réutiliser les mêmes emplacements en pile pour plusieurs *spills*. Nous nous contentons ici de la solution simple mais naïve.

Le dernier cas d'instruction particulière est le `Set`. Celle-ci va être traduite en l'une des instructions X86-64 `set`*cc* qui a la particularité de mettre 1 ou 0 (cf. section 12.4.2) mais uniquement dans un registre de 8 bits. Ainsi, le registre destination ne pourra pas être directement celui assigné par l'allocation de registres mais sa « version 8 bits ». On se donnera à cet effet une fonction *byte_reg* (r) qui prend un nom de registre physique sur 64 bits et retourne le nom du registre représentant ses 8 bits de poids faible. Ainsi par exemple, pour `%rax` nous obtiendrons le registre `%al`, pour `%rsi` ce sera le registre, `%sil`. Les noms de ces registres sont fixés et disponibles dans la documentation du processeur. La modification de ce « sous-registre » n'affecte pas les autres bits du registre 64 bits correspondant. Ainsi, si `%rax` contient initialement la valeur `0xFFFFFFFFFFFFFFFF`, si l'on fait un `set`*cc* sur `%al` qui amène à y écrire 0, le registre `%rax` aura pour valeur `0xFFFFFFFFFFFFFF00` qui n'est clairement pas égale à 0 Aussi, il conviendra d'« effacer » les bits de poids fort avant d'effectuer le `Cmp` de comparaison en insérant un « et bit-à-bit » avec la constante 1 pour ne laisser que 1 ou 0 comme valeur finale dans `%rax`.

Les règles de transformation des instructions sont données ci-dessous. Nous notons $col(r)$ la couleur assignée par le coloriage de graphe au temporaire r. Nous notons r_{tmp} le registre physique temporaire (qui sera `%r15` dans notre implémentation).

$[\![\texttt{Cqto}\ nl]\!] = \texttt{Cqto}\ nl$
$[\![\texttt{Movi}\ (i, r, nl)]\!] = \texttt{Movi}\ (i, col(r), nl)$
$[\![\texttt{Mov}\ (r_1, r_2, nl)]\!] =$
  Soient $c_1 = col(r_1)$ et $c_2 = col(r_2)$
  Si $c_1 = c_2$ alors `Nop` nl
  Sinon si au moins c_1 ou c_2 est un registre physique, `Mov` (c_1, c_2, nl)
  Sinon `Mov` $(c_1, r_{tmp}, \texttt{add\_node}\ (\texttt{Mov}\ (r_{tmp}, c_2, nl)))$
$[\![\texttt{Addi}\ (i, r, nl)]\!] = \texttt{Addi}\ (i, col(r), nl)$
$[\![\texttt{Add}\ (r_1, r_2, nl)]\!] =$
  Soient $c_1 = col(r_1)$ et $c_2 = col(r_2)$
  Si au moins c_1 ou c_2 est un registre physique, `Add` (c_1, c_2, nl)
  Sinon `Mov` $(_1, r_{tmp}, \texttt{add\_node}\ (\texttt{Add}\ (r_{tmp}, c_2, nl)))$
$[\![\texttt{Subi}\ (i, r, nl)]\!] = \texttt{Subi}\ (i, col(r), nl)$
$[\![\texttt{Sub}\ (r_1, r_2, nl)]\!] =$
  Soient $c_1 = col(r_1)$ et $c_2 = col(r_2)$
  Si au moins c_1 ou c_2 est un registre physique, `Sub` (c_1, c_2, nl)
  Sinon `Mov` $(_1, r_{tmp}, \texttt{add\_node}\ (\texttt{Add}\ (r_{tmp}, c_2, nl)))$
$[\![\texttt{Muli}\ (i, r, nl)]\!] =$
  Soit $c = col(r)$
  Si c est un registre physique, `Muli` (i, c, c, nl)
  Sinon `Mov` $(c, r_{tmp}, \texttt{add\_node}\ (\texttt{Muli}\ (i, r_{tmp}, r_{tmp}, \texttt{add\_node}\ (\texttt{Mov}\ (r_{tmp}, c, nl)))))$
$[\![\texttt{Mul}\ (r_1, r_2, nl)]\!] =$
  Soient $c_1 = col(r_1)$ et $c_2 = col(r_2)$
  Si c_2 est un registre physique, `Mul` (c_1, c_2, nl)
  Sinon `Mov` $(c_2, r_{tmp}, \texttt{add_node}\ (\texttt{Mul}\ (c_1, r_{tmp}, \texttt{add_node}\ (\texttt{Mov}\ (r_{tmp}, c_2, nl)))))$$[\![\texttt{Div}(r, nl)]\!] = \texttt{Div}\ (col(r), nl)$
$[\![\texttt{Cmp}\ (r_1, r_2, nl)]\!] =$
  Soient $c_1 = col(r_1)$ et $c_2 = col(r_2)$
  Si au moins c_1 ou c_2 est un registre physique, `Cmp` (c_1, c_2, nl)
  Sinon `Mov` $(c_1, r_{tmp}, \texttt{add\_node}\ (\texttt{Cmp}\ (r_{tmp}, c_2, nl)))$
$[\![\texttt{Cmpi}\ (i, r, nl)]\!] = \texttt{Cmpi}\ (i, col(r), nl)$
$[\![\texttt{Neg}\ (r, nl)]\!] = \texttt{Neg}\ (col(r), nl)$
$[\![\texttt{Call}\ (f, \_, nb_{stacked}, nl)]\!] =$
  Soit fs = taille de la *stack frame* de la fonction courante,
  Soit $shift = (fs + (2 + nb_{stacked}) \times 8) \,\%\, 16$,
  Si $shift \neq 0$
    Soit $adj = 16 - shift$,
    `Addi` $(-adj, \%\texttt{rsp}, \texttt{add\_node}\ (\texttt{Call}\ (f, \texttt{add\_node}\ (\texttt{Addi}\ (adj, \%\texttt{rsp}, nl)))))$
  Sinon `Call` (f, nl)

$[\![$`Jmp` $nl]\!]$ = `Jmp` nl
$[\![$`Jcc` $(cc, l_{ok}, l_{ko})]\!]$ = `Jcc` (cc, l_{ok}, l_{ko})
$[\![$`Label` $(s, comment, nl)]\!]$ = `Label` $(s, comment, nl)$
$[\![$`Nop` $nl]\!]$ = `Nop` nl
$[\![$`Pop` $(r, nl)]\!]$ = `Pop` $(col\,(r), nl)$
$[\![$`Push` $(r, nl)]\!]$ = `Push` $(col\,(r), nl)$
$[\![$`Ret`$]\!]$ = `Ret`
$[\![$`Set` $(cc, r, nl)]\!]$ =
  Soit $c = col\,(r)$
  Si c est un registre physique, `Set` $(cc, byte\_reg\,(r),$ `add_node` (`Andi` $(1, c, nl)))$
  Sinon `Mov` $(cl, r_{tmp},$ `add_node` (`Set` $(cc, byte\_reg\,(reg),$ `add_node` (`Andi` $(1, r_{tmp},$ `add_node` (`Mov` $(r_{tmp}, c, nl)))))))$
$[\![$`Frame_start` $nl]\!]$ =
  Soit nl' = si taille de la *stack frame* de la fonction courante > 0,
    `add_node` (`Addi` $(-$*taille stack frame*$,$ %`rsp`$, nl))$
    Sinon nl
  `Push` (%`rbp`, `add_node` (`Mov` (%`rsp`, %`rbp`, nl')))
$[\![$`Frame_end` $nl]\!]$ =
  Si taille de la *stack frame* de la fonction courante > 0,
    `Addi` (*taille stack frame*, %`rsp`, `add_node` (`Pop` (%`rbp`, nl)))
  Sinon `Pop` (%`rbp`, nl)
$[\![$`Stack_prm_read`$(r, o, nl)]\!]$ =
  Soit $c = col\,(r)$,
  Si c est un registre physique, `Mov_rel_rbp` (o, c, nl)
  Sinon `Mov` ((`C_spill` o), r_{tmp}, `add_node` (`Mov` $(r_{tmp}, c, nl)))$
$[\![$`Lea_rip` $(lab, r, nl)]\!]$ = `Lea_rip` $(lab, col\,(r), nl)$

12.7.3 Implantation en OCaml

L'implantation en OCaml suit les règles données dans la section précédente. Le module `Regs` doit fournir le nom du registre temporaire (r_{tmp}) ainsi que la fonction $byte\_reg\,(r)$.

```
let tmp_reg = r15

let byte_register_of_register = function
  | "%rax" → "%al" | "%rbx" → "%bl" | "%rcx" → "%cl" | "%rdx" → "%dl" | "%rbp" → "%bpl"
  | "%rsi" → "%sil" | "%rdi" → "%dil" | "%r8" → "%r8b" | "%r9" → "%r9b" | "%r10" → "%r10b"
  | "%r11" → "%r11b" | "%r12" → "%r12b" | "%r13" → "%r13b" | "%r14" → "%r14b" | "%r15" → "%r15b"
  | "%rsp" → "%spl" | _ → assert false
```

La fonction $col\,(r)$ prend en fait en argument supplémentaire les nœuds du graphe d'interférence. En effet, c'est dans ces nœuds qu'elle va trouver la couleur associée au temporaire recherché. Une subtilité subsiste : lors de la recherche d'une couleur, il est rarement possible de ne pas trouver de nœud associé à un temporaire. Cela ne peut se produire que si ledit temporaire est un registre physique, en particulier (et uniquement) `%rsp`. En effet, le rajout de correction d'alignement sur 16 octets aux sites d'appel ajoute et soustrait à `%rsp`, or ces instructions sont insérées lors de la génération du LTL, donc après le coloriage de graphe.

Afin de le raccourcir, le listing ci-dessous n'explicite le cas de toutes les instructions, seulement les plus particulières et quelques cas « standards ».

```
let find_reg_color igraph_nodes reg =
  match reg with
  | Regs.R_phys rname →Interfer.C_reg rname
```

```
    | _ →
      let reg_node = List.find (fun n →n.Interfer.n_reg = reg) igraph_nodes in
      reg_node.Interfer.n_color

  let do_instr graph igraph f_def asm =
    let colors = igraph.Interfer.g_nodes in
    match asm with
    | ERTL.Cqto next →LTL.Cqto next
    | ERTL.Movi (imm, reg, next) → LTL.Movi (imm, (find_reg_color colors reg), next)
    | ERTL.Mov (reg1, reg2, next) → (
        let col1 = find_reg_color colors reg1 in
        let col2 = find_reg_color colors reg2 in
        match (col1, col2) with
        | (Interfer.C_none, _) | (_ , Interfer.C_none) →assert false
        | ((Interfer.C_reg _), _) | (_, (Interfer.C_reg _)) →
            if col1 = col2 then LTL.Nop next else LTL.Mov (col1, col2, next)
        | ((Interfer.C_spill _), (Interfer.C_spill _)) →
            if col1 = col2 then LTL.Nop next
            else (
              let tmp_reg = Interfer.C_reg Regs.tmp_reg in
              LTL.Mov (col1, tmp_reg, Graph.add_node graph (LTL.Mov (tmp_reg, col2, next)))
             )
      )
    | ERTL.Addi (imm, reg, next) → LTL.Addi (imm, (find_reg_color colors reg), next)
    | ERTL.Add (reg1, reg2, next) → (
        let col1 = find_reg_color colors reg1 in
        let col2 = find_reg_color colors reg2 in
        match (col1, col2) with
        | (Interfer.C_none, _) | (_ , Interfer.C_none) →assert false
        | ((Interfer.C_reg _), _) | (_, (Interfer.C_reg _)) →
            LTL.Add (col1, col2, next)
        | ((Interfer.C_spill _), (Interfer.C_spill _)) →
            let tmp_reg = Interfer.C_reg Regs.tmp_reg in
            LTL.Mov (col1, tmp_reg, Graph.add_node graph (LTL.Add (tmp_reg, col2, next)))
      )
  ...
    | ERTL.Muli (imm, reg, next) → (
        let col = find_reg_color colors reg in
        match col with
        | Interfer.C_none →assert false
        | Interfer.C_reg _ →LTL.Muli (imm, col, col, next)
        | Interfer.C_spill _ →
            let tmp_reg = Interfer.C_reg Regs.tmp_reg in
            LTL.Mov (col, tmp_reg,
            Graph.add_node graph (LTL.Muli (imm, tmp_reg, tmp_reg,
            Graph.add_node graph (LTL.Mov (tmp_reg, col, next)))))
      )
    | ERTL.Mul (reg1, reg2, next) → (
        let col1 = find_reg_color colors reg1 in
        let col2 = find_reg_color colors reg2 in
        match (col1, col2) with
        | (Interfer.C_none, _) | (_ , Interfer.C_none) →assert false
        | (_, (Interfer.C_reg _)) →LTL.Mul (col1, col2, next)
        | ((Interfer.C_reg _), (Interfer.C_spill _))
        | ((Interfer.C_spill _), (Interfer.C_spill _)) →
```

```
          let tmp_reg = Interfer.C_reg Regs.tmp_reg in
          LTL.Mov (col2, tmp_reg,
          Graph.add_node graph (LTL.Mul (col1, tmp_reg,
          Graph.add_node graph (LTL.Mov (tmp_reg, col2, next)))))
    )
...
  | ERTL.Call (fun_name, _, nb_stacked_args, next) →
      let sp_shift_mod16 =
        (f_def.Interfer.fdef_frame_size + ((2 + nb_stacked_args) * RTL.word_size)) mod 16 in
      if (sp_shift_mod16 ≠0) then (
        let correct = 16 − sp_shift_mod16 in
        LTL.Addi ((Int64.of_int (−correct)), Interfer.C_reg (Regs.rsp),
        (Graph.add_node graph (LTL.Call (fun_name,
        (Graph.add_node graph (LTL.Addi ((Int64.of_int correct), Interfer.C_reg (Regs.rsp), next)))))))
       )
      else LTL.Call (fun_name, next)
  | ERTL.Jmp next →LTL.Jmp next
  | ERTL.Jcc (cc, lab_ok, lab_ko) →LTL.Jcc (cc, lab_ok, lab_ko)
...
  | ERTL.Set (cc, reg, next) → (
      let col = find_reg_color colors reg in
      match col with
      | Interfer.C_none →assert false
      | Interfer.C_reg phys →
          let low_byte_phys_reg = Interfer.C_reg (Regs.byte_register_of_register phys) in
          LTL.Set (cc, low_byte_phys_reg,
          (Graph.add_node graph (LTL.Andi (Int64.one, col, next))))
      | Interfer.C_spill _ →
          let tmp_reg = Interfer.C_reg Regs.tmp_reg in
          let low_byte_phys_reg = Interfer.C_reg (Regs.byte_register_of_register Regs.tmp_reg) in
          LTL.Mov (col, tmp_reg,
          Graph.add_node graph (LTL.Set (cc, low_byte_phys_reg,
          (Graph.add_node graph (LTL.Andi (Int64.one, tmp_reg,
          (Graph.add_node graph (LTL.Mov (tmp_reg, col, next)))))))))
    )
  | ERTL.Frame_start next →
      let adjust_rsp_or_next =
        if f_def.Interfer.fdef_frame_size > 0 then (
          Graph.add_node graph
            (LTL.Addi ((Int64.of_int (− f_def.Interfer.fdef_frame_size)), (Interfer.C_reg Regs.rsp), next)))
        else next in
      LTL.Push (Interfer.C_reg Regs.rbp,
      (Graph.add_node graph
        (LTL.Mov ((Interfer.C_reg Regs.rsp), (Interfer.C_reg Regs.rbp), adjust_rsp_or_next))))
  | ERTL.Frame_end next →
      if f_def.Interfer.fdef_frame_size > 0 then
        LTL.Addi ((Int64.of_int f_def.Interfer.fdef_frame_size), (Interfer.C_reg Regs.rsp),
        (Graph.add_node graph (LTL.Pop ((Interfer.C_reg Regs.rbp), next))))
      else LTL.Pop ((Interfer.C_reg Regs.rbp), next)
  | ERTL.Stack_prm_read (reg, offset, next) →(
      let col = find_reg_color colors reg in
      match col with
      | Interfer.C_none →assert false
      | Interfer.C_reg _ →LTL.Mov_rel_rbp (offset, col, next)
      | Interfer.C_spill _ →
```

```
            LTL.Mov ((Interfer.C_spill offset), (Interfer.C_reg Regs.tmp_reg),
            (Graph.add_node graph (LTL.Mov ((Interfer.C_reg Regs.tmp_reg), col, next))))
        )
    ...
```

Dans le cas du `Mov`, le test vérifiant si les deux opérandes sont identiques (**if** `col1 = col2`) aurait pu n'être effectué qu'une seule fois avant le filtrage au lieu d'être répété dans chaque cas du filtrage. C'est une redondance volontaire pour éviter de ne pas voir un éventuel bug dans l'implémentation qui aurait amené 2 temporaires à ne pas être coloriés (`C_none`) et à être silencieusement ignorés du fait de la suppression de l'instruction.

On peut remarquer dans le cas du `Add` (ce qui est également valable pour le `Sub` qui se comporte de manière identique) que le choix est fait de transférer l'opérande source *spillée* dans le registre temporaire. Ainsi, l'instruction se chargera directement d'aller écrire le résultat dans l'opérande destination qui reste une zone mémoire. Cela nous évite un `Mov` par rapport à si l'on avait fait l'inverse qui aurait nécessité de transférer le contenu du registre résultat dans la zone de *spill*.

Le traitement d'une définition de fonction revient simplement, à partir d'un graphe LTL vide, à parcourir toutes les instructions du graphe ERTL et à leur appliquer la fonction de traduction.

```
let do_fundef f_def =
  let graph = ref (Graph.G.empty) in
  Graph.G.iter
    (fun ertl_label asm →
      let asm' = do_instr graph f_def.Interfer.fdef_igraph f_def asm in
      graph := Graph.G.add ertl_label asm' !graph)
    f_def.Interfer.fdef_graph ;
  { LTL.fdef_name = f_def.Interfer.fdef_name ; LTL.fdef_graph = !graph ;
    LTL.fdef_entry = f_def.Interfer.fdef_entry }
```

12.8 Linéarisation

En fin de construction du LTL, le programme compilé est toujours sous la forme d'un graphe. Il est désormais nécessaire de produire un texte source assembleur, donc une représentation linéaire. Cette production de texte est réalisée par un parcours du graphe depuis sa racine, en mémorisant les nœuds déjà rencontrés (ce qui sert à éviter de boucler et à éliminer certains sauts inconditionnels).

12.8.1 Parcours du LTL

Lorsqu'un saut conditionnel est rencontré, on parcourt d'abord le sous-graphe correspondant à l'exécution implicite (test faux) avant de parcourir celui correspondant au saut.

Lorsque l'on rencontre un saut inconditionnel, si le nœud destination n'a pas encore été vu, alors on ne génère pas de `jmp`. En effet, son successeur sera la prochaine instruction que nous allons émettre, donc inutile de générer un saut à l'instruction suivante. Si au contraire on a déjà rencontré le nœud destination (et donc produit son code), on émet un `jmp` à son étiquette. Comme nous avons pris le soin de générer un `Label` à chaque destination de saut, nous sommes sûrs de trouver une étiquette à la destination du saut. Il est alors inutile de récurser sur le nœud suivant puisque l'on sait qu'on l'a déjà vu, donc délà traité : le `jmp` que l'on vient d'émettre permet justement d'aller à ce code déjà produit.

Pour toutes les autres instructions, si le nœud n'a pas déjà été vu, on émet l'instruction X86-64 correspondante, sinon on émet un saut vers ce nœud (qui est alors forcément un `Label`).

```
let get_children graph label =
  match Graph.G.find label graph with
  | LTL.Cqto next | LTL.Movi (_, _, next) | LTL.Lea_rip (_, _, next)
  | LTL.Addi (_, _, next) | LTL.Subi (_, _, next) | LTL.Mov (_, _, next)
  | LTL.Add (_, _, next) | LTL.Sub (_, _, next) | LTL.Muli (_, _, _, next)
  | LTL.Mul (_, _, next) | LTL.Div (_, next) | LTL.Cmp (_, _, next)
  | LTL.Cmpi (_, _, next) | LTL.Neg (_, next) | LTL.Call (_, next)
  | LTL.Jmp next | LTL.Nop next | LTL.Pop (_, next) | LTL.Push (_, next)
  | LTL.Label (_, _, next) | LTL.Mov_rel_rbp (_, _, next)
  | LTL.Andi (_, _, next) | LTL.Set (_, _, next) →
      [next]
  | LTL.Jcc (_, lab_ok, lab_ko) → [lab_ok ; lab_ko]
  | LTL.Ret → [ ]

let do_root out ltl_graph root =
  let seen = ref GLabelSet.empty in
  let rec rec_do n =
    let asm = Graph.G.find n ltl_graph in
    if not (GLabelSet.mem n !seen) then (
      seen := GLabelSet.add n !seen ;
      match asm with
      | LTL.Jcc (_, lab_ok, lab_ko) →
          Printf.fprintf out "  %a\n" (LTL.pp_x8664 ltl_graph) asm ;
          rec_do lab_ko ;
          rec_do lab_ok
      | LTL.Jmp next →
          if GLabelSet.mem next !seen then
            Printf.fprintf out "  %a\n" (LTL.pp_x8664 ltl_graph) asm ;
          else rec_do next
      | LTL.Label (_, _, next) →
          Printf.fprintf out "%a\n" (LTL.pp_x8664 ltl_graph) asm ;
          rec_do next
      | LTL.Nop next → rec_do next (* On élimine les Nop. *)
      | _ →
          Printf.fprintf out "  %a\n" (LTL.pp_x8664 ltl_graph) asm ;
          List.iter rec_do (get_children ltl_graph n)
    )
    else (
      match asm with
      | LTL.Label (_, _, _) →
          Printf.fprintf out "  %a\n" (LTL.pp_x8664 ltl_graph) (LTL.Jmp n)
      | _ → assert false
    ) in
  rec_do root
```

12.8.2 Émission des instructions

La fonction `LTL.pp_x8664` est une simple fonction d'impression qui émet le texte correspondant aux instructions assembleur réelles, en accord avec la syntaxe AT&T du X86-64. Chaque instruction de saut s'assure que sa destination est bien une étiquette (un `Label`).

```
let pp_x8664 graph outc = function
  | Cqto _ → Printf.fprintf outc "cqto"
```

```
    | Movi (i, reg, _) →
        Printf.fprintf outc "movq $%s,%a" (Int64.to_string i) Interfer.pp_color reg
    | Mov (reg1, reg2, _) →
        Printf.fprintf outc "movq %a,%a" Interfer.pp_color reg1 Interfer.pp_color reg2
  ...
    | Jcc (cc, lab_ok, _) →
        let dest_ok_label =
          (match Graph.G.find lab_ok graph with Label (name, _, _) →name | _ → assert false) in
        Printf.fprintf outc "j%a %s" RTL.pp_cc cc dest_ok_label
    | Nop _ → Printf.fprintf outc "nop"
    | Mov_rel_rbp (offset, reg, _) → Printf.fprintf outc "movq %d(%%rbp),%a" offset Interfer.pp_color reg
    | Lea_rip (str_lbl, reg, _) → Printf.fprintf outc "leaq %s(%%rip),%a" str_lbl Interfer.pp_color reg
```

12.8.3 Émission d'une définition de fonction

Dans le principe, émettre le code d'une fonction revient à émettre toutes ses instructions en parcourant le graphe depuis sa racine. Néanmoins, il convient de le précéder de quelques directives d'assemblage telles que celles présentées en section 11.3.4. Il faut également émettre une étiquette portant le nom de la fonction puisque des instrutions `call` vont y faire référence. Comme discuté précédemment, en fonction de l'OS hôte la syntaxe de ces directives peut varier.

Sous MacOS, avec la chaîne de compilation basée sur clang, le prélude d'une fonction `foo` sera :

```
  .section __TEXT,__text,regular,pure_instructions
  .globl  _foo
  .p2align 4, 0x90

_foo:
```

où la directive `.p2align` permet d'aligner le code qui suit sur une adresse multiple de 16 octets (2^4). Pour ce faire, si besoin elle remplit le binaire avec des instructions de code 0x90 (`nop`) jusqu'à arriver à une adresse alignée sur 16 octets. Ce code de remplissage ne sera bien sûr jamais exécuté.

Sous Linux avec la chaîne de compilation basée sur gcc, ce prélude est légèrement différent en terme de syntaxe :

```
  .text
  .globl foo
  .p2align 4, 0x90
  .type foo, @function

foo:
```

12.8.4 Conclusion du fichier assembleur

Pour la compilation de l'instruction `Print`, lors de la transformation en ERTL (cf. section 12.5.4), nous avions mémorisé les chaînes de caractères à générer pour représenter les formats des appels à `printf`. Même si dans les faits c'est toujours la chaîne `"%d\n"`, nous avions mis en place un mécanisme permettant de gérer le cas de formats différents. Ainsi, en fin de fichier il faut produire le code correspondant à ces définitions de chaînes (comme vu dans la section 11.3.4). La fonction chargée d'émettre ce code sera appelée en fin de génération de code.

```
  let emit_format_strings out =
    let glob_strs = ToERTL.get_format_strings () in
    if glob_strs ≠[ ] then (
      (match Config.asm with
```

```
    | Config.ASM_OSX_LLVM →Printf.fprintf out " .section __TEXT,__cstring,cstring_literals\n"
    | Config.ASM_Linux_gas →Printf.fprintf out " .section .rodata\n");
    let string_directive =
      (match Config.asm with Config.ASM_OSX_LLVM →".asciz" | Config.ASM_Linux_gas →".string") in
    List.iter
      (fun (str, label) →
        Printf.fprintf out "%s: %s \"%s\"\n" label string_directive(String.escaped str))
      glob_strs
  )
```

12.9 Réunir toutes les passes

Maintenant que nous avons décrit toutes les passes du compilateur, il ne reste plus qu'à les enchaîner conformément au schéma 12.1 :

```
let main_compile fname =
  let prgm = parse_file fname in
  let llasts = ToLLAst.do_program prgm in
  let rtls = ToRTL.do_program llasts in
  let ertls = List.map ToERTL.do_fundef rtls in
  let dus = List.map DefUse.do_fundef ertls in
  let igraphs = List.map Interfer.do_fundef dus in
  List.iter RegAlloc.do_fundef igraphs ;
  let ltls = List.map ToLTL.do_fundef igraphs in
  List.iter (LinearLTL.do_fundef oc) ltls ;
  LinearLTL.emit_format_strings oc
```

Pour illustrer le compilateur en action, compilons le programme affichant le nombre d'itérations nécessaires pour que la suite de Syracuse d'une valeur donnée converge vers son cycle trivial.

$$U_{n+1} = \begin{cases} \frac{U_n}{2} & \text{Si } U_n \text{ est pair} \\ 3 \times U_n + 1 & \text{Sinon} \end{cases}$$

Pour rappel, la conjecture de Syracuse (ou encore de Collatz ou d'Ulam) est l'hypothèse que cette suite atteint la valeur 1 pour tout entier strictement positif. Le problème reste ouvert.

```
syrac (int u_n) : int
begin
  int count = 0 ;
  while (u_n != 1) do
    count = count + 1 ;
    if (u_n % 2 == 0) then
      u_n = u_n / 2 ;
    else u_n = 3 * u_n + 1 ;
    endif
  done
  return count ;
end
```

```
main () : int
begin
  print (syrac (134560)) ;
  return 0 ;
end
```

```
.section __TEXT,__text,regular,pure_instructions
.globl  _syrac
.p2align 4, 0x90

_syrac:
  pushq %rbp
  movq %rsp,%rbp
lbl_8:   # syrac_body
  movq $0,%rcx
lbl_7:   # lcond
  movq $1,%rax
  cmpq %rax,%rdi
  je lbl_2
lbl_6:   # lbody
  addq $1,%rcx
  movq %rdi,%rdx
  movq $2,%rsi
  movq %rdx,%rax
  cqto
  idivq %rsi
  movq $0,%rax
  cmpq %rax,%rdx
  jne lbl_4
lbl_5:   # then
  movq %rdi,%rax
  movq $2,%rsi
  cqto
  idivq %rsi
  movq %rax,%rdi
lbl_3:   # endif
  jmp lbl_7
lbl_4:   # else
  imulq $3,%rdi, %rdi
  addq $1,%rdi
  jmp lbl_3
lbl_2:   # lexit
  movq %rcx,%rax
lbl_1:   # fun postlude
  popq %rbp
  ret

.section __TEXT,__text,regular,pure_instructions
.globl  _main
.p2align 4, 0x90

_main:
  pushq %rbp
  movq %rsp,%rbp
lbl_10:   # main_body
  movq $134560,%rdi
  call _syrac
  movq %rax,%rsi
  leaq lstr1(%rip),%rdi
  movq $0,%rax
  call _printf
  movq $0,%rax
lbl_9:   # fun postlude
  popq %rbp
  ret

.section __TEXT,__cstring,cstring_literals
lstr1: .asciz "%ld\n"
```

L'examen de ce code produit appelle plusieurs commentaires. Tout d'abord, pourquoi ne voit-on pas en début de fonction, la sauvegarde des registres à charge de l'appelé (*callee saved*, qui contient les registres `%rbx`, `%r12`, `%r13` et `%r14`) ? Dans le `main`, c'est normal puisque le compilateur omet cette sauvegarde, puisqu'il n'y a pas à proprement parler de fonction appelante. Dans `syrac`, aucun de ces registres *callee saved* n'est utilisé. Aussi, lors de l'allocation de registres, les temporaires assignés à la sauvegarde de ces registres ont été coloriés par ces mêmes registres (on a eu de la chance !). Ainsi, lors du passage en LTL, on s'est retrouvé avec des `mov` entre les mêmes registres, qui ont donc été éliminés.

Lorsque l'on regarde les deux instructions à l'étiquette `lbl_7`, on se rend compte qu'il aurait été plus efficace d'utiliser un `cmpq` avec la constante immédiate 1. La raison est que dans notre compilateur, lors de la sélection d'instructions (construction du LLAST) nous n'avons pas simplifié les expressions de comparaison (pas de *smart constructor*). Il en découle une utilisation inefficace de registres.

Dans le `main`, aucune correction d'alignement sur 16 octets n'est faite avant l'appel à `syrac`. C'est normal : au moment de « l'appel » à `main`, la pile était alignée, puis l'adresse de retour (8 octets) a été empilée de manière transparente, enfin le `push %rbp` a rajouté 8 octets. Le compte est bon, 16 octets, le site de l'appel à `syrac` est bien aligné.

Globalement, aucun temporaire n'a été *spillé*, c'est une bonne nouvelle. D'autre part, tous les arguments de fonctions ont été passés par les registres réservés à cet effet. Ainsi, nous ne constatons aucun accès à la pile.

Pour terminer, à l'étiquette `lbl_2` et juste avant `lbl_9`, nous voyons les transferts de la valeur de retour des fonctions dans `%rax`.

Cet exemple montre un cas où l'allocation de registres a pu être relativement efficace, ne nécessitant pas de *spill*. C'est dû en partie au fait que le programme comporte peu de variables et peu d'appels de fonction. Dans beaucoup de cas, l'allocation (un peu) naïve que nous avons implantée ne permet pas cette économie de *spill*. Si l'on regarde le code produit pour le calcul naïf de la somme des n premiers entiers (par une inefficace boucle) nous voyons apparaître deux points intéressants.

```
sumint (int n) : int
begin
  if n == 0 then
    return 0 ;
  else
    return n + sumint (n - 1) ;
  endif
end
```

```
_sumint:
  pushq %rbp
  movq %rsp,%rbp
  addq $-8,%rsp
lbl_5:   # sumint_body
  movq $0,%rax
  cmpq %rax,%rdi
  jne lbl_3
lbl_4:   # then
  movq $0,%rax
lbl_1:   # fun postlude
  addq $8,%rsp
  popq %rbp
  ret
lbl_3:   # else
  movq %rdi,-8(%rbp)
  subq $1,%rdi
  addq $-8,%rsp
  call _sumint
  addq $8,%rsp
  addq %rax,-8(%rbp)
  movq -8(%rbp),%rax
  jmp lbl_1
```

D'une part on remarque qu'un *spill* a été utilisé pour représenter `n`. D'autre part, on remarque une correction de pile pour l'alignement sur 16 octets juste avant l'appel récursif (juste avant l'instruction `call`). En effet, au site d'appel précédent à `syrac`, la pile était alignée, une fois arrivé dans la fonction, l'adresse de retour et `%rbp` ont été empilés et le *spill* a été alloué. Nous avons donc un décalage de $3 \times 8 = 24$ octets qu'il faut donc compléter par 8 octets supplémentaires pour arriver à 32 qui est bien un multiple de 16. Cette correction est immédiatement annulée au retour de l'appel.

12.10 Conclusion

Ce chapitre a volontairement présenté la structure d'un compilateur simplifié en dépit de la complexité que nous pouvons toutefois constater. Il a permis d'aborder plus profondément les problèmes liés aux ressources physiques de la machine (instructions, registres, pile, contraintes d'alignement, etc.).

Les grandes absentes de cette présentation sont les optimisations. En effet, les compilateurs modernes effectuent des analyses complexes afin de rendre le code généré plus compact ou plus rapide (élimination des récursions terminales, déroulage de boucles, inversions de tests, partage de sous-expressions communes, ré-ordonnancement d'instructions, et bien d'autres choses encore). Certaines s'appuient sur une représentation intermédiaire en forme SSA (*Single Static Assignment*) que nous n'avons pas étudiée.

D'autre part, comme déjà précisé, l'allocation de registres présentée ici reste un peu naïve. Son implantation est également très naïve (nous n'utilisons aucune structure maligne qui permettrait de trouver efficacement les voisins des nœuds, les candidats aux différentes opérations, etc.).

Pour terminer, la vie a été grandement simplifiée par le fait que notre langage ne permet de gérer que des entiers signés sur 64 bits. Si nous avions eu d'autres types de données, des modificateurs de taille ou de signes, des pointeurs, des tableaux, alors de nouvelles complications seraient arrivées. De même, notre langage ne dispose pas de variables globales.

Pour aller plus loin, la lecture du livre suivant est extrêment instructive : *"Modern Compiler Implementation in ML"*, 1998, Andrew W. Appel, Cambridge University Press, ISBN 0-521-60764-7. Notons que la littérature sur la compilation est florissante, avec de nombreux autres ouvrages et articles également dignes du plus grand intérêt.

Chapitre 13

Annexes

On trouve dans cette annexe quelques documents qui présentent de façon succinte les éléments essentiels du langage OCaml, des outils de compilation qu'il offre, de sa bibliothèque standard.

13.1 OCaml en quelques pages

Documents réalisés par Fabrice Le Fessant (http://fabrice.lefessant.net/)

Syntax

Implementations are in `.ml` files, interfaces are in `.mli` files.
Comments can be nested, between delimiters `(*...*)`
Integers: `123`, `1_000`, `0x4533`, `0o773`, `0b1010101`
Chars: `'a'`, `'\255'`, `'\xFF'`, `'\n'` Floats: `0.1`, `-1.234e-34`

Data Types

`unit`	Void, takes only one value: `()`
`int`	Integer of either 31 or 63 bits, like `32`
`int32`	32 bits Integer, like `32l`
`int64`	64 bits Integer, like `32L`
`float`	Double precision float, like `1.0`
`bool`	Boolean, takes two values: `true` or `false`
`char`	Simple ASCII characters, like 'A'
`string`	Strings of chars, like `"Hello"`
`'a list`	Lists, like `head :: tail` or `[1;2;3]`
`'a array`	Arrays, like `[|1;2;3|]`
t_1 * ... * t_n	Tuples, like `(1,"foo", 'b')`

Constructed Types

`type record =`	new record type
`{` *field1* `: bool;`	immutable field
`mutable` *field2* `: int; }`	mutable field
`type enum =`	new variant type
`| Constant`	Constant constructor
`| Param of string`	Constructor with arg
`| Pair of string * int`	Constructor with args

Constructed Values

```
let r = { field1 = true; field2 = 3; }
let r' = { r with field1 = false }
r.field2 <- r.field2 + 1;
let c = Constant
let c' = Param "foo"
let c'' = Pair ("bar",3)
```

References, Strings and Arrays

`let x = ref 3`	integer reference (mutable)
`x := 4`	reference assignation
`print_int !x;`	reference access
`s.[0]`	string char access
`s.[0] <- 'a'`	string char modification
`t.(0)`	array element access
`t.(0) <- x`	array element modification

Imports — Namespaces

`open Unix;;`	global open
`let open Unix in` *expr*	local open
`Unix.(`*expr*`)`	local open

Functions

`let f x =` *expr*	function with one arg
`let rec f x =` *expr*	recursive function
apply:	`f x`
`let f x y =` *expr*	with two args
apply:	`f x y`
`let f (x,y) =` *expr*	with a pair as arg
apply:	`f (x,y)`
`List.iter (fun x ->` *e*`) l`	anonymous function
`let f= function None ->` *act*	function definition
`| Some x ->` *act*	by cases
apply:	`f (Some x)`
`let f ~str ~len =` *expr*	with labeled args
apply:	`f ~str:s ~len:10`
apply (for `~str:str`):	`f ~str ~len`
`let f ?len ~str =` *expr*	with optional arg (`option`)
`let f ?(len=0) ~str =` *expr*	optional arg default
apply (with omitted arg):	`f ~str:s`
apply (with commuting):	`f ~str:s ~len:12`
apply (`len: int option`):	`f ?len ~str:s`
apply (explicitely ommited):	`f ?len:None ~str:s`
`let f (x : int) =` *expr*	arg has constrainted type
`let f : 'a 'b. 'a*'b -> 'a = fun (x,y) -> x`	function with constrainted polymorphic type

Modules

`module M = struct .. end`	module definition
`module M: sig .. end= struct .. end`	module and signature
`module M = Unix`	module renaming
`include M`	include items from
`module type Sg = sig .. end`	signature definition
`module type Sg = module type of M`	signature of module
`let module M = struct .. end in ..`	local module
`let m = (module M : Sg)`	to 1^{st}-class module
`module M = (val m : Sg)`	from 1^{st}-class module
`module Make(S: Sg) = struct .. end`	functor
`module M = Make(M')`	functor application

Module type items:
`val`, `external`, `type`, `exception`, `module`, `open`, `include`, `class`

Pattern-matching

`match` *expr* `with`	
`|` *pattern* `->` *action*	
`|` *pattern* `when` *guard* `->` *action*	conditional case
`| _ ->` *action*	default case

Patterns:

`| Pair (x,y) ->`	variant pattern
`| { field = 3; _ } ->`	record pattern
`| head :: tail ->`	list pattern
`| [1;2;x] ->`	list-pattern
`| (Some x) as y ->`	with extra binding
`| (1,x) | (x,0) ->`	or-pattern

Conditionals

Structural	Physical	
`=`	`==`	Polymorphic Equality
`<>`	`!=`	Polymorphic Inequality

Polymorphic Generic Comparison Function: `compare`

	x < y	x = y	x > y
`compare x y`	-1	0	1

Other Polymorphic Comparisons : `>`, `>=`, `<`, `<=`

Loops

```
while cond do ... done;
for var = min_value to max_value do ... done;
for var = max_value downto min_value do ... done;
```

Exceptions

`exception MyExn`	new exception
`exception MyExn of t * t'`	same with arguments
`exception MyFail = Failure`	rename exception with args
`raise MyExn`	raise an exception
`raise (MyExn (`*args*`))`	raise with args
`try` *expression*	catch `MyException` if raised
`with Myn -> ...`	in *expression*

Objects and Classes

`class virtual foo x =`	virtual class with arg
`let y = x+2 in`	init before object creation
`object (self: 'a)`	object with self reference
`val mutable variable = x`	mutable instance variable
`method get = variable`	accessor
`method set z =`	
`variable <- z+y`	mutator
`method virtual copy : 'a`	virtual method
`initializer`	init after object creation
`self#set (self#get+1)`	
`end`	
`class bar =`	non-virtual class
`let var = 42 in`	class variable
`fun z -> object`	constructor argument
`inherit foo z as super`	inheritance and ancestor reference
`method! set y =`	method explicitely overriden
`super#set (y+4)`	access to ancestor
`method copy = {< x = 5 >}`	copy with change
`end`	
`let obj = new bar 3`	new object
`obj#set 4; obj#get`	method invocation
`let obj = object .. end`	immediate object

Polymorphic variants

`type t = [ 'A | 'B of int ]`	closed variant
`type u = [ 'A | 'C of float ]`	
`type v = [ t | u | ]`	union of variants
`let f : [< t ] -> int = function`	argument must be
`| 'A -> 0 | 'B n -> n`	a subtype of `t`
`let f : [> t ] -> int = function`	`t` is a subtype
`| 'A -> 0 | 'B n -> n | _ -> 1`	of the argument

OCaml Standard Tools

OCaml v. 3.12.0 — June 7, 2011 — Copyright © 2011 OCamlPro SAS — **http://www.ocamlpro.com/**

Standard Tools

`.opt` tools are the same tools, compiled in native-code, thus much faster.

`ocamlopt[.opt]`	native-code compiler
`ocamlc[.opt]`	bytecode compiler
`ocaml`	interactive bytecode toplevel
`ocamllex[.opt]`	lexer compiler
`ocamlyacc`	parser compiler
`ocamldep[.opt]`	dependency analyser
`ocamldoc`	documentation generator
`ocamlrun`	bytecode interpreter

Compiling

A unit interface must be compiled before its implementation. Here, `ocamlopt` can replace `ocamlc` anywhere to target asm.

`ocamlc -c test.mli`	compile an interface
`ocamlc -c test.ml`	compile an implementation
`ocamlc -a -o lib.cma test.cmo`	generate a library
`ocamlc -o prog test.cmo`	generate an executable
`ocamlopt -shared -o p.cmxs test.cmx`	generate a plugin

Generic Arguments

`-config`	print config and exit
`-c`	do not link
`-o` *target*	specify the target to generate
`-a`	build a library
`-pp` *prepro*	use a preprocessor (often `camlp4`)
`-I` *directory*	search directory for dependencies
`-g`	add debugging info
`-annot`	generate source navigation information
`-i`	print inferred interface
`-thread`	generate thread-aware code
`-linkall`	link even unused units
`-nostdlib`	do not use installation directory
`-nopervasives`	do not autoload `Pervasives`

Linking with C

`-cc` *gcc*	use as C compiler/linker
`-cclib` *option*	pass option to the C linker
`-ccopt` *option*	pass option to C compiler/linker
`-output-obj`	link, but output a C object file
`-noautolink`	do not automatically link C libraries

Errors and Warnings

Warnings default is `+a-4-6-7-9-27..29`

`-w` *wlist*	set or unset warnings
`-warn-errors` *wlist*	set or unset warnings as errors
`-warn-help`	print description of warnings
`-rectypes`	allow arbitrarily recursive types

Native-code Specific Arguments

`-p`	compile or link for profiling with `gprof`
`-inline` *size*	set maximal function size for inlining
`-unsafe`	remove array bound checks

Bytecode Specific Arguments

`-custom`	link with runtime and C libraries
`-make-runtime`	generate a pre-customized runtime
`-use-runtime` *runtime*	use *runtime* instead of `ocamlrun`

Packing Arguments

`-pack -o` *file.cmo/.cmx*	pack several units in one unit
`-c -for-pack` *File*	compile unit to be packed into *File*

Interactive Toplevel

Use `;;` to terminate and execute what you typed.

Building your own: `ocamlmktop -o unixtop unix.cma`

`#load "lib.cma";;`	load a compiled library/unit
`#use "file.ml";;`	compile and run a source file
`#directory "dir";;`	add directory to search path
`#trace` *function*`;;`	trace calls to function
`#untrace` *function*`;;`	stop tracing calls to function
`#quit;;`	quit the toplevel

System Variables

`OCAMLLIB`	Installation directory
`OCAMLRUNPARAM`	Runtime settings (e.g. `b,s=256k,v=0x015`)

Flags				
	`p`	ocamlyacc parser trace	`b`	print backtrace
	`i`	major heap increment	`s`	minor heap size
	`O`	compaction overhead	`o`	space overhead
	`s`	stack size	`h`	initial heap size
	`v`	GC verbosity		

Files Extensions

Sources		Objects	
.ml	implementation	.cmo	bytecode object
		.cmx + .o	asm object
.mli	interface	.cmi	interface object
.mly	parser	.cma	bytecode library
.mll	lexer	.cmxa + .a	native library
		.cmxs	native plugin

Generating Documentation

Generate documentation for source files:
`ocamldoc` *format* `-d` *directory sources.mli*

where *format* is:		
	`-html`	Generate HTML
	`-latex`	Generate LaTeX
	`-texi`	Generate TeXinfo
	`-man`	Generate man pages

Parsing `ocamlyacc grammar.mly`

will generate `grammar.mli` and `grammar.ml` from the grammar specification.

`-v` generates `grammar.output` file with debugging info

```
%{
  header
%}
  declarations
%%
  rules
%%
  trailer
```

Declarations:

`%token` *token* `%token <`*type*`>` *token* `%type <`*type*`>` *symbol*
`%start` *symbol* `%left` *symbol* `%right` *symbol* `%nonassoc` *symbol*

Rules:

nonterminal `:`
symbol ... symbol `{` *action* `}`
`|` ...
`|` *symbol ... symbol* `{` *action* `}` `;`

Lexing `ocamllex lexer.mll`

will generate `lexer.ml` from the lexer specification.

`-v` generates `lexer.output` file with debugging info

```
{ header }
let ident = regexp ...
rule entrypoint args =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint args =
  parse ...
and ...
{ trailer }
```

`Lexing.lexeme lexbuf` in *action* to get the current token.

Computing Dependencies

`ocamldep` can be used to automatically compute dependencies. It takes in arguments all the source files (.ml and .mli), and some standard compiler arguments:

`-pp` *prepro*	call a preprocessor
`-I` *dir*	search directory for dependencies
`-modules`	print modules instead of Makefile format
`-slash`	use \ instead of /

Generic Makefile Rules

```
.SUFFIXES: .mli .mll .mly .ml .cmo .cmi .cmx
.ml.cmo :
        ocamlc -c $(OFLAGS) $(INCLUDES) $<
.mli.cmi :
        ocamlc -c $(OFLAGS) $(INCLUDES) $<
.ml.cmi :
        ocamlc -c $(OFLAGS) $(INCLUDES) $<
.ml.cmx :
        ocamlopt -c $(OFLAGS) $(INCLUDES) $<
.mll.ml :
        ocamllex $(OLEXFLAGS) $<
.mly.ml :
        ocamlyacc $(OYACCFLAGS) $<
.mly.mli:
        ocamlyacc $(OYACCFLAGS) $<
```

Standard Modules

Basic Data Types

Pervasives	All basic functions
String	Functions on Strings
Array	Functions on Polymorphic Arrays
List	Functions on Polymorphic Lists
Char	Functions on Characters
Int32	Functions on 32 bits Integers
Int64	Functions on 64 bits Integers
Nativeint	Functions on Native Integers

Advanced Data Types

Buffer	Automatically resizable strings
Complex	Complex Numbers
Digest	MD5 Checksums
Hashtbl	Polymorphic Hash Tables
Queue	Polymorphic FIFO
Stack	Polymorphic LIFO
Stream	Polymorphic Streams
Map	Dictionaries (functor)
Set	Sets (functor)

System

Arg	Argument Parsing
Filename	Functions on Filenames
Format	Pretty-Printing
Genlex	Simple OCaml-Like Lexer
Marshal	Serialization Functions
Lexing	Functions for ocamllex
Parsing	Functions for ocamlyacc
Printexc	Generic Exception Printer
Random	Random Number Generator
Printf	printf-like Functions
Scanf	scanf-like Functions
Sys	OS Low-level Functions

Tweaking

Lazy	Functions on Lazy Values
Gc	Garbage Collection Tuning
Weak	Weak Pointers (GC)

Popular Functions per Module

module Hashtbl

```
let t = Hashtbl.create 117
Hashtbl.add t key value;
let value = Hashtbl.find t key
Hashtbl.iter (fun key value -> ... ) t;
let cond = Hashtbl.mem t key
Hashtbl.remove t key;
Hashtbl.clear t;
```

module List

```
let len = List.length l
List.iter (fun ele -> ... ) l;
let l' = List.map(fun ele -> ... ) l
let l' = List.rev l1
let acc' = List.fold_left (fun acc ele -> ...) acc l
let acc' = List.fold_right (fun ele acc -> ...) l acc
if List.mem ele l then ...
if List.for_all (fun ele -> ele >= 0) l then ...
if List.exists (fun ele -> ele < 0) l then ...
let neg = List.find (fun x -> x < 0) ints
let negs = List.find_all (fun x -> x < 0) ints
let (negs,pos) = List.partition (fun x -> x < 0) ints
let ele = List.nth 2 list
let head = List.hd list
let tail = List.tl list
let value = List.assoc key assocs
if List.mem_assoc key assocs then ...
let assocs = List.combine keys values
let (keys, values) = List.split assocs
let l' = List.sort compare l
let l = List.append l1 l2 or l1 @ l2
let list = List.concat list_of_lists
```

Functions using Physical Equality in List

memq, assq, mem_assq

Non-tail Recursive Functions in List

append, concat, @, map, fold_right, map2, fold_right2, remove_assoc, remove_assq, split, combine,merge

module String

```
let s = String.create len
let s = String.make len char
let len = String.length s
let char = s.[pos]
s.[pos] <- char;
let concat = prefix ^ suffix
let s' = String.sub s pos len'
let s = String.concat "," list_of_strings
let pos = String.index_from s pos char_to_find
let pos = String.rindex_from s pos char_to_find
String.blit src src_pos dst dst_pos len;
let s' = String.copy s
let s' = String.uppercase s
let s' = String.lowercase s
let s' = String.escaped s
String.iter (fun c -> ...) s;
if String.contains s char then ...
```

module Array

```
let t = Array.create len v
let t = Array.init len (fun pos -> v_at_pos)
let v = t.(pos)
t.(pos) <- v;
let len = Array.length t
let t' = Array.sub t pos len
let t = Array.of_list list
let list = Array.to_list t
Array.iter (fun v -> ... ) t;
Array.iteri (fun pos v -> ... ) t;
let t' = Array.map (fun v -> ... ) t
let t' = Array.mapi (fun pos v -> ... ) t
let concat = Array.append prefix suffix
Array.sort compare t;
```

module Char

```
let ascii_65 = Char.code 'A'
let char_A = Char.chr 65
let c' = Char.lowercase c
let c' = Char.uppercase c
let s = Char.escaped c
```

module Buffer

```
let b = Buffer.create 10_000
Printf.bprintf b "Hello %s\n" name
Buffer.add_string b s;
Buffer.add_char b '\n';
let s = Buffer.contents s
```

module Digest

```
let md5sum = Digest.string str
let md5sum = Digest.substring str pos len
let md5sum = Digest.file filename
let md5sum = Digest.channel ic len
let hexa = Digest.to_hex md5sum
```

module Filename

```
if Filename.check_suffix name ".c" then ...
let file = Filename.chop_suffix name ".c"
let file = Filename.basename name
let dir = Filename.dirname name
let name = Filename.concat dir file
if Filename.is_relative file then ...
let file = Filename.temp_file prefix suffix
let file = Filename.temp_file ~temp_dir:"." pref suf
```

Conclusion

Les quelques principes des langages de programmation que nous avons vus dans ce cours tendent à souligner l'importance de la précision de la définition des langages de programmation. Le diable se cache dans les détails : c'est aussi vrai en programmation. Ce cours avait pour objectif de vous faire prendre conscience de ces détails, et de vous familiariser avec les concepts de base des langages comme la portée des variables, le typage et la gestion de la mémoire, ainsi qu'avec des concepts de plus haut niveau comme les fonctions et les procédures, les structures de données et les objets.

Table des matières