École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Édition 2023–2024

Avant-propos

Ce polycopié rassemble les notes d'un cours donné à l'École Normale Supérieure depuis 2008 et à l'École Polytechnique depuis 2016, intitulé *Langages de programmation et compilation*.

Pour bien programmer, il faut comprendre le modèle d'exécution du ou des langages de programmation qu'on utilise, c'est-à-dire la manière dont les différentes constructions sont exécutées, que ce soit au travers d'un interprète ou d'un compilateur. Il est donc important de comprendre tous les mécanismes qui sont en jeu dans ce processus. Les mettre en œuvre soi-même en écrivant un compilateur constitue sans doute la meilleure façon de comprendre un langage. Même si on ne cherche pas à réaliser un compilateur, de nombreux concepts sous-jacents peuvent être réutilisés dans d'autres contextes. Ainsi, les notions de sémantique formelle, de syntaxe abstraite ou encore d'analyse syntaxique sont utilisées tous les jours dans des domaines aussi vastes que les bases de données, la démonstration assistée par ordinateur ou encore la théorie des langages de programmation.

Ce cours ne s'appuie sur aucun ouvrage et est censé se suffire à lui-même. Cependant, certains sujets ne sont que brièvement évoqués et le lecteur désireux d'en savoir plus est renvoyé vers d'autres ouvrages en fin de chapitres. Par exemple, si ce cours aborde plusieurs notions de théorie des langages (grammaires, expressions régulières, automates, etc.) pour les besoins de l'analyse syntaxique, il ne saurait se substituer à un vrai cours sur le sujet. Le lecteur désireux d'en apprendre d'avantage sur les langages formels pourra consulter avec intérêt le livre d'Olivier Carton [5].

Ce cours ne suppose pas non plus de connaissances particulières en matière de langages de programmation. On utilise ponctuellement le langage OCaml pour programmer certaines notions, mais cela reste épisodique. À d'autres moments, on explique des points particuliers de certains langages comme Java, C et C++, notamment dans la section 6.2, mais ces passages doivent pouvoir être lus sans pour autant connaître ces langages. Le chapitre 7 se focalise sur la compilation d'un langage comme OCaml, mais ce fragment est expliqué au préalable en détail dans le chapitre 2. De même, le chapitre 8 se focalise sur la compilation de Java, mais il commence par en expliquer les concepts.

Ce cours démarre par un chapitre sur l'assembleur x86-64. C'est une manière de se jeter dans le bain de la compilation que de démarrer par le langage cible, c'est-à-dire l'objectif final. Ensuite, on perd de vue un moment l'assembleur pour prendre le temps

de poser tous les concepts amont de la compilation. Ce n'est qu'avec la partie III que l'assembleur refera son apparition.

Remerciements. Une partie de ce cours est librement inspirée de notes de cours de Xavier Leroy et d'un cours de compilation donné à l'École Polytechnique par François Pottier entre 2005 et 2015, avec l'aimable autorisation de leurs auteurs. Je les remercie très sincèrement. Je remercie également Didier Rémy pour son excellent paquet LaTeX `exercise`. Enfin, je remercie très chaleureusement toutes les personnes qui ont contribué à ce polycopié par leur relecture attentive ou la suggestion d'exercices ou qui ont assuré les TD de ces cours à l'École Normale Supérieure ou à l'École Polytechnique, à savoir Léo Andrès, Pierre-Gabriel Berlureau, Julien Bertrane, Lélio Brun, Basile Clément, Naëla Courant, Gurvan Debaussart, Quentin Garchery, Léon Gondelman, Jacques-Henri Jourdan, Gaëtan Leurent, Louis Mandel, Valeran Maytié, Simão Melo de Sousa, Robin Morisset, Mário Pereira et Philippe Volte--Vieira.

L'auteur peut être contacté par courrier électronique à l'adresse suivante :

`Jean-Christophe.Filliatre@cnrs.fr`

Historique.

— Version 1 : 7 mars 2017
— Version 2 : 7 décembre 2017
— Version 3 : 5 décembre 2018
— Version 4 : 16 décembre 2019
— Version 5 : 7 décembre 2020
— Version 6 : 17 décembre 2021
— Version 7 : 15 décembre 2022
— Version 8 : 7 décembre 2023

Compilé à partir du commit `0b35ddfc`.

Table des matières

I Préliminaires **1**

1 Assembleur x86-64 **3**
1.1 Arithmétique des ordinateurs 3
1.2 L'architecture x86-64 4
1.3 Le défi de la compilation 10
1.4 Un exemple de compilation 13

2 Qu'est-ce qu'un langage de programmation **17**
2.1 Syntaxe abstraite 18
2.2 Sémantique opérationnelle 20
2.2.1 Sémantique à grands pas 22
2.2.2 Sémantique à petits pas 28
2.2.3 Équivalence des deux sémantiques 31
2.3 Interprète 33
2.4 Langages impératifs 37
2.5 Preuve de correction d'un compilateur 38

II Partie avant du compilateur **43**

3 Analyse lexicale **45**
3.1 Expression régulière 46
3.2 Automate fini 47
3.3 Analyseur lexical 50
3.4 L'outil `ocamllex` 51

4 Analyse syntaxique **55**
4.1 Analyse syntaxique élémentaire 55
4.2 Grammaire 59
4.3 Analyse descendante 66
4.4 Analyse ascendante 69
4.5 L'outil `menhir` 75

5 Typage statique 79
- 5.1 Typage simple de Mini-ML . . . 80
- 5.2 Sûreté du typage . . . 83
- 5.3 Polymorphisme paramétrique . . . 85
- 5.4 Inférence de types . . . 89

III Partie arrière du compilateur 95

6 Passage des paramètres 97
- 6.1 Stratégie d'évaluation . . . 97
- 6.2 Comparaison des langages Java, OCaml, C et C++ . . . 99
- 6.3 Compilation d'un mini Pascal . . . 108

7 Compilation des langages fonctionnels 115
- 7.1 Fonctions comme valeurs de première classe . . . 115
- 7.2 Optimisation des appels terminaux . . . 120
- 7.3 Filtrage . . . 123

8 Compilation des langages à objets 131
- 8.1 Brève présentation des concepts objets avec Java . . . 131
- 8.2 Compilation de Java . . . 138
- 8.3 Quelques mots sur C++ . . . 145

9 Compilateur optimisant 151
- 9.1 Sélection d'instructions . . . 151
- 9.2 Production de code RTL . . . 156
- 9.3 Production de code ERTL . . . 159
- 9.4 Production de code LTL . . . 164
  - 9.4.1 Analyse de durée de vie . . . 164
  - 9.4.2 Graphe d'interférence . . . 166
  - 9.4.3 Coloriage du graphe d'interférence . . . 168
  - 9.4.4 Traduction vers LTL . . . 172
- 9.5 Production de code assembleur x86-64 . . . 175

Annexes 181

A Solutions des exercices 181

B Petit lexique français-anglais de la compilation 197

Bibliographie 199

Index 201

Première partie

Préliminaires

1 Assembleur x86-64

Nous faisons le choix d'utiliser ici l'architecture x86-64, car il s'agit d'une architecture très répandue aujourd'hui. Néanmoins, beaucoup de ce que nous allons expliquer reste valable pour d'autres architectures.

L'architecture x86-64 est issue d'une longue lignée d'architectures conçues par Intel depuis 1986 puis par AMD et Intel depuis 2000. Elle appartient à la famille CISC, acronyme pour *Complex Instruction Set Computing*. Cela signifie que son jeu d'instructions est vaste, avec notamment beaucoup d'instructions permettant d'accéder en lecture ou écriture à la mémoire, à l'opposé de la famille RISC (pour *Reduced Instruction Set Computing*) qui cherche au contraire à limiter le jeu d'instructions. Un exemple d'architecture de la famille RISC est ARM (pour *Advanced RISC Machine*).

1.1 Arithmétique des ordinateurs

Dans la machine, les nombres sont représentés en base deux. Un chiffre en base deux vaut 0 ou 1 et est appelé un *bit* (de l'anglais *binary digit*). Un entier représenté sur n bits est conventionnellement écrit avec ses chiffres les moins significatifs à droite et ses chiffres les plus significatifs à gauche, exactement comme on le fait en base 10.

b_{n-1}	b_{n-2}	$\dots$	b_1	b_0

Les bits b_{n-1}, b_{n-2}, etc. sont dits de *poids fort* et les bits b_0, b_1, etc. sont dits de *poids faible*. Le nombre n de bits est typiquement égal à 8, 16, 32 ou 64. Lorsque $n = 8$ on parle d'un *octet* (en anglais *byte*).

Si on interprète un tel entier comme *non signé*, c'est-à-dire comme un entier naturel, sa valeur est donnée par

$$\sum_{i=0}^{n-1} b_i 2^i$$

On a au total 2^n valeurs possibles, de 0 ($000\dots000_2$) à $2^n - 1$ ($111\dots111_2$) [1]. Avec $n = 8$, on peut représenter tous les entiers de 0 à 255. Ainsi, l'octet 00101010_2 représente l'entier 42, car $42 = 2^5 + 2^3 + 2^1$.

1. On utilise un 2 en indice pour bien préciser qu'il s'agit d'un entier en base 2.

Pour représenter un entier *signé*, on utilise le *complément à deux*, qui interprète le bit de poids fort b_{n-1} différemment des autres, de la façon suivante :

$$-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

bits	valeur
100...000	-2^{n-1}
100...001	$-2^{n-1}+1$
$\vdots$	$\vdots$
111...110	-2
111...111	-1
000...000	0
000...001	1
000...010	2
$\vdots$	$\vdots$
011...110	$2^{n-1}-2$
011...111	$2^{n-1}-1$

Le bit b_{n-1} est alors appelé *bit de signe*. Lorsque $b_{n-1} = 0$, on retrouve la représentation d'un entier non signé sur $n-1$ bits, avec des valeurs allant de 0 (000...000) à $2^{n-1}-1$ (011...111). Si en revanche $b_{n-1} = 1$, on a des valeurs allant de -2^{n-1} ($100\dots000_2$) à -1 ($111\dots111_2$). Ainsi, l'entier -42 est représenté par 11010110 car $-42 = -128+86 = -2^7+2^6+2^4+2^2+2^1$. Cette représentation dissymétrique peut paraître étrange à première vue, mais elle a de nombreux avantages. En particulier, pour augmenter la taille d'un entier, il suffit de répliquer le bit de signe dans tous les nouveaux bits de poids fort ; on parle d'*extension de signe*.

Il est important de comprendre que lorsque n bits sont rangés en mémoire, rien ne permet de déterminer s'il s'agit d'un entier signé ou non signé. C'est le contexte qui va déterminer si on interprète ou non le bit b_{n-1} comme un bit de signe. Ainsi, le même octet 11010110_2 peut être interprété comme l'entier signé -42 ou comme l'entier non signé 214 dans des contextes différents.

La mémoire de l'ordinateur peut être vue comme un grand tableau contenant des octets. On accède à un octet particulier de la mémoire avec son *adresse*, c'est-à-dire un entier compris entre 0 et $N-1$ si la mémoire contient N octets. Si par exemple on dispose d'un Go de mémoire, cela signifie que $N = 10^9$. Les octets stockés en mémoire peuvent représenter aussi bien des données (des entiers, des chaînes de caractères, des adresses, etc.) que des instructions. C'est le contexte qui détermine comment on interprète les octets situés à une certaine adresse, et en particulier si plusieurs octets consécutifs doivent être interprétés comme un tout. Pour la machine, il n'y a pas de différence entre deux entiers 16 bits consécutifs dans la mémoire et un entier 32 bits.

Dans la suite, on utilisera aussi le mot *pointeur* pour désigner une adresse, même si en toute rigueur un pointeur désigne plutôt une variable ou une expression de programme dont la valeur est une adresse.

1.2 L'architecture x86-64

Comme son nom le suggère, l'architecture x86-64 est une architecture 64 bits, ce qui signifie que les opérations arithmétiques, logiques et de transfert de/vers la mémoire se font avec des entiers représentés sur 64 bits. En particulier, la taille d'un pointeur est 64 bits. L'accès à la mémoire d'un ordinateur est une opération coûteuse. Pour cette raison, un microprocesseur contient un petit nombre de *registres* dans lesquels peuvent être stockées quelques valeurs et sur lesquels on peut effectuer des calculs sans accéder à la mémoire. L'architecture x86-64 propose 16 registres, illustrés figure 1.1. Il s'agit de registres 64 bits, dont les noms commencent par `%r`, comme `%rax`. On peut accéder à des portions 32, 16 ou 8 bits de ces registres en utilisant d'autres noms, comme illustré dans

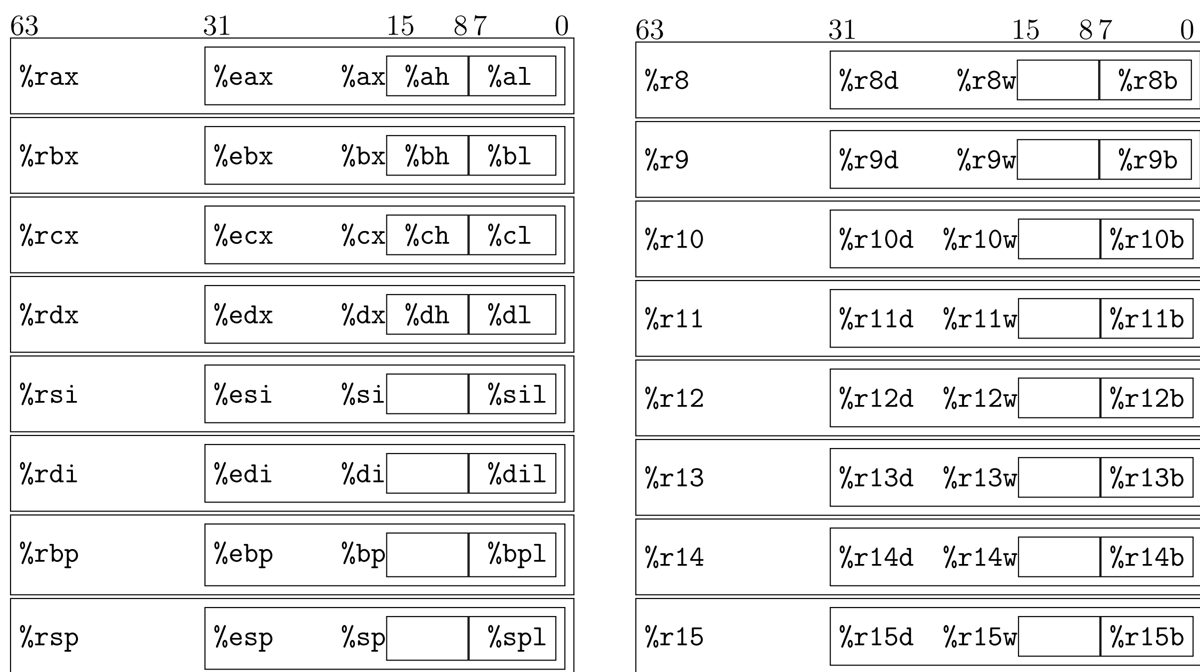


FIGURE 1.1 – Les 16 registres x86-64.

la figure. Ainsi, le registre `%esi` représente les 32 bits de poids faible du registre `%rsi` et le registre `%al` représente l'octet du poids faible du registre `%rax`.

Assembleur x86-64. On ne programme pas un ordinateur directement en langage machine, en général, mais en utilisant un langage appelé *assembleur*. L'assembleur fournit un certain nombre de facilités, notamment la possibilité d'utiliser des étiquettes symboliques plutôt que des adresses absolues ou encore la possibilité d'allouer des données globales. Le langage assembleur est transformé en langage machine par un programme qu'on appelle également *assembleur*[2]. On note qu'il s'agit donc d'un compilateur.

On utilise ici l'assembleur GNU, avec la syntaxe dite AT&T, dans un environnement Linux. Sous d'autres systèmes, les outils peuvent être différents. En particulier, l'assembleur peut utiliser une syntaxe différente, dite Intel. L'ordre des opérandes y est notamment inversé, ce qui peut être très perturbant.

Comme avec tout langage, écrivons en premier lieu un programme qui écrit la chaîne `"hello, world"` sur la sortie standard, suivie d'un retour chariot. On écrit notre programme dans un fichier portant le suffixe `.s`, par exemple `hello.s`. Le texte du programme est donné figure 1.2. Ce programme contient des instructions de plusieurs natures. On trouve d'une part des déclarations qui commencent par un point, comme `.text`, `.globl`, `.data` et `.string`. On trouve également des étiquettes, suivies par un deux-points, à savoir `main` et `message`. Enfin, on trouve des instructions de l'assembleur x86-64, à savoir `movq`, `call` et `ret`.

La directive `.text` indique que ce qui suit est un programme, c'est-à-dire une suite d'instructions. La directive `.globl main` rend le symbole `main` visible, afin que l'on puisse construire un exécutable qui démarrera à cet endroit-là. Vient ensuite le programme proprement dit, qui démarre à l'étiquette `main`. On ignore pour l'instant les deux premières

2. En anglais, on utilise deux mots différents, à savoir *assembly language* pour le langage et *assembler* pour l'outil.

```
        .text
        .globl  main
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    $message, %rdi
        call    puts
        movq    $0, %rax
        popq    %rbp
        ret
        .data
message:
        .string "hello, world"
```

FIGURE 1.2 – Notre premier programme assembleur.

instructions, dont on expliquera le sens dans la section suivante. On souhaite appeler la fonction de bibliothèque `puts` qui affiche sur la sortie standard une chaîne de caractères suivie d'un retour chariot. Pour cela, on doit passer son argument, à savoir la chaîne de caractères, dans le registre `%rdi`. On place donc l'adresse où est stockée la chaîne de caractères, à savoir ici l'adresse correspondant à l'étiquette `message`, dans le registre `%rdi` avec l'instruction `movq $message, %rdi`. Dans la syntaxe assembleur AT&T, l'opérande de destination est située à droite. Le signe dollar devant l'étiquette `message` signifie que c'est la valeur (l'adresse) que représente cette étiquette qui nous intéresse ici et non la valeur située en mémoire à cet emplacement. La deuxième instruction, `call puts`, appelle la fonction `puts`. Une fois l'appel effectué, on souhaite terminer notre programme. Pour cela, on place la valeur de retour de notre programme dans le registre `%rax`, avec l'instruction `movq $0, %rax`. Là encore, le signe dollar signifie que c'est la valeur 0 que l'on souhaite mettre dans le registre `%rax` et non pas la valeur située en mémoire à l'adresse 0. Enfin, on exécute l'instruction `ret` pour terminer notre programme. Le fait qu'il faille placer l'argument de `puts` dans le registre `%rdi` ou encore la valeur de retour dans le registre `%rax` fait partie de conventions que nous expliquerons plus loin.

Le reste du programme consiste à allouer la chaîne de caractères `"hello, world"`. Pour cela, on commence par la directive `.data` qui signifie que ce qui suit sont des données et non plus des instructions. Vient ensuite l'étiquette `message` qui représente l'adresse à laquelle la chaîne est stockée. Enfin, on déclare la chaîne proprement dite avec la directive `.string` qui stocke en mémoire la chaîne qu'elle reçoit en argument en lui ajoutant un caractère terminal de code 0. C'est ce dernier caractère qui permet à `puts` de détecter la fin de la chaîne, selon la convention en C. Ceci termine notre premier programme assembleur.

On assemble ce programme en appelant la commande `as`, l'assembleur GNU du Linux, en lui demandant de produire le code assemblé dans le fichier `hello.o` :

```
> as hello.s -o hello.o
```

On appelle ensuite l'éditeur de liens `ld` pour construire un exécutable à partir du fichier `hello.o`. Comme le nom l'indique, c'est là qu'on fait le lien entre la référence à la fonction `puts` dans notre programme et sa définition dans la bibliothèque C de GNU. Invoquer `ld` directement est assez complexe et on peut se reposer sur le compilateur C `gcc` pour le

faire à notre place[3] :

```
> gcc hello.o -o hello
```

Ici le compilateur C ne fait pas d'autre travail que d'appeler l'éditeur de liens, car il n'y a pas de fichiers à compiler sur sa ligne de commande, seulement un fichier déjà assemblé. On peut enfin exécuter notre programme, avec le résultat escompté :

```
./hello
> hello, world
```

Plus simplement encore, on aurait pu passer directement notre fichier `hello.s` à `gcc`, qui aurait appelé successivement `as` pour l'assembler et `ld` pour l'édition de liens, avec le même résultat au final.

Si on est curieux, on peut examiner le résultat de l'assemblage de notre programme. Pour cela, on utilise un *désassembleur*, c'est-à-dire un programme qui convertit le langage machine en langage assembleur. Sous Linux, la commande `objdump -d` a cet effet. On peut commencer par observer le contenu du fichier `hello.o` produit par `as`.

```
> objdump -d hello.o
0000000000000000 <main>:
   0: 55                     push   %rbp
   1: 48 89 e5               mov    %rsp,%rbp
   4: 48 c7 c7 00 00 00 00   mov    $0x0,%rdi
   b: e8 00 00 00 00         call   10 <main+0x10>
  10: 48 c7 c0 00 00 00 00   mov    $0x0,%rax
  17: 5d                     pop    %rbp
  18: c3                     ret
```

On note deux choses en particulier. D'une part, les adresses de la chaîne et de la fonction `puts` ont la valeur nulle. D'autre part, le programme commence à l'adresse 0. La raison est qu'on n'a pas encore réalisé l'édition de liens et que ces différentes adresses ne sont pas encore connues. Si en revanche on désassemble maintenant l'exécutable, on observe la différence apportée par l'édition de liens.

```
> objdump -d hello
0000000000401126 <main>:
  401126: 55                     push   %rbp
  401127: 48 89 e5               mov    %rsp,%rbp
  40112a: 48 c7 c7 30 40 40 00   mov    $0x404030,%rdi
  401131: e8 fa fe ff ff         call   401030 <puts@plt>
  401136: 48 c7 c0 00 00 00 00   mov    $0x0,%rax
  40113d: 5d                     pop    %rbp
  40113e: c3                     ret
```

D'une part, le programme commence maintenant à l'adresse `0x401126`. D'autre part, la chaîne est stockée à l'adresse `0x404030` et la fonction `puts` à l'adresse `0x401030`. On note par ailleurs que les octets de l'entier `0x404030`, sont rangés en mémoire dans l'ordre `30`, `40`, `40`, `00`, c'est-à-dire en commençant par les octets de poids faible. On dit que la machine est *petit-boutiste* (en anglais *little-endian*). D'autres architectures sont au contraire *gros-boutistes* (en anglais *big-endian*), c'est-à-dire qu'elles rangent les octets en mémoire

3. En passant l'option `-v` à `gcc`, on peut observer comment `gcc` appelle l'éditeur de liens.

en commençant par les octets de poids fort [4]. En première approximation, le programmeur peut ignorer ce détail, tant qu'il manipule les entiers dans leur intégralité par l'intermédiaire des opérations fournies par la machine et par les différents langages de plus haut niveau, y compris l'assembleur. En revanche, dès lors que l'on accède à une portion seulement d'un entier en mémoire, on doit être conscient du boutisme de la machine.

Jeu d'instructions x86-64. Le jeu d'instructions x86-64 contient littéralement des milliers d'instructions, avec pour chacune de nombreuses combinaisons possibles d'opérandes, et il n'est pas question de les décrire toutes ici. On trouve en ligne des documentations exhaustives de l'architecture x86-64 [5]. On se contente de décrire ici les instructions les plus courantes et en particulier celles que nous utiliserons par la suite.

Les opérandes d'une instruction peuvent être de plusieurs natures. Il peut s'agir d'une *opérande immédiate* de la forme `$`n. L'entier n est alors limité à 32 bits. Une opérande peut également être un registre. Enfin, une opérande peut désigner un emplacement mémoire, soit sous la forme d'une adresse immédiate (un entier 32 bits), soit sous la forme d'un *adressage indirect* relatif à un registre r. Dans ce dernier cas, l'opérande s'écrit `(`r`)` et signifie l'emplacement mémoire désigné par l'adresse contenue dans le registre r. De manière plus générale, une opérande indirecte prend la forme n`(`r_1`,`r_2`,`m`)` où n est une constante, r_1 et r_2 des registres et m un entier valant 1, 2, 4 ou 8. Une telle opérande désigne l'emplacement mémoire à l'adresse $n + r_1 + m \times r_2$.

Transfert de données. On a déjà croisé l'instruction `mov`, pour charger une constante dans un registre. De manière plus générale, l'instruction `mov` op_1`,` op_2 effectue une copie de l'opérande op_1 vers l'opérande op_2. Lorsque les opérandes ne permettent pas de déterminer la taille de la donnée qui doit être copiée, on peut le préciser avec `movb` (un octet), `movw` (deux octets, ou *mot*), `movl` (quatre octets, ou *mot long*) ou `movq` (huit octets, ou *quadruple mot*). Ainsi, l'instruction `movq $42, (%rdi)` écrit l'entier 42 sur 64 bits à l'adresse donnée par `%rdi`. Le suffixe `q` est ici nécessaire pour préciser qu'il s'agit d'un entier 64 bits. À noter que toutes les combinaisons d'opérandes ne sont pas possibles. En particulier, on ne peut pas faire deux accès à la mémoire dans une même instruction. On ne pourra donc pas écrire `mov (%rax), (%rcx)` et il faudra utiliser deux instructions `mov` successivement, par exemple `mov (%rax), %rax` suivie de `mov %rax, (%rcx)`.

Lorsqu'on copie une valeur d'une opérande plus petite vers une opérande plus grande, il faut préciser si on souhaite faire une extension de signe (`movs`) ou mettre des zéros dans les bits de poids fort (`movz`). Il faut aussi préciser les tailles des deux opérandes. Ainsi, l'instruction `movsbq %al, %rdi` copie l'entier 8 bits signé contenu dans `%al` dans les 64 bits du registre `%rdi`. Enfin, il existe une instruction particulière `movabsq` pour charger une constante 64 bits dans un registre.

Opérations arithmétiques et logiques. Sans surprise, la machine fournit des opérations arithmétiques, à savoir l'addition (`add`), la soustraction (`sub`), la multiplication (`imul`) et la division (`idiv`). Ces opérations prennent seulement deux opérandes, en affectant à la seconde le résultat du calcul. Ainsi, `add $2, %rax` réalise l'opération `%rax` ←

4. Le vocabulaire *little/big-endian* en anglais et petit/gros-boutiste en français fait référence aux voyages de Gulliver de Jonathan Swift.

5. par exemple sur `https://developer.amd.com/resources/developer-guides-manuals/`.

`%rax` $+ 2$ et `sub %rdi, %rsi` l'opération `%rsi` $\leftarrow$ `%rsi` $-$ `%rdi`. C'est pourquoi on parle de *code à deux adresses*, par opposition à d'autres architectures où les opérations arithmétiques prennent trois opérandes — on parle alors de code *à trois adresses*.

Comme pour `mov`, beaucoup de combinaisons d'opérandes sont possibles, tant qu'on n'accède pas plus d'une fois à la mémoire. La multiplication exige par ailleurs que son opérande de destination soit un registre. La division est un peu particulière, dans le sens où elle prend une unique opérande et divise l'entier contenu dans l'ensemble des deux registres `%rdx` et `%rax` par cette opérande, puis met le quotient dans `%rax` et le reste dans `%rdx`. Pour cette raison, une instruction particulière (`cqto`) permet de copier dans l'ensemble `%rdx`-`%rax` la valeur (signée) contenue dans `%rax`, c'est-à-dire qu'elle réplique le bit de signe de `%rax` partout dans `%rdx`. Il existe également des opérations arithmétiques unaires pour incrémenter (`inc`), décrémenter (`dec`) et prendre la négation (`neg`).

Une opération arithmétique particulière, `lea`, calcule l'adresse effective d'une opérande indirecte $n(r_1, r_2, m)$, c'est-à-dire la valeur $n + r_1 + m \times r_2$, sans pour autant accéder à la mémoire. Cette valeur est stockée dans la seconde opérande. Ainsi, l'instruction `leaq -1(%rdi), %rsi` affecte au registre `%rsi` la valeur `%rdi` $- 1$. On peut avantageusement utiliser l'instruction `lea` pour effectuer des calculs arithmétiques.

D'autres opérations, dites *logiques*, permettent de manipuler les bits sans interprétation arithmétique particulière. Ainsi, on peut effectuer un *et logique* (`and`), un *ou logique* (`or`) ou encore un *ou exclusif* (`xor`) entre deux opérandes. De même, on peut prendre la *négation logique* (`not`). On peut aussi décaler les bits d'un entier, vers la gauche (`sal`) en introduisant des bits 0 à droite, vers la droite (`shr`) en introduisant des bits 0 à gauche ou vers la droite (`sar`) en répliquant le bit de signe. Les opérations de décalage à gauche et à droite peuvent être interprétées comme des opérations arithmétiques, respectivement de multiplication et de division par une puissance de deux. Ainsi, `sarq $2, %rdi` peut être vu comme la division de `%rdi` par quatre.

Exercice 1. Que fait l'instruction `xorq %rax, %rax` ? Solution □

Exercice 2. Que fait l'instruction `andq $-16, %rsp` ? Solution □

Opérations de branchement. Les instructions arithmétiques et logiques, à l'exception de `lea`, positionnent des *drapeaux* booléens à l'issue de leur calcul.

Parmi ces drapeaux, on trouve notamment ZF (le résultat vaut 0), CF (le résultat a provoqué une retenue au delà du bit de poids fort), SF (le résultat est négatif en arithmétique signée) et OF (le résultat a provoqué un débordement en arithmétique signée). La subtile différence entre CF et OF tient à l'interprétation du résultat comme étant un entier signé ou non. Ces drapeaux peuvent être consultés par d'autres instructions, afin de prendre des décisions conditionnelles. Ainsi, l'instruction `jz L` poursuivra l'exécution du programme à l'instruction située à l'étiquette `L` si le drapeau ZF indique un résultat nul. Sinon, l'exécution se poursuivra avec l'instruction suivante. On appelle cela un *branchement conditionnel*. On trouve toutes les variantes dans le tableau ci-contre. On peut également calculer une valeur 0 ou 1 en fonction de la position des

saut	signification
`jz`	$= 0$
`jnz`	$\neq 0$
`js`	< 0
`jns`	≥ 0
`jg`	$>$ signé
`jge`	$\geq$ signé
`jl`	$<$ signé
`jle`	$\leq$ signé
`ja`	$>$ non signé
`jae`	$\geq$ non signé
`jb`	$<$ non signé
`jbe`	$\leq$ non signé

drapeaux. Cette valeur est calculée dans une opérande 8 bits avec les instructions `setz`, `setnz`, `sets`, etc. On peut aussi effectuer une opération `mov` conditionnellement selon la valeur des drapeaux, avec les instructions `cmovz`, `cmovnz`, `cmovs`, etc.

Deux instructions permettent de positionner les drapeaux sans pour autant modifier de registres. Il s'agit de l'instruction `cmp` qui affecte les drapeaux comme l'aurait fait l'instruction `sub` et l'instruction `test` qui affecte les drapeaux comme l'aurait fait l'instruction `and`. Ainsi, si on souhaite obtenir le résultat du test `%rdi` ≤ 100 dans `%al`, on peut utiliser `cmpq $100, %rdi` suivie de `setle %al`. Il faut prêter attention ici au sens de la soustraction, qui peut être perturbant.

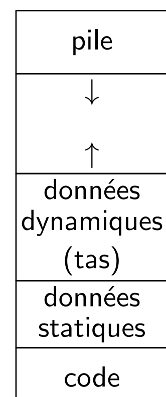
Branchement inconditionnel. Enfin, mentionnons qu'on peut effectuer un saut inconditionnel avec l'instruction `jmp`. L'opérande peut être une étiquette, comme pour un saut conditionnel, mais aussi une adresse calculée, avec la syntaxe `call *`*op*. Ainsi, `jmp *%rax` saute à l'adresse contenue dans le registre `%rax`. Une telle possibilité de transférer le contrôle à une adresse calculée est cruciale pour pouvoir compiler des langages fonctionnels ou orientés objets (voir les chapitres 7 et 8).

1.3 Le défi de la compilation

Le défi de la compilation, c'est de traduire un programme d'un langage de haut niveau vers le jeu d'instructions que nous venons de présenter. En particulier, il faut traduire un grand nombre de concepts qui ne sont pas directement présents dans l'assembleur, tels que les structures de contrôle (tests, boucles, exceptions, etc.), les appels de fonctions, les structures de données complexes (tableaux, enregistrements, objets, clôtures, etc.), ou encore l'allocation dynamique de mémoire.

Commençons par l'appel de fonction. Un premier constat est que les appels de fonctions peuvent être arbitrairement imbriqués et que les seize registres peuvent ne pas suffire pour toutes les variables manipulées par ces fonctions. Il va donc falloir allouer de la mémoire pour cela. Fort heureusement, dans la plupart des langages de programmation, les fonctions procèdent selon une logique bien parenthésée, c'est-à-dire que si une fonction f appelle une fonction g, alors l'appel à g terminera avant que l'appel à f ne termine. Cette propriété a pour conséquence que l'on peut utiliser une *pile* pour organiser l'information relative aux appels de fonctions.

Chaque programme en cours d'exécution possède une telle pile, qu'on nomme *pile d'appels* (en anglais *call stack*). Cette pile est matérialisée tout en haut de la mémoire et croît dans le sens des adresses décroissantes. À tout instant, le registre `%rsp` pointe sur le sommet de la pile[6]. Le reste de la mémoire disponible pour le programme est appelé le *tas*. On y a alloue les données qui doivent survivre aux appels de fonctions. Dans le système Linux, le code du programme se situe tout en bas de la mémoire, les données allouées statiquement (le segment de données) juste au-dessus et vient ensuite le tas. Ainsi, on ne se marche pas sur les pieds. Cette organisation est rendue possible parce que chaque programme a l'illusion d'avoir toute la mémoire pour lui tout seul. C'est le système d'exploitation qui crée cette illusion, en utilisant un mécanisme fourni par le matériel appelé MMU.



6. Le nom `%rsp` signifie *stack pointer*.

Pour manipuler la pile, on dispose d'instruction `push` et `pop` qui permettent respectivement d'empiler et de dépiler des valeurs. Ainsi, l'instruction `pushq $42` empile 64 bits représentant l'entier 42 et l'instruction `popq %rdi` dépile 64 bits et les écrit dans le registre `%rdi`. Ces deux instructions mettent à jour la valeur du pointeur de pile `%rsp`. Bien entendu, on peut manipuler la pile explicitement. Par exemple, `addq $48, %rsp` dépile d'un coup 48 octets.

Expliquons maintenant comment est compilé un appel de fonction. Lorsqu'une fonction `f`, qu'on appelle *l'appelant* (en anglais *caller*), souhaite appeler une fonction `g`, qu'on appelle *l'appelé* (en anglais *callee*), on ne peut pas se contenter d'exécuter l'instruction `jmp g` car il faudra revenir dans le code de `f` quand `g` aura terminé. La solution consiste à se servir de la pile. Deux instructions sont là pour cela. D'une part, l'instruction

```
call    g
```

empile l'adresse de l'instruction située juste après le `call` et transfert le contrôle à l'adresse `g`. D'autre part, l'instruction

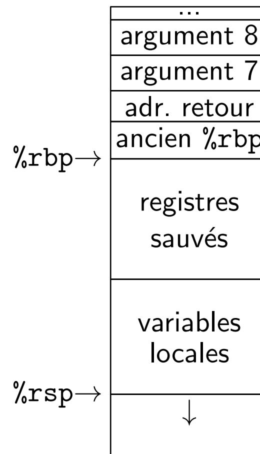
```
ret
```

dépile une adresse et y transfert le contrôle. Dans notre premier programme écrit plus haut, nous avons justement utilisé l'instruction `ret` pour terminer notre programme. En réalité, il n'était pas exactement terminé. Nous n'avions fait que revenir dans un morceau de programme installé par le système, qui avait appelé notre programme. L'adresse de retour se situait au sommet de la pile et notre programme n'avait pas modifié l'état de la pile — dans le cas contraire, nous aurions eu une désagréable surprise.

Nous avons donc un mécanisme pour effectuer un appel de fonction et revenir au point d'appel une fois qu'il est terminé. Il nous reste cependant un problème de taille : tout registre modifié par l'appelé sera potentiellement perdu pour l'appelant. Il existe de multiples manières de s'en sortir, mais on s'accorde en général sur des *conventions d'appel*. Ces conventions dépendent de l'architecture et parfois même du système d'exploitation. Sur l'architecture x86-64, ces conventions sont les suivantes. Les six premiers arguments sont passés dans les registres `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`. Les autres arguments sont passés sur la pile, le cas échéant. La valeur de retour est passée dans `%rax`. Les registres `%rbx`, `%rbp`, `%r12`, `%r13`, `%14` et `%r15` sont dits *callee-saved*, c'est-à-dire que l'appelé se doit de les sauvegarder. On y met donc des données de durée de vie longue, ayant besoin de survivre aux appels. Les autres registres sont *caller-saved*, c'est-à-dire que l'appelant se doit de les sauvegarder. On y met donc typiquement des données qui n'ont pas besoin de survivre aux appels. Bien entendu, si l'appelé ne modifie pas un registre *callee-saved*, il n'aura rien à faire pour le sauvegarder ; de même pour un registre *caller-saved* qui n'est pas modifié par l'appelant. Enfin, les registres `%rsp` et `%rbp` sont réservés à la gestion de la pile.

L'appel d'une fonction `f` implique donc la construction au sommet de la pile d'un plus ou moins grand nombre de données. Au minimum, on trouvera l'adresse de retour. Mais on peut trouver également des arguments (s'il y en a plus que six), des registres *callee-saved* sauvegardés que la fonction `f` souhaite utiliser, des variables locales qui ne peuvent être allouées dans des registres, etc. La totalité de ces données forme ce qu'on appelle le *tableau d'activation* de cet appel (en anglais *stack frame*). La taille de ce tableau pouvant varier pendant l'exécution de l'appel, il est de coutume d'utiliser le registre `%rbp`

pour désigner le début du tableau d'activation [7]. Ainsi, les deux registres `%rbp` et `%rsp` encadrent le tableau d'activation situé au sommet de la pile. Comme le registre `%rbp` fait partie des registres *callee-saved*, il doit être sauvegardé, ce qui implique que son ancienne valeur fait elle-même partie du tableau d'activation. On se retrouve donc dans la situation illustrée ci-dessous.



La construction et la destruction du tableau d'activation lors d'un appel de fonction se décomposent en quatre temps, de la manière suivante :

1. juste avant l'appel, l'appelant passe les arguments dans `%rdi`, ..., `%r9`, les autres sur la pile s'il y en a plus de six ; sauvegarde les registres *caller-saved* qu'il compte utiliser après l'appel (dans son propre tableau d'activation) ; et exécute l'instruction `call`.
2. au début de l'appel, l'appelé sauvegarde `%rbp` puis le positionne, par exemple en exécutant les deux instructions `pushq %rbp` et `movq %rsp, %rbp` ; alloue son tableau d'activation, par exemple en diminuant la valeur de `%rsp` ; et sauvegarde les registres *callee-saved* dont il aura besoin.
3. à la fin de l'appel, l'appelé place le résultat dans `%rax` ; restaure les registres sauvegardés ; dépile son tableau d'activation et restaure `%rbp`, par exemple en exécutant les deux instructions [8] `movq %rbp, %rsp` et `popq %rbp` ; et enfin exécute `ret`.
4. juste après l'appel, l'appelant dépile les éventuels arguments qu'il avait mis sur la pile ; et restaure les registres *caller-saved* qu'il avait sauvegardés.

Enfin, les conventions d'appel stipulent que la pile doit être *alignée* au moment d'un appel, c'est-à-dire que `%rsp` doit être un multiple de 16 au moment d'exécuter une instruction `call` (et vaudra donc 8 modulo 16 juste après l'appel, car l'adresse de retour aura été déposée sur la pile par `call`). Cet alignement de la pile est notamment requis par certaines fonctions qui vont chercher à sauvegarder des registres vectoriels sur la pile. En particulier, des fonctions de bibliothèque comme `scanf` ou `printf` peuvent planter si l'alignement n'est pas respecté. Aligner la pile peut être fait explicitement

7. Le nom `%rbp` signifie *base pointer*.
8. L'instruction `leave` est équivalente à ces deux instructions.

```
f:   subq $8, %rsp  # aligner la pile
     ...
     ... # car on fait des appels à des fonctions externes
     ...
     addq $8, %rsp
     ret
```

ou être obtenu gratuitement car on sauvegarde un registre sur la pile

```
f:   pushq %rbp  # on sauvegarde %rbp
     ...
     ...
     popq  %rbp  # et on le restaure
     ret
```

Il est important de comprendre que les conventions d'appel ne sont que des conventions. En particulier, on est libre de ne pas les respecter tant qu'on reste dans le périmètre de notre propre code. Si on se lie à du code externe, en revanche, on se doit de respecter les conventions d'appel.

Note. En français, on dit qu'une fonction *renvoie* une valeur. Malheureusement, on entend très souvent dire qu'une fonction « retourne une valeur ». C'est là un anglicisme, car on dit en effet en anglais *to return a value*. Mais c'est surtout un contresens tragique quand on dit par exemple « écrire une fonction qui retourne la liste des entiers de 1 à n », car on ne sait plus s'il s'agit de renverser la liste ou seulement de la renvoyer. On peut conserver néanmoins le verbe « retourner » lorsqu'il s'agit de dire qu'une fonction retourne à l'appelant, au sens de la terminaison de son calcul [9].

1.4 Un exemple de compilation

Afin de bien comprendre l'enjeu de la compilation, rien ne vaut l'exercice consistant à compiler un programme à la main. On considère le programme C donné figure 1.3. Il s'agit d'un programme extrêmement astucieux qui calcule le nombre de solutions du problème dit des n reines, c'est-à-dire le nombre de façons différentes de placer n reines sur un échiquier $n \times n$ sans qu'aucune ne soit en prise avec une autre. Ce programme est particulièrement intéressant du point de vue de la compilation, car il contient un test, une boucle, une fonction récursive et des calculs variés, tout en restant de taille raisonnable. Accessoirement, c'est aussi un programme digne d'intérêt car c'est une excellente solution au problème des n reines.

Exercice 3. Expliquer le fonctionnement de ce programme. Indications : le programme cherche à remplir l'échiquier ligne par ligne ; les entiers `a`, `b`, `c`, `d` et `e` doivent être vus plutôt comme des tableaux de booléens. Solution □

Commençons par la compilation de la fonction récursive `t`. Il faut allouer des registres pour cette fonction. Les trois arguments `a`, `b` et `c` sont passés dans les registres `%rdi`, `%rsi` et `%rdx`. Comme le résultat sera renvoyé dans `%rax`, on choisit d'allouer la variable

9. Cette subtilité dans la traduction de l'anglais *return* est expliquée dans un ouvrage de l'Association Française de Normalisation (AFNOR) datant de 1975 [7].

```
int t(int a, int b, int c) {
  int f = 1;
  if (a) {
    int d, e = a & ~b & ~c;
    f = 0;
    while (d = e & -e) {
      f += t(a-d, (b+d)*2, (c+d)/2);
      e -= d;
    }
  }
  return f;
}
int main() {
  int n;
  scanf("%d", &n);
  printf("q(%d) = %d\n", n, t(~(~0<<n), 0, 0));
}
```

FIGURE 1.3 – Un programme C à compiler.

locale `f` dans ce registre pour faciliter la compilation de l'instruction `return f`. Pour les deux autres variables locales, à savoir `d` et `e`, on choisit de les allouer respectivement dans les registres `%r8` et `%rcx`. En cas d'appel récursif, les valeurs des six variables `a`, `b`, `c`, `d`, `e` et `f` auront besoin d'être sauvegardées, car elles sont toutes utilisées après l'appel. Comme il s'agit de six registres *caller-saved*, c'est la responsabilité de l'appelant de les sauvegarder. Dans le cas d'un appel récursif, comme ici, l'appelant et l'appelé sont la même fonction et cela ne fait donc pas de différence qu'on choisisse des registres *caller-save* sauvegardés par l'appelant ou des registres *callee-saved* sauvegardés par l'appelé. Si on se dispense de l'usage de `%rbp`, inutile ici, le tableau d'activation de la fonction `t` prend la forme suivante :

	⋮
	adr. retour
	`%rax` (`f`)
	`%rcx` (`e`)
	`%r8` (`d`)
	`%rdx` (`c`)
	`%rsi` (`b`)
`%rsp` →	`%rdi` (`a`)

Le code assembleur de la fonction `t` est donné figure 1.4, dans la colonne de gauche. On commence par affecter la valeur 1 à `f` (ligne 4) puis on teste la valeur de `a` (lignes 5–6). Si `a` est nul, on saute à la fin de la fonction (étiquette `t_return`, ligne 43). Sinon, on alloue 48 octets pour le tableau d'activation (ligne 7). Puis on affecte 0 à la variable `f` (ligne 8) et la valeur de l'expression `a & ~b & ~c` à la variable `e` (lignes 9–15). Vient ensuite la boucle `while`. Pour n'effectuer qu'un seul branchement par tour de boucle, on peut placer le corps de la boucle en premier (lignes 17–35) puis le test de la boucle (lignes 36–42).

Par conséquent, on commence par un saut inconditionnel vers le test (ligne 16). Le corps de la boucle commence par la sauvegarde des six variables dans le tableau d'activation (lignes 18–23). Puis on prépare les trois arguments de l'appel récursif (lignes 24–28) avant d'effectuer celui-ci (ligne 29). Une fois l'appel effectué, on restaure les variables, tout en effectuant les mises à jour des variables `e` et `f` (lignes 30–35). Enfin, le test de la boucle consiste à affecter à `d` la valeur de l'expression `e & -e` (lignes 37–40) puis à tester si le résultat est non nul (ligne 41). On utilise ici le fait que la dernière opération arithmétique effectuée (`andq` ligne 40) a positionné les drapeaux avec la valeur de `d`. Lorsqu'on sort de la boucle, la ligne 42 désalloue le tableau d'activation.

Passons maintenant au code assembleur de la fonction `main`, donné figure 1.4 dans la colonne de droite. On commence par sauvegarder `%rbp` et positionner, ce qui a notamment pour effet d'aligner la pile. Puis on prépare l'appel à `scanf`. Les deux arguments sont mis dans `%rdi` et `%rsi` respectivement. Noter que le second argument est l'*adresse* de la variable `n`, et non sa valeur, et c'est pourquoi on doit écrire `$n` et non pas `n` (ligne 49). La fonction `scanf` étant une fonction variadique, on doit indiquer son nombre d'arguments en virgule flottante dans `%rax`. Comme il n'y en a pas ici, on met `%rax` à zéro (ligne 50). Puis on prépare les arguments de l'appel à la fonction `t`. En particulier, on calcule la valeur de `~(~0 << n)` dans `%rdi` (lignes 53–57). Lorsqu'il n'est pas constant, l'instruction `salq` exige la valeur du décalage dans le registre `%cl` et c'est pourquoi on a dû copier la valeur de `n` dans `%rcx`. Après l'appel à `t` (ligne 60), il ne reste plus qu'à effectuer l'appel à `printf` (lignes 62–66). Comme pour `scanf`, on doit mettre `%rax` à zéro (ligne 65). On termine notre programme avec la valeur de retour (lignes 67–69). Au delà du code, on trouve les données allouées statiquement, à savoir la variable `n` et les deux chaînes de format passées respectivement à `scanf` et `printf`. La directive `.quad 0` réserve huit octets initialisés avec la valeur 0. La directive `.string` alloue une chaîne terminée par un caractère de code ASCII 0, selon la convention du langage C et en particulier des fonctions `scanf` et `printf` qu'on utilise ici.

Exercice 4. Écrire en assembleur la fonction suivante qui calcule la partie entière de la racine carrée de n :

```
isqrt(n) ≡
  c ← 0
  s ← 1
  while s ≤ n
    c ← c + 1
    s ← s + 2c + 1
  return c
```

Afficher la valeur de `isqrt`(17). Solution □

Exercice 5. Écrire en assembleur la fonction factorielle, d'abord avec une boucle, puis avec une fonction récursive. Solution □

Notes bibliographiques. Le livre *Computer Systems* de Bryant et O'Hallaron [4] est une excellente source d'information concernant l'architecture des ordinateurs, notamment x86-64, du point de vue du programmeur en général et de celui qui souhaite écrire un compilateur en particulier.

```
1          .text
2          .globl main
3   t:
4          movq    $1, %rax
5          testq   %rdi, %rdi
6          jz      t_return
7          subq    $48, %rsp
8          xorq    %rax, %rax
9          movq    %rdi, %rcx
10         movq    %rsi, %r9
11         notq    %r9
12         andq    %r9, %rcx
13         movq    %rdx, %r9
14         notq    %r9
15         andq    %r9, %rcx
16         jmp     loop_test
17  loop_body:
18         movq    %rdi,  0(%rsp)
19         movq    %rsi,  8(%rsp)
20         movq    %rdx, 16(%rsp)
21         movq    %r8,  24(%rsp)
22         movq    %rcx, 32(%rsp)
23         movq    %rax, 40(%rsp)
24         subq    %r8, %rdi
25         addq    %r8, %rsi
26         salq    $1, %rsi
27         addq    %r8, %rdx
28         shrq    $1, %rdx
29         call    t
30         addq    40(%rsp), %rax
31         movq    32(%rsp), %rcx
32         subq    24(%rsp), %rcx
33         movq    16(%rsp), %rdx
34         movq     8(%rsp), %rsi
35         movq     0(%rsp), %rdi
36  loop_test:
37         movq    %rcx, %r8
38         movq    %rcx, %r9
39         negq    %r9
40         andq    %r9, %r8
41         jnz     loop_body
42         addq    $48, %rsp
43  t_return:
44         ret
```

```
45  main:
46         pushq   %rbp
47         movq    %rsp, %rbp
48         movq    $input, %rdi
49         movq    $n, %rsi
50         xorq    %rax, %rax
51         call    scanf
52
53         xorq    %rdi, %rdi
54         notq    %rdi
55         movq    (n), %rcx
56         salq    %cl, %rdi
57         notq    %rdi
58         xorq    %rsi, %rsi
59         xorq    %rdx, %rdx
60         call    t
61
62         movq    $msg, %rdi
63         movq    (n), %rsi
64         movq    %rax, %rdx
65         xorq    %rax, %rax
66         call    printf
67         xorq    %rax, %rax
68         popq    %rbp
69         ret
70
71         .data
72  n:
73         .quad   0
74  input:
75         .string "%d"
76  msg:
77         .string "q(%d) = %d\n"
```

Figure 1.4 – Le résultat de la compilation du programme de la figure 1.3.

2 Qu'est-ce qu'un langage de programmation

Comment définir la signification des programmes écrits dans un langage de programmation ? La plupart du temps, on se contente d'une description informelle, en langue naturelle : norme ISO, standard, ouvrage de référence, documentation en ligne, etc. Une telle définition se révèle souvent incomplète, imprécise, voire ambiguë.

Le langage Java, par exemple, est défini dans l'ouvrage *The Java Language Specification* [11]. Bien que très précis, cet ouvrage n'en contient pas moins quelques ambiguïtés. Ainsi on trouve la spécification suivante concernant l'ordre d'évaluation des opérandes :

> The Java programming language guarantees that the operands of operators appear to be evaluated in a specific *evaluation order*, namely, from left to right.

Mais elle est immédiatement suivie d'un commentaire qui ni peut que jeter le lecteur dans la perplexité :

> It is recommended that code not rely crucially on this specification.

Loin de nous cependant l'idée de blâmer les auteurs de cette définition de Java. Beaucoup de langages n'offrent pas une définition aussi détaillée et se contentent souvent d'offrir leur compilateur comme seule référence. Les programmeurs doivent alors s'engager dans une découverte expérimentale de la sémantique ou une lecture attentive du code du compilateur. Lorsqu'ils n'ont pas ce courage, ils s'en remettent à d'autres pour faire ce travail. Les sites comme `stackoverflow.com` contiennent d'innombrables questions concernant la sémantique des langages de programmation. L'archivage de toutes ces réponses devient *de facto* une référence pour bon nombre de programmeurs.

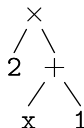
Il y a cependant des situations où l'on ne peut se contenter d'une définition informelle. C'est le cas notamment lorsqu'on souhaite *raisonner* sur les programmes, par exemple pour prouver la correction d'un système de types ou d'un compilateur. La *sémantique formelle* est un outil mathématique qui permet de donner un sens précis à un programme, en décrivant les calculs qu'il effectue.

Avant de chercher à définir une sémantique formelle, il convient de s'entendre sur ce qu'est un programme. En tant qu'objet syntaxique, c'est-à-dire la suite de caractères qui compose le texte source du programme, il est trop difficile à manipuler. En particulier, la présence de blancs, de commentaires ou encore de parenthèses superflues viendrait

compliquer le discours mathématique. À cette notion de syntaxe concrète on préfère la notion de *syntaxe abstraite*.

2.1 Syntaxe abstraite

La syntaxe abstraite est une représentation d'un programme sous la forme d'un type de données arborescent. Ainsi une expression de programme qui représente la multiplication de la constante 2 par une autre expression qui représente l'addition de la variable `x` et de la constante 1 est matérialisée par l'*arbre de syntaxe abstraite* suivant :



Dans la syntaxe concrète d'un certain langage de programmation, une telle expression pourrait s'écrire

```
2*(x+1)
```

mais aussi avec plus de parenthèses, ici inutiles

```
(2 * ((x) + 1))
```

ou bien encore avec un commentaire, ici également inutile

```
2 * (* je multiplie par deux *) ( x + 1 )
```

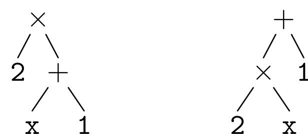
Ces trois expressions de programme, et bien d'autres encore, représentent toutes le même arbre de syntaxe abstraite donné plus haut.

Définissons formellement ce qu'est un arbre de syntaxe abstraite pour de telles expressions arithmétiques. En nous limitant aux quatre constructions utilisées ci-dessus, une expression e est donc soit une constante, soit une variable, soit l'addition de deux expressions, soit la multiplication de deux expressions. On l'exprime formellement à l'aide d'une *grammaire*, de la manière suivante :

$$
\begin{array}{rcll}
e & ::= & c & \textit{constante} \\
  & | & x & \textit{variable} \\
  & | & e + e & \textit{addition} \\
  & | & e \times e & \textit{multiplication}
\end{array}
$$

On peut voir cela comme la définition d'un type d'arbres dont les feuilles sont des constantes ou des variables et dont les nœuds internes sont des additions ou des multiplications. Ici c représente une constante quelconque (`1`, `42`, etc.) et x représente une variable quelconque (`a`, `foo`, etc). En particulier, on ne doit pas confondre la méta-variable x, qui représente n'importe quelle variable, et la variable de nom `x`.

Le lecteur attentif a déjà remarqué que la syntaxe abstraite ne représente pas explicitement les parenthèses parmi ses constructions. Les parenthèses sont utiles dans la syntaxe concrète, par exemple pour distinguer les deux expressions `2*(x+1)` et `2*x+1` si on suppose la multiplication prioritaire sur l'addition comme de coutume. Dans la syntaxe abstraite, ces deux expressions sont représentées par deux arbres différents, à savoir



Il n'y a pas lieu de conserver les parenthèses de la première expression dans l'arbre de syntaxe abstraite. La forme de ces deux arbres se suffit à elle-même. On verra dans le chapitre 4 comment l'analyse syntaxique traduit la syntaxe concrète vers la syntaxe abstraite et notamment comment la présence de parenthèses influe sur la construction des arbres de syntaxe abstraite.

Dans la grammaire ci-dessus, on a réutilisé pour l'addition et la multiplication des notations infixes qui font écho à celles de la syntaxe concrète. C'est simplement par commodité. On aurait pu tout aussi bien choisir des symboles différents, comme $\oplus$ ou *Add*. C'est également par commodité que l'on ne va pas systématiquement dessiner les arbres de syntaxe abstraite mais utiliser une écriture en ligne où les parenthèses sont utilisées uniquement pour montrer la structure. Ainsi on s'autorisera d'écrire $2 \times (\mathtt{x} + 1)$, tout en ayant conscience qu'il s'agit là de syntaxe abstraite et non de syntaxe concrète et que les parenthèses ne sont là que pour montrer que la racine de l'arbre est la multiplication.

Il n'y a pas que les parenthèses qui constituent une différence entre la syntaxe concrète et la syntaxe abstraite. Ainsi, certaines constructions de la syntaxe concrète peuvent être exprimées directement à l'aide d'autres constructions de la syntaxe abstraite. On appelle cela le *sucre syntaxique*. Ajoutons par exemple à nos expressions arithmétiques une nouvelle construction syntaxique `succ` pour représenter l'opération +1, nous permettant d'écrire `2 * succ x` à la place de `2 * (x + 1)`. Une telle opération pourrait être ajoutée à la syntaxe abstraite comme une cinquième construction. Mais elle peut être aussi exprimée à l'aide d'opérations qui existent déjà, à savoir l'addition et la constante `1`. L'intérêt de cette seconde option est de limiter le nombre des constructions de la syntaxe abstraite. Ainsi dans le langage C la construction `a[i]` est du sucre syntaxique pour `*(a+i)` et dans le langage OCaml la construction `[a; b; c]` est du sucre syntaxique pour `a :: b :: c :: []`.

Représentation en machine. La syntaxe abstraite n'est pas que l'outil du mathématicien qui cherche à définir une sémantique formelle. C'est également la représentation des programmes en machine dans un interprète ou un compilateur. Dans un langage comme OCaml, un type algébrique est une transcription immédiate de la syntaxe abstraite :

```
type expression =
  | Cte of int
  | Var of string
  | Add of expression * expression
  | Mul of expression * expression
```

Les constructeurs `Cte` et `Var` contiennent respectivement la valeur de la constante et le nom de la variable.

Exercice 6. Écrire une fonction `succ: expression -> expression` correspondant à l'opération +1. Solution □

Exercice 7. Écrire une fonction `eval: (string -> int) -> expression -> int` qui calcule la valeur d'une expression. Le premier argument est une fonction qui donne la valeur de chaque variable. Solution □

2.2 Sémantique opérationnelle

La *sémantique opérationnelle* décrit l'enchaînement des calculs élémentaires qui mènent d'une expression de programme à sa valeur. Elle opère directement sur la syntaxe abstraite. On distingue deux formes de sémantique opérationnelle : la sémantique `naturelle`, encore appelée « à grands pas » (en anglais *big-steps semantics*) ; et la sémantique *à réductions*, encore appelée « à petits pas » (en anglais *small-steps semantics*).

Pour illustrer ces deux notions, nous faisons le choix du langage Mini-ML. Il s'agit d'un fragment de ML réduit à l'essentiel. On y trouve des fonctions de première classe, des paires et des variables locales. La syntaxe abstraite de Mini-ML est la suivante :

$$\begin{array}{rcll} e & ::= & x & \text{identificateur} \\ & | & c & \text{constante } (1,\ 2,\ \ldots,\ \mathit{true},\ \ldots) \\ & | & op & \text{primitive } (+,\ \times,\ \mathit{fst},\ \ldots) \\ & | & \texttt{fun}\ x \rightarrow e & \text{fonction anonyme} \\ & | & e\ e & \text{application} \\ & | & (e,\ e) & \text{paire} \\ & | & \texttt{let}\ x = e\ \texttt{in}\ e & \text{liaison locale} \end{array}$$

Même s'il est qualifié de « mini », ce langage n'en contient pas moins toute la complexité d'un vrai langage de programmation. En fait, la seule présence de la variable, de l'abstraction (`fun`) et de l'application suffit à capturer n'importe quel modèle de calcul [1]. Mini-ML y ajoute les constantes, les opérations primitives, les paires et les variables locales pour le rapprocher d'un langage de programmation usuel.

On reste volontairement flou sur l'ensemble des constantes et des opérations primitives. Afin de pouvoir écrire des exemples intéressants, on peut supposer que cela inclut au moins des entiers et des opérations arithmétiques usuelles. Ainsi on peut écrire le programme

$$\begin{array}{l} \texttt{let}\ \mathit{compose} = \texttt{fun}\ f \rightarrow \texttt{fun}\ g \rightarrow \texttt{fun}\ x \rightarrow f\ (g\ x)\ \texttt{in} \\ \texttt{let}\ \mathit{plus} = \texttt{fun}\ x \rightarrow \texttt{fun}\ y \rightarrow +\ (x,\ y)\ \texttt{in} \\ \mathit{compose}\ (\mathit{plus}\ 2)\ (\mathit{plus}\ 4)\ 36 \end{array} \tag{2.1}$$

ou encore le programme

$$\begin{array}{l} \texttt{let}\ \mathit{distr_pair} = \texttt{fun}\ f \rightarrow \texttt{fun}\ p \rightarrow (f\ (\mathit{fst}\ p),\ f\ (\mathit{snd}\ p))\ \texttt{in} \\ \texttt{let}\ p = \mathit{distr_pair}\ (\texttt{fun}\ x \rightarrow x)\ (40,\ 2)\ \texttt{in} \\ +\ (\mathit{fst}\ p,\ \mathit{snd}\ p) \end{array} \tag{2.2}$$

Exercice 8. Quelle est la différence entre $+$ et $\texttt{fun}\ x \rightarrow \texttt{fun}\ y \rightarrow +\ (x,\ y)$? Solution □

Le lecteur peut être surpris de ne pas trouver dans ce langage de construction conditionnelle de type `if-then-else` ou encore de fonction récursive. Si on souhaite de telles constructions dans nos programmes, on peut les ajouter comme autant de primitives particulières. Ainsi, la conditionnelle peut être matérialisée par une primitive *opif* et associée au sucre syntaxique suivant :

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \stackrel{\text{def}}{=} \mathit{opif}\ (e_1,\ ((\texttt{fun}\ _ \rightarrow e_2),\ (\texttt{fun}\ _ \rightarrow e_3)))$$

1. C'est ce qu'on appelle le λ-calcul. L'abstraction $\texttt{fun}\ x \rightarrow e$ s'y note $\lambda x.e$ et s'appelle pour cette raison une λ-abstraction.

On note que les deux branches e_2 et e_3 sont représentées par des abstractions, ceci afin de geler le calcul correspondant, dans un sens qui sera justement précisé par la sémantique que nous nous apprêtons à définir. De la même façon, la récursivité peut être introduite par une autre primitive, *opfix*, également associée à un sucre syntaxique :

$$\texttt{rec}\ f\ x = e \quad \stackrel{\text{def}}{=} \quad \mathit{opfix}\ (\texttt{fun}\ f \rightarrow \texttt{fun}\ x \rightarrow e)$$

Il peut sembler qu'introduire des constructions aussi essentielles que la conditionnelle ou la récursivité sous forme de primitives est une façon maladroite de limiter le nombre de constructions de Mini-ML. Mais comme nous l'avons dit plus haut, l'abstraction et l'application sont les constructions vraiment fondamentales, contenant toute l'essence du calcul. Nous montrerons même plus loin qu'il est possible de définir les primitives *opif* et *opfix*.

Variables libres et liées. Dans l'expression $\texttt{fun}\ x \rightarrow e$, la variable x est dite *liée* dans l'expression e et on dit que la construction `fun` est un *lieur*. Le nom de la variable importe moins que la liaison elle-même. Ainsi, on peut écrire aussi bien $\texttt{fun}\ x \rightarrow +(x, 1)$ que $\texttt{fun}\ y \rightarrow +(y, 1)$ et ces deux expressions dénotent exactement la même abstraction. On dit qu'elles sont α-équivalentes. De la même façon, la construction `let` est un lieur. Dans l'expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, la variable x est liée dans l'expression e_2. En revanche, elle n'est pas liée dans l'expression e_1. Une variable qui n'est pas liée est dite *libre*. L'ensemble des variables libres d'une expression peut être défini par récurrence.

Définition 1 (variables libres)**.** L'ensemble des *variables libres* d'une expression e, noté $\mathit{fv}(e)$, est défini par récurrence sur e de la manière suivante :

$$\begin{array}{rcl}
\mathit{fv}(x) & = & \{x\} \\
\mathit{fv}(c) & = & \emptyset \\
\mathit{fv}(op) & = & \emptyset \\
\mathit{fv}(\texttt{fun}\ x \rightarrow e) & = & \mathit{fv}(e) \setminus \{x\} \\
\mathit{fv}(e_1\ e_2) & = & \mathit{fv}(e_1) \cup \mathit{fv}(e_2) \\
\mathit{fv}((e_1,\ e_2)) & = & \mathit{fv}(e_1) \cup \mathit{fv}(e_2) \\
\mathit{fv}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & \mathit{fv}(e_1) \cup (\mathit{fv}(e_2) \setminus \{x\})
\end{array}$$

Une expression sans variable libre est dite *close*. □

La suppression de la variable x de l'ensemble calculé pour les constructions `fun` et `let` traduit très exactement le caractère lié de cette variable. En particulier, on a correctement parenthésé le dernier cas pour exprimer une liaison dans e_2 mais pas dans e_1. Ainsi, on a

$$\begin{array}{rcl}
\mathit{fv}\,(\texttt{let}\ x = +(20, 1)\ \texttt{in}\ (\texttt{fun}\ y \rightarrow +(y, y))\ x) & = & \emptyset \\
\mathit{fv}\,(\texttt{let}\ x = z\ \texttt{in}\ (\texttt{fun}\ y \rightarrow (x\ y)\ t)) & = & \{z, t\}
\end{array}$$

Si on a le loisir de renommer à volonté les variables liées, il faut cependant être attentif à d'éventuels conflit avec des variables libres. Ainsi, dans l'expression

$$\texttt{fun}\ x \rightarrow \texttt{let}\ y = +(x,\ x)\ \texttt{in}\ \times (x,\ y)$$

on ne peut pas renommer x en y dans le premier lieur, ni y en x dans le second, sans changer la nature de l'expression. Ce conflit de renommage entre une variable libre et une variable liée de même nom est connu sous le nom de *capture de variable*.

2.2.1 Sémantique à grands pas

La sémantique à grands pas cherche à définir une relation entre une expression e et une *valeur* v, notée $e \twoheadrightarrow v$, dont la signification est « l'évaluation de l'expression e termine sur la valeur v ». Il convient de définir ce qu'on entend ici par « valeur ». Il s'agit du sous-ensemble des expressions ainsi défini :

$$\begin{array}{rcll} v & ::= & c & \text{constante} \\ & | & op & \text{primitive non appliquée} \\ & | & \texttt{fun}\ x \rightarrow e & \text{fonction} \\ & | & (v, v) & \text{paire} \end{array}$$

Cette définition est inductive, car une paire n'est une valeur que s'il s'agit d'une paire de valeurs. On note qu'une abstraction est toujours une valeur. Son corps reste une expression, ce qui traduit notre intention de ne pas évaluer sous les abstractions. Ainsi, l'expression $\texttt{fun}\ x \rightarrow +(1,\ 2)$ est une valeur, même si l'expression $+(1,\ 2)$ pourrait être évaluée.

Substitution. Pour définir la sémantique à grands pas, nous avons besoin d'une opération de *substitution* pour remplacer dans le corps d'une abstraction, lors d'une application, le paramètre formel par l'argument effectif.

Définition 2 (substitution)**.** Si e est une expression, x une variable et v une valeur, on note $e[x \leftarrow v]$ la *substitution* de x par v dans e, définie par récurrence sur e de la manière suivante :

$$\begin{array}{rcl} x[x \leftarrow v] & = & v \\ y[x \leftarrow v] & = & y \quad \text{si } y \neq x \\ c[x \leftarrow v] & = & c \\ op[x \leftarrow v] & = & op \\ (\texttt{fun}\ x \rightarrow e)[x \leftarrow v] & = & \texttt{fun}\ x \rightarrow e \\ (\texttt{fun}\ y \rightarrow e)[x \leftarrow v] & = & \texttt{fun}\ y \rightarrow e[x \leftarrow v] \quad \text{si } y \neq x \\ (e_1\ e_2)[x \leftarrow v] & = & (e_1[x \leftarrow v]\ e_2[x \leftarrow v]) \\ (e_1, e_2)[x \leftarrow v] & = & (e_1[x \leftarrow v], e_2[x \leftarrow v]) \\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)[x \leftarrow v] & = & \texttt{let}\ x = e_1[x \leftarrow v]\ \texttt{in}\ e_2 \\ (\texttt{let}\ y = e_1\ \texttt{in}\ e_2)[x \leftarrow v] & = & \texttt{let}\ y = e_1[x \leftarrow v]\ \texttt{in}\ e_2[x \leftarrow v] \\ & & \text{si } y \neq x \end{array}$$

Dans le cas des constructions `fun` et `let`, on peut supposer que $y \notin fv(v)$, sans quoi la substitution créerait une capture de variable. Si ce n'est pas le cas, il suffit de renommer la variable y liée par `fun` ou `let`. □

Voici quelques exemples de substitutions. On prendra soin de bien faire attention aux parenthésage des expressions et aux lieurs.

$$\begin{array}{rcl} ((\texttt{fun}\ x \rightarrow +(x, x))\ x)\,[x \leftarrow 21] & = & (\texttt{fun}\ x \rightarrow +(x, x))\ 21 \\ (+(x,\ \texttt{let}\ x = 17\ \texttt{in}\ x))\,[x \leftarrow 3] & = & +(3,\ \texttt{let}\ x = 17\ \texttt{in}\ x) \\ (\texttt{fun}\ y \rightarrow y\ y)\,[y \leftarrow 17] & = & \texttt{fun}\ y \rightarrow y\ y \end{array}$$

Règles d'inférence. Pour définir la relation $e \twoheadrightarrow v$ qui nous intéresse, nous allons utiliser le concept de *règles d'inférence* issu de la logique. Il consiste à définir une relation comme la *plus petite relation* satisfaisant un ensemble d'axiomes de la forme

$$\frac{}{P}$$

et un ensemble d'implications de la forme

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{P}$$

On peut définir ainsi la relation $\mathsf{Pair}(n)$, qui caractérise les entiers naturels pairs, par les deux règles

$$\frac{}{\mathsf{Pair}(0)} \qquad \text{et} \qquad \frac{\mathsf{Pair}(n)}{\mathsf{Pair}(n+2)}$$

Ces deux règles doivent se lire comme

$$\begin{array}{rl} \text{d'une part} & \mathsf{Pair}(0), \\ \text{et d'autre part} & \forall n.\ \mathsf{Pair}(n) \Rightarrow \mathsf{Pair}(n+2). \end{array}$$

On peut se persuader facilement que la plus petite relation satisfaisant ces deux propriétés coïncide avec la propriété « n est un entier naturel pair ». D'une part, les entiers naturels pairs sont clairement dedans, par récurrence. D'autre part, s'il y avait au moins un entier impair, on pourrait enlever le plus petit, ce qui contredirait la minimalité de la relation.

Étant donnée une relation définie par des règles d'inférence, une *dérivation* est un arbre dont les nœuds correspondent aux règles et les feuilles aux axiomes. Avec l'exemple précédent, on a entre autres l'arbre

$$\cfrac{\cfrac{\cfrac{}{\mathsf{Pair}(0)}}{\mathsf{Pair}(2)}}{\mathsf{Pair}(4)}$$

On peut voir cet arbre comme une preuve que $\mathsf{Pair}(4)$ est vrai. Plus généralement, l'ensemble des dérivations possibles caractérise exactement la plus petite relation satisfaisant les règles d'inférence.

Pour établir une propriété d'une relation définie par un ensemble de règles d'inférence, on peut raisonner par *récurrence* structurelle sur la dérivation. Cela signifie que l'on peut appliquer l'hypothèse de récurrence à toute sous-dérivation. De manière équivalente, on peut dire qu'on raisonne par récurrence sur la hauteur de la dérivation. En pratique, on raisonne par récurrence sur la dérivation et par cas sur la dernière règle utilisée. Ainsi, à supposer que n est un entier relatif dans la définition de $\mathsf{Pair}(n)$ ci-dessus, on peut montrer $\forall n,\ \mathsf{Pair}(n) \Rightarrow n \geq 0$ par récurrence sur la dérivation de $\mathsf{Pair}(n)$.

Définition de la sémantique à grands pas. On est maintenant en mesure de définir la sémantique à grands pas de Mini-ML. Elle est donnée sous la forme de six règles d'inférences, une pour chaque construction du langage autre qu'une variable. Les trois premières règles expriment qu'une constante, une primitive et une abstraction sont déjà des valeurs.

$$\frac{}{c \twoheadrightarrow c} \qquad \frac{}{op \twoheadrightarrow op} \qquad \frac{}{(\texttt{fun}\ x \to e) \twoheadrightarrow (\texttt{fun}\ x \to e)}$$

La règle suivante définit l'évaluation d'une paire. Très naturellement, elle exprime que chacune des deux composantes doit être évaluée en une valeur et que la valeur finale est la paire de ces deux valeurs.

$$\frac{e_1 \twoheadrightarrow v_1 \quad e_2 \twoheadrightarrow v_2}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)}$$

Les deux dernières règles sont les plus complexes. Elles mettent en jeu les lieurs et l'opération de substitution. Pour l'expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, la sémantique exprime que e_1 doit s'évaluer en une valeur v_1, qui est ensuite substituée à x dans e_2 avant de pouvoir poursuivre l'évaluation.

$$\frac{e_1 \twoheadrightarrow v_1 \quad e_2[x \leftarrow v_1] \twoheadrightarrow v}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \twoheadrightarrow v}$$

D'une façon analogue, l'évaluation d'une application exprime le fait que l'argument doit d'abord être évalué, puis que sa valeur doit être substituée au paramètre formel dans le corps de l'abstraction avant d'évaluer celui-ci.

$$\frac{e_1 \twoheadrightarrow (\texttt{fun}\ x \rightarrow e) \quad e_2 \twoheadrightarrow v_2 \quad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

Ces deux dernières règles expriment donc le choix d'une stratégie d'*appel par valeur*, où l'argument est complètement évalué avant l'appel. On aurait pu faire d'autres choix, comme un appel par nom ou encore un appel par nécessité [2].

À ces règles doivent s'ajouter des règles spécifiques à chacune des primitives. Ainsi, pour que la primitive + représente effectivement l'addition de deux entiers, on ajoute la règle suivante :

$$\frac{e_1 \twoheadrightarrow + \quad e_2 \twoheadrightarrow (n_1, n_2) \quad n = n_1 + n_2}{e_1\ e_2 \twoheadrightarrow n}$$

Ici, n_1 et n_2 représentent des valeurs entières, c'est-à-dire des constantes qui se trouvent être des entiers (par opposition à d'autres constantes d'autres types, telles que *true* ou *false*). La prémisse $n = n_1 + n_2$ signifie que n désigne la constante entière qui est la somme de n_1 et n_2.

La figure 2.1 regroupe l'ensemble des règles définissant la sémantique à grands pas de Mini-ML, avec les primitives *opif* et *opfix* introduites plus haut et des primitives *fst* et *snd* pour les deux projections associées aux paires. Les deux règles (IF-TRUE) et (IF-FALSE) exigent notamment que les deux branches s'évaluent vers des fonctions, dont les corps e_3 et e_4 ne sont pas évalués. Ainsi, seule l'expression e_3 (resp. e_4) est évaluée lorsque le test est vrai (resp. faux). La dernière règle, appelée (FIX), exprime le fait que la primitive *opfix* représente un *opérateur de point fixe* vérifiant l'identité $\mathit{opfix}\ f = f\ (\mathit{opfix}\ f)$ pour toute fonction f.

Exemple. En appliquant les règles précédentes, on peut montrer que l'expression $\texttt{let}\ x = +(20, 1)\ \texttt{in}\ (\texttt{fun}\ y \rightarrow +(y, y))\ x$ s'évalue en la valeur 42. L'arbre de dérivation a la forme suivante :

$$\dfrac{\dfrac{\dfrac{}{+ \twoheadrightarrow +} \quad \dfrac{\dfrac{}{20 \twoheadrightarrow 20} \quad \dfrac{}{1 \twoheadrightarrow 1}}{(20,1) \twoheadrightarrow (20,1)}}{+(20,1) \twoheadrightarrow 21} \quad \dfrac{\dfrac{}{\texttt{fun}\ldots \twoheadrightarrow \texttt{fun}\ldots} \quad \dfrac{}{21 \twoheadrightarrow 21} \quad \dfrac{\vdots}{+(21,21) \twoheadrightarrow 42}}{(\texttt{fun}\ y \rightarrow +(y,y))\ 21 \twoheadrightarrow 42}}{\texttt{let}\ x = +(20,1)\ \texttt{in}\ (\texttt{fun}\ y \rightarrow +(y,y))\ x \twoheadrightarrow 42}$$

2. Ces notions seront introduites dans le chapitre 6.

$$\frac{}{c \twoheadrightarrow c}(\text{CST}) \qquad \frac{}{op \twoheadrightarrow op}(\text{OP}) \qquad \frac{}{(\texttt{fun}\ x \to e) \twoheadrightarrow (\texttt{fun}\ x \to e)}(\text{ABS})$$

$$\frac{e_1 \twoheadrightarrow v_1 \quad e_2 \twoheadrightarrow v_2}{(e_1,\, e_2) \twoheadrightarrow (v_1,\, v_2)}(\text{PAIR}) \qquad \frac{e_1 \twoheadrightarrow v_1 \quad e_2[x \leftarrow v_1] \twoheadrightarrow v}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \twoheadrightarrow v}(\text{LET})$$

$$\frac{e_1 \twoheadrightarrow (\texttt{fun}\ x \to e) \quad e_2 \twoheadrightarrow v_2 \quad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}(\text{APP})$$

$$\frac{e_1 \twoheadrightarrow + \quad e_2 \twoheadrightarrow (n_1, n_2) \quad n = n_1 + n_2}{e_1\ e_2 \twoheadrightarrow n}(\text{ADD})$$

$$\frac{e_1 \twoheadrightarrow \mathit{fst} \quad e_2 \twoheadrightarrow (v_1, v_2)}{e_1\ e_2 \twoheadrightarrow v_1}(\text{FST}) \qquad \frac{e_1 \twoheadrightarrow \mathit{snd} \quad e_2 \twoheadrightarrow (v_1, v_2)}{e_1\ e_2 \twoheadrightarrow v_2}(\text{SND})$$

$$\frac{e_1 \twoheadrightarrow \mathit{opif} \quad e_2 \twoheadrightarrow (\mathit{true}, ((\texttt{fun}\ _ \to e_3), (\texttt{fun}\ _ \to e_4))) \quad e_3 \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}(\text{IF-TRUE})$$

$$\frac{e_1 \twoheadrightarrow \mathit{opif} \quad e_2 \twoheadrightarrow (\mathit{false}, ((\texttt{fun}\ _ \to e_3), (\texttt{fun}\ _ \to e_4))) \quad e_4 \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}(\text{IF-FALSE})$$

$$\frac{e_1 \twoheadrightarrow \mathit{opfix} \quad e_2 \twoheadrightarrow (\texttt{fun}\ f \to e) \quad e[f \leftarrow \mathit{opfix}\ (\texttt{fun}\ f \to e)] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}(\text{FIX})$$

FIGURE 2.1 – Sémantique à grands pas de Mini-ML.

(Certaines parties de la dérivation ont été omises.)

Exercice 9. Donner la dérivation de l'expression

$$(\texttt{rec}\ \mathit{fact}\ n = \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times (n,\ \mathit{fact}\ (+(n,\ -1))))\ 2$$

On se rappellera que les constructions `rec` et `if-then-else` ont été introduites plus haut comme du sucre syntaxique pour les primitives *opfix* et *opif*. Solution □

Il existe des expressions e pour lesquelles il n'y a pas de valeur v telle que $e \twoheadrightarrow v$. L'exemple le plus simple est sûrement l'expression $1\ 2$, c'est-à-dire l'application de la constante 1 à la constante 2. Clairement, la règle (APP) ne s'applique pas car 1 ne s'évalue pas en une abstraction. Et parmi les règles spécifiques aux primitives, aucune ne traite le cas d'une application où la fonction s'évaluerait en une constante entière. Nous ne sommes absolument pas choqués par le fait que l'expression $1\ 2$ n'ait pas de valeur ; au contraire, c'est plutôt rassurant.

Il existe cependant d'autres expressions moins problématiques qui n'ont pas de valeur pour la sémantique à grands pas. Considérons l'expression

$$(\texttt{fun}\ x \to x\ x)\ (\texttt{fun}\ x \to x\ x) \tag{2.3}$$

c'est-à-dire l'application de l'expression $\texttt{fun}\ x \to x\ x$, que nous appellerons Δ par la suite, à elle-même. Il s'agit de l'application d'une abstraction à une valeur et le premier pas de calcul est donc la substitution de x par l'argument Δ dans le corps de la fonction, c'est-à-dire $x\ x$. On retombe donc exactement sur la même expression. On comprend qu'on vient de trouver là une expression dont l'évaluation ne termine pas.

Exercice 10. Montrer qu'il n'y a pas de valeur pour l'expression (2.3) dans la sémantique à grands pas. Solution □

Voici un premier résultat très simple concernant notre sémantique.

Proposition 1 (l'évaluation produit des valeurs closes)**.** Si $e \twoheadrightarrow v$ et si de plus e est close, alors v est close également.

PREUVE. On procède par récurrence sur la dérivation $e \twoheadrightarrow v$. Considérons le cas d'une application, c'est-à-dire une dérivation dont la conclusion est de la forme

$$\frac{\begin{matrix}(D_1)\\ \vdots \\ e_1 \twoheadrightarrow (\texttt{fun}\ x \to e_3)\end{matrix} \qquad \begin{matrix}(D_2)\\ \vdots \\ e_2 \twoheadrightarrow v_2\end{matrix} \qquad \begin{matrix}(D_3)\\ \vdots \\ e_3[x \leftarrow v_2] \twoheadrightarrow v\end{matrix}}{e_1\ e_2 \twoheadrightarrow v}$$

Si e est close alors les expressions e_1 et e_2 le sont également. Par hypothèses de récurrence appliquées aux dérivations (D_1) et (D_2), les valeurs $\texttt{fun}\ x \to e_3$ et v_2 sont également closes. On en déduit que l'expression $e_3[x \leftarrow v_2]$ est close, car x est la seule variable libre possible de e_3, et donc que v est close par hypothèse de récurrence appliquée à (D_3).

Les autres cas sont laissés au lecteur. □

Une autre propriété moins évidente de notre sémantique est qu'elle est déterministe, c'est-à-dire qu'une expression n'a qu'une seule valeur possible.

Proposition 2 (déterminisme de l'évaluation)**.** Si $e \twoheadrightarrow v$ et $e \twoheadrightarrow v'$ alors $v = v'$.

PREUVE. On procède par récurrence sur les dérivations $e \twoheadrightarrow v$ et de $e \twoheadrightarrow v'$. Examinons le cas d'une paire $e = (e_1, e_2)$. On a donc deux dérivations de la forme suivante :

$$\frac{\begin{matrix}(D_1) & & (D_2) \\ \vdots & & \vdots \\ e_1 \twoheadrightarrow v_1 & & e_2 \twoheadrightarrow v_2\end{matrix}}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)} \qquad \frac{\begin{matrix}(D'_1) & & (D'_2) \\ \vdots & & \vdots \\ e_1 \twoheadrightarrow v'_1 & & e_2 \twoheadrightarrow v'_2\end{matrix}}{(e_1, e_2) \twoheadrightarrow (v'_1, v'_2)}$$

Par hypothèses de récurrence, on a $v_1 = v'_1$ et $v_2 = v'_2$. On en déduit

$$v = (v_1, v_2) = (v'_1, v'_2) = v'.$$

Les autres cas sont laissés au lecteur. □

Il est important de noter que le déterminisme de l'évaluation n'est en rien une nécessité. On pourrait par exemple imaginer ajouter une primitive *random* et la règle

$$\frac{e_1 \twoheadrightarrow \mathit{random} \quad e_2 \twoheadrightarrow n_1 \quad 0 \leq n < n_1}{e_1\ e_2 \twoheadrightarrow n}$$

On aurait alors $\mathit{random}\ 2 \twoheadrightarrow 0$ aussi bien que $\mathit{random}\ 2 \twoheadrightarrow 1$.

Retour sur les primitives *opif* et *opfix*. Maintenant que nous connaissons la sémantique de Mini-ML, nous pouvons montrer comment définir les primitives *opif* et *opfix*. Rappelons qu'on a défini la conditionnelle comme

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \stackrel{\text{def}}{=} \mathit{opif}\ (e_1, ((\texttt{fun}\ _ \to e_2), (\texttt{fun}\ _ \to e_3)))$$

Dès lors, on peut définir les constantes booléennes et la primitive *opif* de la façon suivante :

$$\begin{array}{rcl} \mathit{true} & \stackrel{\text{def}}{=} & \texttt{fun}\ x \to \texttt{fun}\ y \to x\ () \\ \mathit{false} & \stackrel{\text{def}}{=} & \texttt{fun}\ x \to \texttt{fun}\ y \to y\ () \\ \mathit{opif} & \stackrel{\text{def}}{=} & \texttt{fun}\ p \to \mathit{fst}\ p\ (\mathit{fst}\ (\mathit{snd}\ p))\ (\mathit{snd}\ (\mathit{snd}\ p)) \end{array}$$

Ici, () désigne une constante dont la valeur n'est pas significative ; on aurait pu prendre un entier. L'idée est simple : lorsque e_1 s'évalue en *true* (resp. *false*), on applique la fonction dont le corps est e_2 (resp. e_3). On pourrait même faire plus simple, pour éviter les paires, en définissant dès le départ $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ comme $e_1\ (\texttt{fun}\ _ \to e_2)\ (\texttt{fun}\ _ \to e_3)$.

De même, il est possible de donner une définition à l'opérateur *opfix* que nous avions introduit pour définir des fonctions récursives, de la manière suivante :

$$\mathit{opfix} \stackrel{\text{def}}{=} \texttt{fun}\ f \to (\texttt{fun}\ x \to f\ (\texttt{fun}\ y \to x\ x\ y))\ (\texttt{fun}\ x \to f\ (\texttt{fun}\ y \to x\ x\ y))$$

Le corps de cette fonction ressemble étrangement au terme $\Delta\ \Delta$ dont nous avons montré que l'évaluation ne termine pas.

2.2.2 Sémantique à petits pas

La sémantique naturelle ne permet pas de distinguer les expressions dont le calcul « plante », comme 1 2, des expressions dont l'évaluation ne termine pas, comme $\Delta\ \Delta$. C'est assez gênant car lorsque nous nous intéresserons au typage statique, dans le chapitre 5, nous chercherons à montrer que les programmes bien typés s'évaluent sans planter. Et bien entendu on ne voudra pas rejeter des programmes au typage parce qu'ils sont susceptibles de ne pas terminer.

La sémantique opérationnelle *à petits pas* remédie à ce problème en introduisant une notion d'étape élémentaire de calcul, notée $e_1 \rightarrow e_2$, que l'on va itérer. Il devient alors possible de distinguer trois situations : soit l'itération aboutit à une valeur

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$$

soit l'itération bloque sur e_n irréductible qui n'est pas une valeur

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n$$

soit enfin l'itération ne termine pas

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots$$

Pour définir la relation $\rightarrow$, on commence par définir une relation $\stackrel{\epsilon}{\rightarrow}$ plus simple, correspondant à une réduction « en tête », c'est-à-dire impliquant la construction la plus extérieure de l'expression. Il y a en particulier deux règles de réduction en tête correspondant à la liaison d'une valeur à une variable :

$$\begin{array}{rcl} (\texttt{fun}\ x \rightarrow e)\ v & \stackrel{\epsilon}{\rightarrow} & e[x \leftarrow v] \\ \texttt{let}\ x = v\ \texttt{in}\ e & \stackrel{\epsilon}{\rightarrow} & e[x \leftarrow v] \end{array}$$

Ces deux règles traduisent le choix d'une stratégie d'*appel par valeur*, comme pour la sémantique à grands pas. On se donne également des règles de réduction en tête pour les primitives :

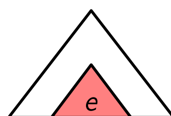
$$\begin{array}{rcl} +\ (n_1,\ n_2) & \stackrel{\epsilon}{\rightarrow} & n \qquad \text{avec } n = n_1 + n_2 \\ \\ \mathit{fst}\ (v_1,\ v_2) & \stackrel{\epsilon}{\rightarrow} & v_1 \\ \mathit{snd}\ (v_1,\ v_2) & \stackrel{\epsilon}{\rightarrow} & v_2 \\ \\ \mathit{opfix}\ (\texttt{fun}\ f \rightarrow e) & \stackrel{\epsilon}{\rightarrow} & e[f \leftarrow \mathit{opfix}\ (\texttt{fun}\ f \rightarrow e)] \\ \\ \mathit{opif}\ (\mathit{true},\ ((\texttt{fun}\ _ \rightarrow e_1),\ (\texttt{fun}\ _ \rightarrow e_2))) & \stackrel{\epsilon}{\rightarrow} & e_1 \\ \mathit{opif}\ (\mathit{false},\ ((\texttt{fun}\ _ \rightarrow e_1),\ (\texttt{fun}\ _ \rightarrow e_2))) & \stackrel{\epsilon}{\rightarrow} & e_2 \end{array}$$

Il convient ensuite d'expliquer comment le calcul va pouvoir être réalisé « en profondeur ». Si on considère par exemple l'expression $\texttt{let}\ x = +(1,\ 2)\ \texttt{in}\ +(x,\ x)$, aucune règle ci-dessus ne s'applique car il ne s'agit pas directement de l'addition de deux entiers. Il faut commencer par réduire la sous-expression $+(1,\ 2)$ avant de pouvoir obtenir la réduction en tête de $\texttt{let}\ x = 3\ \texttt{in}\ +(x,\ x)$. Pour désigner une sous-expression telle que $+(1,\ 2)$

dans l'exemple ci-dessus, on introduit la notion de *contexte*. Un contexte E est défini par la grammaire suivante :

$$\begin{array}{rcl} E & ::= & \Box \\ & | & E\ e \\ & | & v\ E \\ & | & \texttt{let}\ x = E\ \texttt{in}\ e \\ & | & (E,\ e) \\ & | & (v,\ E) \end{array}$$

Un contexte est un « terme à trou », ce dernier étant représenté par le symbole $\Box$. Comme on peut le constater, un contexte E contient exactement un trou. On définit alors $E(e)$ comme étant le contexte E dans lequel le trou a été remplacé par l'expression e, le résultat étant une expression. On peut le schématiser comme ceci

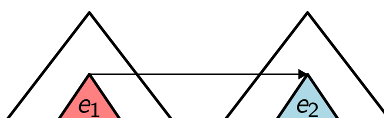


la partie blanche représentant le contexte E. Dans l'exemple donné plus haut, le contexte de la sous-expression $+(1,\ 2)$ est $\texttt{let}\ x = \Box\ \texttt{in}\ +(x,\ x)$.

On peut maintenant définir la sémantique à petits pas, c'est-à-dire la relation $\rightarrow$, en se servant de la notion de contexte pour effectuer des réductions de tête dans n'importe quel contexte. Une seule règle suffit pour cela :

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \rightarrow E(e_2)}$$

Elle exprime qu'une réduction de tête $e_1 \xrightarrow{\epsilon} e_2$ peut être effectuée à l'endroit spécifié par le contexte E. On peut le schématiser comme ceci



avec toujours la partie blanche représentant le contexte E. Si on reprend l'exemple de l'expression $\texttt{let}\ x = +(1,\ 2)\ \texttt{in}\ +(x,\ x)$, on peut montrer qu'elle s'évalue en la valeur 6 en trois étapes de réductions de la manière suivante :

$$\begin{array}{lll} & \texttt{let}\ x = +(1,\ 2)\ \texttt{in}\ +(x,\ x) & \\ \rightarrow & \texttt{let}\ x = 3\ \texttt{in}\ +(x,\ x) & \text{avec } E = \texttt{let}\ x = \Box\ \texttt{in}\ +(x,\ x) \\ \rightarrow & +(3,\ 3) & \text{avec } E = \Box \\ \rightarrow & 6 & \text{avec } E = \Box \end{array}$$

À chaque étape est utilisée une réduction de tête (respectivement, l'addition, le `let` et encore une fois l'addition) associée à un contexte indiqué à droite de la réduction. L'ensemble des règles de la sémantique à petits pas est donné figure 2.2.

Exercice 11. Donner les étapes de réduction de l'expression

$$(\texttt{rec}\ \mathit{fact}\ n = \texttt{if}\ =(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ \mathit{fact}\ (+(n,\ -1))))\ 2$$

On se rappellera que les constructions `rec` et `if-then-else` ont été introduites dans la section précédente comme du sucre syntaxique pour les primitives *opfix* et *opif*.

Solution $\Box$

réductions de tête

$$
\begin{array}{rcl}
(\texttt{fun}\ x \to e)\ v & \stackrel{\epsilon}{\to} & e[x \leftarrow v] \\
\texttt{let}\ x = v\ \texttt{in}\ e & \stackrel{\epsilon}{\to} & e[x \leftarrow v]
\end{array}
$$

$$
+\ (n_1,\ n_2) \stackrel{\epsilon}{\to} n \qquad \text{avec } n = n_1 + n_2
$$

$$
\begin{array}{rcl}
\mathit{fst}\ (v_1,\ v_2) & \stackrel{\epsilon}{\to} & v_1 \\
\mathit{snd}\ (v_1,\ v_2) & \stackrel{\epsilon}{\to} & v_2
\end{array}
$$

$$
\mathit{opfix}\ (\texttt{fun}\ f \to e) \stackrel{\epsilon}{\to} e[f \leftarrow \mathit{opfix}\ (\texttt{fun}\ f \to e)]
$$

$$
\begin{array}{rcl}
\mathit{opif}\ (\mathit{true},\ ((\texttt{fun}\ _ \to e_1),\ (\texttt{fun}\ _ \to e_2))) & \stackrel{\epsilon}{\to} & e_1 \\
\mathit{opif}\ (\mathit{false},\ ((\texttt{fun}\ _ \to e_1),\ (\texttt{fun}\ _ \to e_2))) & \stackrel{\epsilon}{\to} & e_2
\end{array}
$$

contextes de réduction

$$
\begin{array}{rcl}
E & ::= & \square \\
& | & E\ e \\
& | & v\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ e \\
& | & (E,\ e) \\
& | & (v,\ E)
\end{array}
$$

réduction dans un contexte

$$
\frac{e_1 \stackrel{\epsilon}{\to} e_2}{E(e_1) \to E(e_2)}
$$

FIGURE 2.2 – Sémantique à petits pas de Mini-ML.

On constate que les contextes ont une forme bien particulière, qui traduit un ordre d'évaluation de la gauche vers la droite. En effet, on a le contexte $E\ e$ qui permet de réduire à gauche d'une application mais on a seulement le contexte $v\ E$ pour réduire à droite d'une application. Dès lors, on est forcé de réduire le membre gauche d'une application en premier lieu. De la même manière, les contextes de l'évaluation d'une paire, respectivement $(E,\ e)$ et $(v,\ E)$, nous obligent à évaluer la composante de gauche en premier lieu. Ainsi, $(+(1,\ 2),\ \square)$ n'est pas un contexte d'évaluation valide.

On aurait très bien pu faire un autre choix, comme par exemple évaluer de la droite vers la gauche, ou encore ne pas fixer d'ordre d'évaluation en proposant aussi bien le contexte $E\ e$ que le contexte $e\ E$. Dans le cas de Mini-ML, cela ne ferait pas de différence quant à la valeur d'une expression, mais dans un langage plus complexe avec des effets de bord ou des comportements exceptionnels, évaluer dans un ordre plutôt que dans un autre peut faire une différence. Nous y reviendrons dans la section 2.4.

Définition 3 (évaluation et forme normale)**.** On note $\xrightarrow{\star}$ la clôture réflexive et transitive de la relation $\rightarrow$, c'est-à-dire $e_1 \xrightarrow{\star} e_2$ si et seulement si e_1 se réduit en e_2 en zéro, une ou plusieurs étapes. On appelle *forme normale* toute expression e telle qu'il n'existe pas d'expression e' telle que $e \rightarrow e'$. □

D'une façon évidente, les valeurs sont des formes normales. Les formes normales qui ne sont pas des valeurs sont les expressions erronées, comme 1 2.

2.2.3 Équivalence des deux sémantiques

Nous allons montrer que les deux sémantiques opérationnelles définies ci-dessus, à grands pas et à petits pas, sont équivalentes pour les expressions dont l'évaluation termine sur une valeur, *i.e.*,

$$e \twoheadrightarrow v \quad \text{si et seulement si} \quad e \xrightarrow{\star} v$$

Commençons par le sens « grands pas impliquent petits pas ». On a besoin du lemme suivant qui dit que les réductions à petits pas peuvent être effectuées en profondeur.

Lemme 1 (passage au contexte des réductions)**.** Supposons $e \rightarrow e'$. Alors pour toute expression e_2 et toute valeur v, on a

1. $e\ e_2 \rightarrow e'\ e_2$
2. $v\ e \rightarrow v\ e'$
3. $\texttt{let}\ x = e\ \texttt{in}\ e_2 \rightarrow \texttt{let}\ x = e'\ \texttt{in}\ e_2$
4. $(e,\ e_2) \rightarrow (e',\ e_2)$
5. $(v,\ e) \rightarrow (v,\ e')$

Preuve. De $e \rightarrow e'$ on sait qu'il existe un contexte E tel que

$$e = E(r) \qquad e' = E(r') \qquad r \xrightarrow{\epsilon} r'$$

Considérons le contexte $E_1 \stackrel{\text{def}}{=} E\ e_2$; alors

$$\frac{r \xrightarrow{\epsilon} r'}{E_1(r) \rightarrow E_1(r')} \qquad i.e. \qquad \frac{r \xrightarrow{\epsilon} r'}{e\ e_2 \rightarrow e'\ e_2}$$

On procède de même pour les cas 2 à 5. □

On peut maintenant montrer la première implication.

Proposition 3 (grands pas impliquent petits pas)**.** Si $e \twoheadrightarrow v$, alors $e \xrightarrow{\star} v$.

PREUVE. On procède par récurrence sur la dérivation de $e \twoheadrightarrow v$. Supposons que la dernière règle soit celle d'une application de fonction :

$$\frac{e_1 \twoheadrightarrow (\texttt{fun}\ x \to e_3) \quad e_2 \twoheadrightarrow v_2 \quad e_3[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

Par hypothèses de récurrence pour chacune des trois dérivations en prémisses, et en notant v_1 la valeur $\texttt{fun}\ x \to e_3$, on a les trois évaluations à petits pas

$$\begin{array}{l} e_1 \to \cdots \to v_1 \\ e_2 \to \cdots \to v_2 \\ e_3[x \leftarrow v_2] \to \cdots \to v \end{array}$$

Par passage au contexte (lemme précédent) on a donc également les trois évaluations

$$\begin{array}{l} e_1\ e_2 \to \cdots \to v_1\ e_2 \\ v_1\ e_2 \to \cdots \to v_1\ v_2 \\ e_3[x \leftarrow v_2] \to \cdots \to v \end{array}$$

En insérant la réduction

$$(\texttt{fun}\ x \to e_3)\ v_2 \xrightarrow{\epsilon} e_3[x \leftarrow v_2]$$

entre la deuxième et la troisième ligne, on obtient la réduction complète

$$e_1\ e_2 \to \cdots \to v$$

Les autres cas sont laissés au lecteur. □

Pour montrer l'autre implication, c'est-à-dire le sens « petits pas impliquent grand pas », on commence par établir deux lemmes. Le premier est une évidence.

Lemme 2 (les valeurs sont déjà évaluées)**.** $v \twoheadrightarrow v$ pour toute valeur v.

Le second indique qu'un petit pas suivi d'un grand pas est un grand pas.

Lemme 3 (réduction et évaluation)**.** Si $e \to e'$ et $e' \twoheadrightarrow v$, alors $e \twoheadrightarrow v$.

PREUVE. On commence par les réductions de tête *i.e.* $e \xrightarrow{\epsilon} e'$. Supposons par exemple que $e = (\texttt{fun}\ x \to e_1)\ v_2$ et $e' = e_1[x \leftarrow v_2]$. On peut construire la dérivation

$$\frac{(\texttt{fun}\ x \to e_1) \twoheadrightarrow (\texttt{fun}\ x \to e_1) \quad v_2 \twoheadrightarrow v_2 \quad e_1[x \leftarrow v_2] \twoheadrightarrow v}{(\texttt{fun}\ x \to e_1)\ v_2 \twoheadrightarrow v}$$

en utilisant le lemme précédent ($v_2 \twoheadrightarrow v_2$) et l'hypothèse $e' \twoheadrightarrow v$. On traite de façon similaire les autres cas d'une réduction de tête.

Montrons maintenant que si $e \xrightarrow{\epsilon} e'$ et $E(e') \twoheadrightarrow v$ alors $E(e) \twoheadrightarrow v$ pour un contexte E quelconque, par récurrence structurelle sur E (ou, si l'on préfère, par récurrence sur la hauteur du contexte). On vient de faire le cas de base $E = \square$.

Considérons par exemple le cas $E = E'\ e_2$. On a $E(e') \twoheadrightarrow v$ c'est-à-dire $E'(e')\ e_2 \twoheadrightarrow v$. Dans le cas d'une abstraction, cette réduction a la forme

$$\frac{E'(e') \twoheadrightarrow (\texttt{fun}\ x \to e_3) \quad e_2 \twoheadrightarrow v_2 \quad e_3[x \leftarrow v_2] \twoheadrightarrow v}{E'(e')\ e_2 \twoheadrightarrow v}$$

Par hypothèse de récurrence, on a $E'(e) \twoheadrightarrow (\texttt{fun}\ x \to e_3)$ et donc

$$\frac{E'(e) \twoheadrightarrow (\texttt{fun}\ x \to e_3) \quad e_2 \twoheadrightarrow v_2 \quad e_3[x \leftarrow v_2] \twoheadrightarrow v}{E'(e)\ e_2 \twoheadrightarrow v}$$

c'est-à-dire $E(e) \twoheadrightarrow v$. Les autres cas sont laissés au lecteur. □

On peut alors en déduire le second sens de l'équivalence.

Proposition 4 (petits pas impliquent grand pas)**.** Si $e \xrightarrow{\star} v$, alors $e \twoheadrightarrow v$.

Preuve. Écrivons la réduction à petits pas sous la forme $e \to e_1 \to \cdots \to e_n \to v$. On a $v \twoheadrightarrow v$ et donc par le lemme précédent $e_n \twoheadrightarrow v$. De même $e_{n-1} \twoheadrightarrow v$ et ainsi de suite jusqu'à $e \twoheadrightarrow v$. □

2.3 Interprète

Un *interprète* est un programme qui réalise l'exécution d'un autre programme, sur des entrées données. Il se distingue du compilateur par le fait que le travail est refait à chaque fois, pour tout nouveau programme ou toutes nouvelles entrées. Le compilateur, au contraire, construit un programme une fois pour toute, qui sera ensuite exécuté par la machine sur toute entrée.

On peut programmer un interprète en suivant les règles de la sémantique[3]. On se donne un type pour la syntaxe abstraite du langage et on écrit une fonction qui, étant donnée une expression du langage, calcule sa valeur. En substance, c'est ce que l'exercice 7 page 19 proposait de faire sur un langage minimal d'expressions arithmétique. Écrivons ici un interprète pour Mini-ML, dans le langage OCaml, qui suit la sémantique naturelle de la figure 2.1 page 25. On commence par se donner un type OCaml pour la syntaxe abstraite de Mini-ML.

```
type expression =
  | Var   of string
  | Const of int
  | Op    of string
  | Fun   of string * expression
  | App   of expression * expression
  | Paire of expression * expression
  | Let   of string * expression * expression
```

Les constantes sont ici limitées aux entiers et les primitives sont représentées par des chaînes de caractères. On réalise ensuite l'opération de substitution $e[x \leftarrow v]$ pour une valeur v close. Comme il n'y a pas de capture possible de variable, la définition reste simple[4].

3. On peut également définir la sémantique d'un langage par la donnée d'un interprète, qu'on appelle alors interprète de référence.

4. Sans l'hypothèse que v est close, on peut être amené à devoir renommer les lieurs sont lesquels la valeur v est substituée.

```
let rec subst e x v = match e with
  | Var y ->
    if y = x then v else e
  | Const _ | Op _ ->
    e
  | Fun (y, e1) ->
    if y = x then e else Fun (y, subst e1 x v)
  | App (e1, e2) ->
    App (subst e1 x v, subst e2 x v)
  | Paire (e1, e2) ->
    Paire (subst e1 x v, subst e2 x v)
  | Let (y, e1, e2) ->
    Let (y, subst e1 x v, if y = x then e2 else subst e2 x v)
```

Enfin, on écrit l'interprète sous la forme d'une fonction `eval` qui reçoit en argument une expression close et renvoie sa valeur. Pour une constante, une primitive ou une abstraction, l'expression est déjà une valeur.

```
let rec eval = function
  | Const _ | Op _ | Fun _  as v ->
    v
```

Pour une paire, il suffit d'évaluer chacune des composantes.

```
  | Paire (e1, e2) ->
    Paire (eval e1, eval e2)
```

Pour une expression de la forme `let x = e1 in e2`, on commence par évaluer `e1`, pour substituer ensuite sa valeur dans `e2` et poursuivre avec l'évaluation de cette nouvelle expression.

```
  | Let(x, e1, e2) ->
    eval (subst e2 x (eval e1))
```

Comme on évalue une expression close, la valeur de `e1` est une expression close (d'après la propriété 1 page 26) et on peut donc la substituer avec `subst`. Enfin, il reste à évaluer l'application. On se limite ici, outre le cas de l'application d'une abstraction, aux primitives +, *fst* et `snd`.

```
  | App (e1, e2) ->
    begin match eval e1 with
    | Fun (x, e) ->
      eval (subst e x (eval e2))
    | Op "+" ->
      let (Paire (Const n1, Const n2)) = eval e2 in
      Const (n1 + n2)
    | Op "fst" ->
      let (Paire(v1, v2)) = eval e2 in v1
    | Op "snd" ->
      let (Paire(v1, v2)) = eval e2 in v2
    end
```

Le filtrage est ici volontairement non-exhaustif. En particulier, notre interprète échoue si l'argument de + ne s'évalue pas en une paire d'entier, si l'argument de *fst* ou *snd* n'est

pas une paire, ou encore si l'expression `e1` s'évalue en autre chose qu'une abstraction ou l'une de ces trois primitives[5].

```
# eval (App (Const 1, Const 2));;
Exception: Match_failure ("", 87, 6).
```

En ce sens, notre interprète réalise un *typage dynamique*, par opposition au typage statique que nous verrons dans le chapitre 5. Notre interprète peut aussi ne pas terminer, par exemple sur ($\texttt{fun}\ x \rightarrow x\ x$) ($\texttt{fun}\ x \rightarrow x\ x$).

```
# let b = Fun ("x", App (Var "x", Var "x")) in
  eval (App (b, b));;
Interrupted.
```

Exercice 12. Ajouter les primitives *opif* et *opfix* à cet interprète. Solution □

Un interprète plus efficace. Notre interprète de Mini-ML n'est pas très efficace car il passe son temps à effectuer des substitutions et donc à reconstruire des expressions. Une idée simple et naturelle pour éviter ces substitutions consiste à maintenir dans un dictionnaire les valeurs des variables qui sont connues. On appelle *environnement* un tel dictionnaire. Notre fonction `eval` va donc prendre le type suivant

```
  val eval: environment -> expression -> value
```

où le type `environment` peut être facilement réalisé de manière purement applicative avec les dictionnaires du module `Map` de la bibliothèque standard d'OCaml :

```
  module Smap = Map.Make(String)
  type environment = value Smap.t
```

Il y a cependant une difficulté dans cette approche. Si on évalue l'expression

$$\texttt{let}\ x = 1\ \texttt{in}\ \texttt{fun}\ y \rightarrow +(x, y)$$

le résultat est une fonction qui doit « mémoriser » que $x = 1$. Dans notre premier interprète, c'est la substitution de x par 1 dans $\texttt{fun}\ y \rightarrow +(x, y)$ qui assurait cela. Mais puisque nous cherchons justement à éviter la substitution, il faut trouver un autre moyen de mémoriser la valeur de x dans $\texttt{fun}\ y \rightarrow +(x, y)$. La solution consiste à attacher un environnement à toute valeur de la forme $\texttt{fun}\ y \rightarrow e$. On appelle cela une *fermeture*[6]. On définit donc un nouveau type `value` pour les valeurs :

```
type value =
  | Vconst of int
  | Vop    of string
  | Vpair  of value * value
  | Vfun   of string * environment * expression
and environment = value Smap.t
```

5. Bien entendu, on pourrait programmer plus proprement cet échec, avec un type `option` ou une exception plus explicite.

6. Nous en reparlerons au chapitre 7 lorsque nous compilerons les langages comme Mini-ML.

On peut maintenant écrire la fonction `eval` qui calcule la valeur d'une expression dans un environnement donné. On commence par les cas simples d'une expression qui est déjà une valeur :

```
let rec eval env = function
  | Const n ->
      Vconst n
  | Op op ->
      Vop op
  | Fun (x, e) ->
      Vfun (x, env, e)
```

Dans ce dernier cas, on sauvegarde l'environnement `env` dans la valeur de type `Vfun`. Pour cette raison, il est important d'avoir choisi ici une structure purement applicative pour représenter les environnements. Pour une paire, il suffit d'évaluer ses deux composantes :

```
  | Pair (e1, e2) ->
      Vpair (eval env e1, eval env e2)
```

Le principal changement par rapport à l'interprète précédent se situe dans la construction et l'utilisation de l'environnement. On y ajoute la valeur d'une variable lorsqu'on rencontre une construction `let`

```
  | Let (x, e1, e2) ->
      eval (Smap.add x (eval env e1) env) e2
```

et on la consulte lorsque l'expression est une variable :

```
  | Var x ->
      Smap.find x env
```

Vient enfin le cas d'une application $e_1\ e_2$. Comme dans l'interprète précédent, on commence par évaluer e_1 et par examiner sa valeur. Le cas intéressant est celui de l'application d'une fonction `fun` $x \to e$. L'environnement contenu dans la fermeture `Vfun` est récupéré, étendu avec la valeur de x, c'est-à-dire la valeur de e_2, puis le corps e de la fonction est évalué dans cet environnement.

```
  | App (e1, e2) ->
      begin match eval env e1 with
      | Vfun (x, clos, e) ->
          eval (Smap.add x (eval env e2) clos) e
```

On utilise notamment ici le fait qu'une variable y apparaissant dans e est soit la variable x, soit une variable dont la valeur est nécessairement donnée par l'environnement `clos`, puisqu'on est en train d'évaluer une expression close. Les autres cas de l'application correspondent à des primitives et ne diffèrent pas de ce que nous avions écrit précédemment.

```
      | Vop "+" ->
          let Vpair (Vconst n1, Vconst n2) = eval env e2 in
          Vconst (n1 + n2)
      | Vop "fst" ->
          let Vpair (v1, _) = eval env e2 in v1
      | Vop "snd" ->
          let Vpair (_, v2) = eval env e2 in v2
      end
```

Exercice 13. Expliquer pourquoi il serait difficile de réaliser l'environnement par une structure impérative, par exemple `type environment = value Hashtbl.t`. Solution □

2.4 Langages impératifs

On peut définir une sémantique opérationnelle, à grands pas ou à petits pas, pour un langage avec des traits impératifs tels que des variables mutables, des tableaux, des exceptions, etc. Prenons l'exemple d'un petit langage impératif avec des variables mutables, que nous supposerons uniquement globales et fixées à l'avance pour simplifier légèrement. La syntaxe abstraite des expressions est

$$\begin{array}{rcll} e & ::= & c & \text{constante } (\texttt{true}, 42, \dots) \\ & | & x & \text{variable globale} \\ & | & e \; op \; e & \text{opérateur binaire } (+, <, \dots) \end{array}$$

Pour définir la sémantique d'une expression, il faut se donner un *état* dans lequel évaluer cette expression. Un état est ici une fonction E qui associe à chaque variable la valeur qu'elle contient. Dès lors, la relation de réduction, qu'elle soit à grands pas ou à petits pas, ne relie plus seulement des expressions, mais des couples état/expression. Ainsi, la sémantique à grands pas prendra la forme

$$E,\ e \twoheadrightarrow v$$

où v désigne la valeur de l'expression e dans l'état E. Une telle relation est facilement définie avec des règles telles que

$$\frac{}{E, c \twoheadrightarrow c} \qquad \frac{}{E, x \twoheadrightarrow E(x)} \qquad \frac{E, e_1 \twoheadrightarrow n_1 \quad E, e_2 \twoheadrightarrow n_2 \quad n = n_1 + n_2}{E, e_1 + e_2 \twoheadrightarrow n} \qquad \text{etc.}$$

On pourrait également choisir une sémantique à petits pas, mais c'est inutile car l'évaluation d'une expression termine toujours. On se donne maintenant une syntaxe abstraite pour des instructions[7] :

$$\begin{array}{rcll} s & ::= & x \leftarrow e & \text{affectation} \\ & | & \texttt{if } e \texttt{ then } s \texttt{ else } s & \text{conditionnelle} \\ & | & \texttt{while } e \texttt{ do } s & \text{boucle} \\ & | & s;\ s & \text{séquence} \\ & | & \texttt{skip} & \text{ne rien faire} \end{array}$$

L'évaluation d'une instruction pouvant ne pas terminer, on va se donner une sémantique à petits pas pour les instructions, sous la forme d'une relation

$$E,\ s \to E'$$

où E et E' représentent respectivement l'état au début et à la fin de l'évaluation de l'instruction s. Les règles définissant cette relation sont les suivantes :

$$\frac{E, e \twoheadrightarrow v}{E, x \leftarrow e \to E\{x \mapsto v\}, \texttt{skip}}$$

7. Un tel langage est connu sous le nom de « langage WHILE ».

$$\frac{}{E, \texttt{skip}; s \to E, s} \qquad \frac{E, s_1 \to E_1, s_1'}{E, s_1; s_2 \to E_1, s_1'; s_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{true}}{E, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to E, s_1} \qquad \frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to E, s_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{true}}{E, \texttt{while } e \texttt{ do } s \to E, s; \texttt{while } e \texttt{ do } s}$$

$$\frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{while } e \texttt{ do } s \to E, \texttt{skip}}$$

On notera en particulier comment la boucle `while` est « dépliée » lorsque son test est évalué en `true`. Enfin, on peut dire que l'évaluation d'une instruction s termine dans un état E si on a

$$E, s \to^{\star} E', \texttt{skip}$$

pour un certain état E'.

Exercice 14. Donner une sémantique à grands pas pour l'évaluation des instructions.
Solution □

2.5 Preuve de correction d'un compilateur

La sémantique formelle est un outil puissant, qui peut notamment être utilisé pour montrer la correction d'un compilateur. Par cela on entend que si le langage source est muni d'une sémantique $\to_s$ et le langage machine d'une sémantique $\to_m$, et si l'expression e est compilée en $C(e)$ alors on doit avoir un « diagramme qui commute » de la forme

$$\begin{array}{ccc} e & \xrightarrow{\star}_s & v \\ \downarrow & & \approx \\ C(e) & \xrightarrow{\star}_m & v' \end{array}$$

où $v \approx v'$ exprime que les valeurs v et v' coïncident. Illustrons-le sur l'exemple minimaliste d'un langage d'expressions arithmétiques ne contenant que des constantes entières et des additions.

$$e ::= n \mid e + e$$

Notre compilateur produit ici du code assembleur x86-64 destiné à mettre au final la valeur de l'expression dans le registre `%rdi`, en utilisant la pile pour stocker les résultats intermédiaires.

$$\begin{array}{rcl} code(n) & = & \texttt{movq } \$n, \texttt{\%rdi} \\ \\ code(e_1 + e_2) & = & code(e_1) \\ & & \texttt{addq } \ \$-8, \texttt{\%rsp} \\ & & \texttt{movq } \ \texttt{\%rdi}, (\texttt{\%rsp}) \\ & & code(e_2) \\ & & \texttt{movq } \ (\texttt{\%rsp}), \texttt{\%rsi} \\ & & \texttt{addq } \ \$8, \texttt{\%rsp} \\ & & \texttt{addq } \ \texttt{\%rsi}, \texttt{\%rdi} \end{array}$$

Pour montrer formellement la correction de ce micro-compilateur, on commence par se donner une sémantique pour chacun des deux langages. On opte ici pour la sémantique opérationnelle à petits pas. Pour le langage source, elle se définit simplement avec les notions suivantes de valeur et de contexte

$$\begin{array}{rcl} v & ::= & n \\ E & ::= & \square \mid E + e \mid v + E \end{array}$$

et l'unique règle de réduction de tête

$$n_1 + n_2 \xrightarrow{\epsilon} n \quad \text{avec } n = n_1 + n_2.$$

Pour le langage cible, ici l'assembleur x86-64, on commence par introduire les notions d'instruction m et de registre r, en se limitant ici aux instructions et aux registres utilisés par le compilateur.

$$\begin{array}{rcl} m & ::= & \texttt{movq } \$n, r \\ & \mid & \texttt{addq } \$n, r \mid \texttt{addq } r, r \\ & \mid & \texttt{movq } (r), r \mid \texttt{movq } r, (r) \mid \\ r & ::= & \texttt{\%rdi} \mid \texttt{\%rsi} \mid \texttt{\%rsp} \end{array}$$

Comme il s'agit là d'un langage impératif, on définit un état comme les valeurs des trois registres d'une part, noté R, et d'un état de la mémoire d'autre part, noté M.

$$\begin{array}{rcl} R & ::= & \{\texttt{\%rdi} \mapsto n; \texttt{\%rsi} \mapsto n; \texttt{\%rsp} \mapsto n\} \\ M & ::= & \mathbb{N} \to \mathbb{Z} \end{array}$$

La sémantique opérationnelle d'une instruction m peut être alors définie par une réduction de la forme

$$R, M, m \xrightarrow{m} R', M'$$

où R, M désigne l'état avant l'exécution de l'instruction et R', M' l'état après l'exécution. Cette réduction est définie sans difficulté, avec une règle pour chaque instruction.

$$\begin{array}{rcl} R, M, \texttt{movq } \$n, r & \xrightarrow{m} & R\{r \mapsto n\}, M \\ R, M, \texttt{addq } \$n, r & \xrightarrow{m} & R\{r \mapsto R(r) + n\}, M \\ R, M, \texttt{addq } r_1, r_2 & \xrightarrow{m} & R\{r_2 \mapsto R(r_1) + R(r_2)\}, M \\ R, M, \texttt{movq } (r_1), r_2 & \xrightarrow{m} & R\{r_2 \mapsto M(R(r_1))\}, M \\ R, M, \texttt{movq } r_1, (r_2) & \xrightarrow{m} & R, M\{R(r_2) \mapsto R(r_1)\} \end{array}$$

On peut maintenant montrer la correction de la compilation. Précisément, on souhaite montrer que

$$\text{si } e \xrightarrow{\star} n \text{ et } R, M, \mathit{code}(e) \xrightarrow{m}{}^{\star} R', M' \text{ alors } R'(\texttt{\%rdi}) = n.$$

On va naturellement procéder par récurrence structurelle sur e. Malheureusement, la preuve ne passe pas car, lors de la compilation de $e_1 + e_2$, l'hypothèse de récurrence sur e_2 ne nous permet pas d'assurer que la valeur de e_1 déposée sur la pile n'a pas été corrompue par la compilation de e_2. Il nous faut donc établir un résultat plus fort, à savoir

$$\text{si } e \xrightarrow{\star} n \text{ et } R, M, \mathit{code}(e) \xrightarrow{m}{}^{\star} R', M' \text{ alors } \left\{ \begin{array}{l} R'(\texttt{\%rdi}) = n \\ R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) \\ \forall a \geq R(\texttt{\%rsp}),\ M'(a) = M(a) \end{array} \right.$$

Le cas $e = n$ s'établit trivialement, car on a $e \xrightarrow{\star} n$ et $code(e) = \texttt{movq}\ \$n, \texttt{\%rdi}$. Dans le cas où $e = e_1 + e_2$, on a $e \xrightarrow{\star} n_1 + e_2 \xrightarrow{\star} n_1 + n_2$ avec $e_1 \xrightarrow{\star} n_1$ et $e_2 \xrightarrow{\star} n_2$, ce qui nous permet d'invoquer l'hypothèse de récurrence sur e_1 et e_2. La preuve est alors conduite avec les étapes suivantes :

code	état	justification
$code(e_1)$	R_1, M_1	par hypothèse de récurrence $R_1(\texttt{\%rdi}) = n_1$ et $R_1(\texttt{\%rsp}) = R(\texttt{\%rsp})$ $\forall a \geq R(\texttt{\%rsp}),\ M_1(a) = M(a)$
`addq  $-8,%rsp` `movq  %rdi,(%rsp)`	R_1', M_1'	$R_1' = R_1\{\texttt{\%rsp} \mapsto R(\texttt{\%rsp}) - 8\}$ $M_1' = M_1\{R(\texttt{\%rsp}) - 8 \mapsto n_1\}$
$code(e_2)$	R_2, M_2	par hypothèse de récurrence $R_2(\texttt{\%rdi}) = n_2$ et $R_2(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8$ $\forall a \geq R(\texttt{\%rsp}) - 8,\ M_2(a) = M_1'(a)$

`movq  (%rsp),%rsi`
`addq  $8,%rsp`
`addq  %rsi,%rdi`
	R', M_2	$R'(\texttt{\%rdi}) = n_1 + n_2$
		$R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8 + 8 = R(\texttt{\%rsp})$
		$\forall a \geq R(\texttt{\%rsp}),$
		$M_2(a) = M_1'(a) = M_1(a) = M(a)$

Ceci conclut la preuve de notre micro-compilateur.

Notes bibliographiques. La sémantique opérationnelle a été introduite par Gordon Plotkin en 1981 [23]. Un excellent ouvrage d'introduction à la sémantique opérationnelle est *Types and Programming Languages* de Benjamin Pierce [22]. Il traite aussi des interprètes et du typage — nous parlerons du typage plus loin, dans le chapitre 5. Mini-ML a été introduit par Gilles Kahn et ses collègues dans un article de 1985 [13]. Le compilateur CompCert est un compilateur C dont la correction a été vérifiée avec l'assistant de preuve Coq. CompCert a été conçu et développé par Xavier Leroy [19].

Il existe bien d'autres façons de définir la sémantique d'un langage de programmation. On peut faire par exemple le choix d'une *sémantique axiomatique* qui caractérise les constructions du langage par les propriétés logiques qu'elles permettent d'établir. La plus connue des sémantiques axiomatiques est la *logique de Hoare*, introduite dans l'article devenu célèbre *An axiomatic basis for computer programming* [12]. Cet article introduit la notion de triplet $\{P\}\ i\ \{Q\}$ signifiant « si la formule P est vraie avant l'exécution de l'instruction i, alors la formule Q sera vraie après », ainsi qu'un ensemble de règles de déduction pour établir de tels triplets. La règle pour l'affectation, par exemple, s'énonce ainsi :

$$\{P[x \leftarrow E]\}\ x := E\ \{P(x)\}$$

Avec cette règle, on peut établir la validité du triplet $\{x \geq 0\}\ x := x + 1\ \{x > 0\}$, en supposant que l'addition ne provoque pas de débordement arithmétique.

La *sémantique dénotationnelle* associe à chaque expression de programme e sa dénotation $[\![e]\!]$, qui est un objet mathématique représentant le calcul désigné par e. Si on prend l'exemple d'expressions arithmétiques très simples avec *une seule variable* `x`, c'est-à-dire

$$e ::= \texttt{x} \mid n \mid e \texttt{+} e \mid e \texttt{*} e \mid \ \ldots$$

la dénotation peut être une fonction qui associe à la valeur de la variable `x` la valeur de l'expression, c'est-à-dire

$$\begin{aligned}
[\![\mathtt{x}]\!] &= x \mapsto x \\
[\![n]\!] &= x \mapsto n \\
[\![e_1 \mathtt{+} e_2]\!] &= x \mapsto [\![e_1]\!](x) + [\![e_2]\!](x) \\
[\![e_1 \mathtt{*} e_2]\!] &= x \mapsto [\![e_1]\!](x) \times [\![e_2]\!](x)
\end{aligned}$$

Enfin, mentionnons aussi la *sémantique par traduction*, encore appelée sémantique dénotationnelle à la Strachey [25]. Elle consiste à définir la sémantique d'un langage en le traduisant vers un langage dont la sémantique est déjà connue.

Deuxième partie

Partie avant du compilateur

3 Analyse lexicale

L'analyse lexicale est le découpage du texte source en « mots ». De même que dans les langues naturelles, ce découpage en mots facilite le travail de la phase suivante, l'analyse syntaxique, qui sera expliquée dans le chapitre suivant.

Dans le contexte de l'analyse lexicale, les mots sont appelés des *lexèmes* (en anglais *tokens*). Si on est par exemple en train de réaliser l'analyse lexicale d'un langage tel que OCaml, et que le texte source est de la forme

```
fun x -> (* ma fonction *)
    x + 1
```

alors la liste des lexèmes construite par l'analyseur lexical sera de la forme

fun	ident(x)	->	ident(x)	+	const(1)

c'est-à-dire une séquence de six lexèmes, représentant successivement le mot-clé `fun`, l'identificateur `x`, le symbole `->`, etc. En particulier, les espaces, retours chariot et commentaires présents dans le texte source initial, qu'on appelle les *blancs*, n'apparaissent pas dans la séquence de lexèmes renvoyée. Ils jouent néanmoins un rôle dans l'analyse lexicale. Ils permettent notamment de séparer deux lexèmes. Ainsi `funx`, écrit sans espace entre `fun` et `x`, aurait été compris comme un seul lexème, à savoir l'identificateur `funx`. D'autres blancs sont inutiles, comme les espaces dans `x + 1`, et simplement ignorés.

Les conventions lexicales diffèrent selon les langages, et certains des caractères « blancs » peuvent parfois être significatifs. Un exemple est le langage d'entrée de l'outil `make`, où les tabulations de début de ligne sont significatives. Les langages Python et Haskell sont des exemples où les retours chariot et espaces de début de ligne sont significatifs, car utilisés pour définir la structure.

Comme nous venons de le dire, les commentaires jouent le rôle de blancs. Ainsi, si on écrit

```
fun(* et hop *)x -> x + (* j'ajoute un *) 1
```

alors le commentaire `(* et hop *)` joue le rôle d'un blanc, utile pour séparer deux lexèmes, et le commentaire `(* j'ajoute un *)` celui d'un blanc inutile (en plus d'être un commentaire inutile). Les commentaires sont parfois exploités par certains outils

(`ocamldoc`, `javadoc`, etc.), qui les traitent alors différemment dans leur propre analyse lexicale.

Pour réaliser l'analyse lexicale, on va utiliser des *expressions régulières* d'une part, pour décrire les lexèmes, et des *automates finis* d'autre part, pour les reconnaître. On exploite notamment la capacité à construire mécaniquement un automate fini reconnaissant le langage décrit par une expression régulière.

Dans tout ce qui suit, on se donne un alphabet A, qui représente l'ensemble des caractères des textes sources à analyser. En pratique, il s'agira des caractères ASCII 7 bits, des caractères UTF-8, etc. Un *mot* sur l'alphabet A est une séquence finie, possiblement vide, de caractères. Si un mot est formé des caractères a_1, a_2, ..., a_n, dans cet ordre, on le note naturellement $a_1 a_2 \cdots a_n$ et on appelle n sa longueur. Le mot vide, de longueur zéro, est noté ϵ. Un *langage* est un ensemble de mots, non nécessairement fini. L'ensemble de tous les mots possibles est un langage parmi d'autres, noté $A^\star$.

3.1 Expression régulière

Les expressions régulières constituent un moyen de définir des langages. Nous commençons par introduire leur syntaxe.

Définition 4 (expression régulière)**.** La syntaxe abstraite des expressions régulières est la suivante :

$$
\begin{array}{rcll}
r & ::= & \emptyset & \text{langage vide} \\
& | & \epsilon & \text{mot vide} \\
& | & a & \text{caractère } a \in A \\
& | & r\,r & \text{concaténation} \\
& | & r\,|\,r & \text{alternative} \\
& | & r\star & \text{étoile}
\end{array}
$$

□

Pour écrire des expressions régulières par la suite, nous prenons la convention que l'étoile a la priorité la plus forte, puis la concaténation, puis enfin l'alternative. Si r est une expression régulière et n un entier naturel, on définit l'expression régulière r^n par récurrence sur n en posant $r^0 = \epsilon$ et $r^{n+1} = r\,r^n$.

Définition 5 (langage d'une expression régulière)**.** Le *langage* défini par l'expression régulière r est l'ensemble de mots $L(r)$ défini par

$$
\begin{aligned}
L(\emptyset) &= \emptyset \\
L(\epsilon) &= \{\epsilon\} \\
L(a) &= \{a\} \\
L(r_1\,r_2) &= \{w_1 w_2 \mid w_1 \in L(r_1) \wedge w_2 \in L(r_2)\} \\
L(r_1\,|\,r_2) &= L(r_1) \cup L(r_2) \\
L(r\star) &= \textstyle\bigcup_{n \geq 0} L(r^n)
\end{aligned}
$$

□

On constate en particulier que $L(r_1\,(r_2\,r_3)) = L((r_1\,r_2)\,r_3)$, c'est-à-dire que la concaténation est associative. Pour cette raison, on s'épargnera des parenthèses inutiles en écrivant directement $r_1\,r_2\,r_3$, car il n'y a pas d'ambiguïté. De la même manière, on écrira $r_1 \,|\, r_2 \,|\, r_3$ car $L(r_1 \,|\, (r_2 \,|\, r_3)) = L((r_1 \,|\, r_2) \,|\, r_3)$.

Exemples. Sur l'alphabet $A = \{a, b\}$, l'expression régulière $r = (a|b)(a|b)(a|b)$ définit le langage des mots de trois caractères, c'est-à-dire $L(r) = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$. L'expression régulière $(a|b)\star a$ définit le langage (infini) des mots se terminant par a. Enfin, l'expression régulière $(b|\epsilon)(ab) \star (a|\epsilon)$ définit le langage des mots alternant a et b.

Exercice 15. Toujours sur l'alphabet $A = \{a, b\}$, donner une expression régulière définissant le langage des mots contenant au plus deux caractères b.

Solution □

3.2 Automate fini

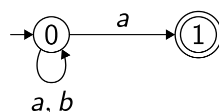
Les automates finis constituent un autre moyen de définir des langages.

Définition 6 (automate fini)**.** Un automate fini sur un alphabet A est un quadruplet (Q, T, I, F) où

- Q est un ensemble fini d'états ;
- $T \subseteq Q \times A \times Q$ un ensemble de transitions ;
- $I \subseteq Q$ un ensemble d'états initiaux ;
- $F \subseteq Q$ un ensemble d'états terminaux.

Si de plus T est une fonction de ces deux premiers arguments, c'est-à-dire que pour $q \in Q$ et $q \in A$ il existe au plus un $q' \in Q$ tel que $(q, a, q') \in T$, on dit que l'automate est *déterministe*. □

Exemple. Sur l'alphabet $A = \{a, b\}$, on peut considérer l'automate défini par $Q = \{0, 1\}$, $T = \{(0, a, 0), (0, b, 0), (0, a, 1)\}$, $I = \{0\}$ et $F = \{1\}$. On représente traditionnellement un tel automate sous la forme



avec une flèche entrante sur les états initiaux et un double encerclement des états terminaux. Une transition $(q, c, q') \in Q$ est représentée par un arc reliant les états q et q', étiqueté par le caractère c. Cet automate n'est pas déterministe, car il y a deux transitions étiquetées par a sortant de l'état 0.

Définition 7 (langage d'un automate fini)**.** Un mot $a_1 a_2 \dots a_n \in A^\star$ est *reconnu* par un automate (Q, T, I, F) si et seulement s'il existe des transitions

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s_n$$

avec $s_0 \in I$, $s_n \in F$ et $(s_{i-1}, a_i, s_i) \in T$ pour tout i tel que $1 \leq i \leq n$. Le langage défini par un automate est l'ensemble des mots reconnus par cet automate. □

Ainsi, l'automate fini donné en exemple plus haut définit le langage des mots se terminant par le caractère a.

Les expressions régulières et les automates finis sont liés par un résultat fondamental de la théorie des langages qui dit qu'ils définissent tous deux les mêmes langages. Cela signifie qu'un langage défini par une expression régulière l'est aussi par (au moins) un automate fini et que le langage défini par un automate fini l'est aussi par (au moins) une expression régulière. Ainsi, l'automate fini ci-dessus définit le même langage que l'expression régulière $(a|b) \star a$.

Construction de l'automate fini. Il existe de multiples façons de construire l'automate fini correspondant à une expression régulière donnée. Nous illustrons ici une méthode effectivement mise en pratique dans la construction d'analyseurs lexicaux. L'idée consiste à mettre en correspondance les caractères d'un mot reconnu et celles apparaissant dans l'expression régulière. Ainsi, le mot *aabaab* appartient au langage de l'expression régulière $(a|b) \star a(a|b)$ et on peut faire la correspondance suivante :

$$\begin{array}{ll} a & (\mathbf{a}|b) \star a(a|b) \\ a & (\mathbf{a}|b) \star a(a|b) \\ b & (a|\mathbf{b}) \star a(a|b) \\ a & (\mathbf{a}|b) \star a(a|b) \\ a & (a|b) \star \mathbf{a}(a|b) \\ b & (a|b) \star a(a|\mathbf{b}) \end{array}$$

Cette correspondance n'est pas nécessairement unique, même si ici elle l'est.

On va construire un automate fini dont les états sont des ensembles de caractères de l'expression régulière. Pour cela, nous allons distinguer les différents caractères de l'expression régulière, par exemple en leur associant des indices distincts :

$$(a_1|b_1) \star a_2(a_3|b_2)$$

Cette distinction reste conceptuelle : a_1, a_2 et a_3 représentent toujours le caractère a et b_1 et b_2 représentent toujours le caractère b. À partir d'un état s de notre automate, on va reconnaître des mots qui sont des suffixes de mots reconnus par l'expression régulière et dont la première lettre appartient à s. Dans l'exemple ci-dessus, on reconnaîtra à partir de l'état $\{a_1, a_2, b_1\}$ des mots dont le premier caractère peut être mis en correspondance avec a_1, a_2 ou b_1.

Pour construire les transitions entre un état s_1 et un état s_2, il faut déterminer les caractères qui peuvent apparaître après un autre dans un mot reconnu. Appelons $\mathit{follow}(c, r)$ les caractères qui peuvent apparaître après le caractère c dans un mot reconnu par l'expression régulière r. Par exemple, toujours avec $r = (a_1|b_1) \star a_2(a_3|b_2)$, on a

$$\mathit{follow}(a_1, r) = \{a_1, a_2, b_1\}$$

Pour calculer *follow*, on a besoin de calculer les premières (resp. dernières) lettres possibles d'un mot reconnu. Notons $\mathit{first}(r)$ (resp. $\mathit{last}(r)$) cet ensemble pour une expression régulière r donnée. Toujours avec le même exemple, on a

$$\begin{array}{rcl} \mathit{first}(r) & = & \{a_1, a_2, b_1\} \\ \mathit{last}(r) & = & \{a_3, b_2\} \end{array}$$

Et pour calculer *first* et *last*, on a besoin d'une dernière notion : est-ce que le mot vide appartient au langage $L(r)$? Notons $null(r)$ cette propriété. Il s'avère qu'il est très facile de déterminer $null(r)$ par récurrence structurelle sur l'expression régulière r, de la manière suivante :

$$
\begin{array}{rcl}
null(\emptyset) & = & \texttt{false} \\
null(\epsilon) & = & \texttt{true} \\
null(a) & = & \texttt{false} \\
null(r_1\, r_2) & = & null(r_1) \wedge null(r_2) \\
null(r_1 \,|\, r_2) & = & null(r_1) \vee null(r_2) \\
null(r\star) & = & \texttt{true}
\end{array}
$$

On peut en déduire alors la définition de *first* et *last*, là encore en procédant par récurrence structurelle sur l'expression régulière r. Pour *first*, on procède de la manière suivante :

$$
\begin{array}{rcl}
first(\emptyset) & = & \emptyset \\
first(\epsilon) & = & \emptyset \\
first(a) & = & \{a\} \\
first(r_1\, r_2) & = & first(r_1) \cup first(r_2) \quad \text{si } null(r_1) \\
 & = & first(r_1) \quad \text{sinon} \\
first(r_1 \,|\, r_2) & = & first(r_1) \cup first(r_2) \\
first(r\star) & = & first(r)
\end{array}
$$

La seule subtilité se situe dans la concaténation $r_1 r_2$. En effet, lorsque $null(r_1)$ est vrai, il faut incorporer aussi $first(r_2)$ dans le résultat. La définition de *last* est similaire et laissée en exercice.

Exercice 16. Donner la définition de *last*. Solution □

Il en découle enfin la définition de *follow*, toujours par récurrence structurelle :

$$
\begin{array}{rcl}
follow(c, \emptyset) & = & \emptyset \\
follow(c, \epsilon) & = & \emptyset \\
follow(c, a) & = & \emptyset \\
follow(c, r_1\, r_2) & = & follow(c, r_1) \cup follow(c, r_2) \cup first(r_2) \quad \text{si } c \in last(r_1) \\
 & = & follow(c, r_1) \cup follow(c, r_2) \quad \text{sinon} \\
follow(c, r_1 \,|\, r_2) & = & follow(c, r_1) \cup follow(c, r_2) \\
follow(c, r\star) & = & follow(c, r) \cup first(r) \quad \text{si } c \in last(r) \\
 & = & follow(c, r) \quad \text{sinon}
\end{array}
$$

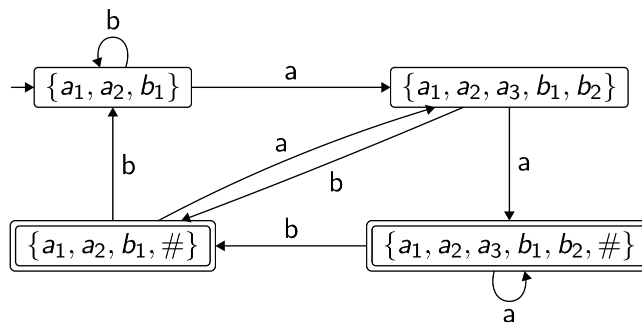
Là encore, la seule difficulté se trouve au niveau de la concaténation $r_1\, r_2$. En effet, le caractère c et celui qui le suit peuvent se retrouver à cheval sur r_1 et r_2, lorsque $c \in last(r_1)$, auquel cas $first(r_2)$ doit être inclus dans le résultat. Le même cas de figure peut se produire avec la concaténation de deux occurrence de r dans le cas de $r\star$.

Nous avons maintenant tout ce qu'il nous faut pour construire l'automate fini reconnaissant le langage de r. On commence par ajouter à la fin de r un caractère n'appartenant pas à l'alphabet A ; notons-le #. L'état initial de notre automate est l'ensemble $\mathit{first}(r\#)$. On construit ensuite les autres états et les transitions par nécessité, avec l'algorithme suivant :

tant qu'il existe un état s dont il faut calculer les transitions
  pour chaque caractère c de l'alphabet
    soit s' l'état $\bigcup_{c_i \in s} \mathit{follow}(c_i, r\#)$
    ajouter la transition $s \xrightarrow{c} s'$

Il est clair que cette construction termine, car les états possibles sont en nombre fini (ce sont des sous-ensembles de l'ensemble des caractères de l'expression régulière). Les états terminaux sont les états contenant le caractère #.

Sur l'exemple $(a|b) \star a(a|b)$, on obtient l'automate suivant :



On peut vérifier facilement qu'il reconnaît bien le langage des mots contenant un caractère a en avant-dernière position. Il se trouve que cet automate est minimal, au sens du nombre d'états, mais ce n'est pas nécessairement toujours le cas.

Exercice 17. Donner le résultat de cette construction sur l'expression régulière $(a|b) \star a$. Comparer avec l'automate donné page 47. Solution □

3.3 Analyseur lexical

Un *analyseur lexical* est un automate fini pour la « réunion » de toutes les expressions régulières définissant les lexèmes. Le fonctionnement d'un l'analyseur lexical, cependant, est différent de la simple reconnaissance d'un mot par un automate, car

— il faut décomposer un mot (le texte source) en une *suite* de mots reconnus ;
— il peut y avoir des *ambiguïtés* ;
— il faut construire les lexèmes, au lieu de simplement reconnaître des mots, *i.e.*, les états terminaux contiennent des *actions* à effectuer.

On a donné plus haut des exemples d'ambiguïté. Si par exemple on a écrit `funx` dans un langage où `fun` est un mot-clé, alors on pourrait reconnaître `funx` comme un identificateur mais aussi comme le mot-clé `fun` suivi de l'identificateur `x`. Pour lever cette ambiguïté, on fait en général le choix de reconnaître le lexème le plus long possible.

Une autre ambiguïté potentielle vient du fait que le mot `fun` est tout autant reconnu par l'expression régulière du mot-clé `fun` que par celle des identificateurs. Plutôt que de chercher à définir une expression régulière pour les identificateurs qui exclurait tous

les mots-clés, ce qui serait extrêmement pénible, on fait plutôt le choix de classer les expressions régulières par ordre de priorité. À longueur égale, c'est l'expression régulière la plus prioritaire qui définit l'action à effectuer.

Enfin, une particularité notable d'un analyseur lexical est qu'on ne cherche pas à faire de retour en arrière en cas d'échec. Ainsi, si a, ab et bc sont les trois lexèmes possibles de notre langage, notre analyseur lexical va *échouer* sur l'entrée

$$abc$$

alors même que ce texte est décomposable en deux lexèmes, à savoir a et bc. La raison de cet échec est qu'après avoir reconnu le lexème le plus long possible, à savoir ab, on ne cherche plus à revenir sur cette décision. On échoue donc sur l'entrée c, qui n'est pas dans le langage des lexèmes. C'est uniquement une considération pragmatique qui conduit à cette décision de ne pas faire de retour en arrière, pour de meilleures performances.

3.4 L'outil `ocamllex`

En pratique, on dispose d'outils qui construisent un analyseur lexical à partir de sa description par des expressions régulières. C'est la grande famille de l'outil `lex`, qui est décliné pour une majorité de langages : `flex` pour C, `jflex` pour Java, `ocamllex` pour OCaml, etc. On présente ici l'outil `ocamllex` mais le fonctionnement des autres outils de la famille `lex` est très similaire.

Un exemple de fichier `ocamllex` est donné figure 3.1 pour l'analyse lexicale d'un petit langage d'expressions arithmétiques avec des constantes littérales, l'addition, la multiplication et des parenthèses. Un tel fichier débute par un prélude de code OCaml arbitraire entre accolades. Ici, on y définit un type `token` pour les lexèmes. Vient ensuite la déclaration d'une fonction d'analyse lexicale `token`, dans une syntaxe propre à l'outil `ocamllex` de la forme

```
rule token = parse
| regexp1 { action1 }
| regexp2 { action2 }
| ...
```

Chaque `regexp`i est une expression régulière. Le mot-clé `parse` signifie que c'est le plus long préfixe possible de l'entrée reconnu par l'une des expressions régulières qui détermine la ligne sélectionnée [1]. À longueur égale, c'est l'expression régulière qui vient en premier qui est sélectionnée.

À chaque expression régulière `regexp`i est associée une action `action`i, qui est un morceau de code OCaml arbitraire. Dans notre exemple, ce code construit une valeur du type `token`. Par exemple, les deux lignes

```
  | ['0'-'9']+ as s
      { CONST (int_of_string s) }
```

reconnaissent les constantes littérales avec l'expression régulière `['0'-'9']+`, c'est-à-dire une répétition d'au moins un caractère parmi `'0'`,...,`'9'`. L'action associée renvoie un lexème `CONST` avec la valeur de la constante. Celle-ci est obtenue en récupérant la chaîne `s`

1. Un autre mot-clé, `shortest`, permet au contraire de sélectionner le plus court préfixe.

```
{
  type token =
    | CONST of int
    | PLUS
    | TIMES
    | LEFTPAR
    | RIGHTPAR
    | EOF
}

rule token = parse
  | [' ' '\t' '\n']+
      { token lexbuf }
  | ['0'-'9']+ as s
      { CONST (int_of_string s) }
  | '+'
      { PLUS }
  | '*'
      { TIMES }
  | '('
      { LEFTPAR }
  | ')'
      { RIGHTPAR }
  | eof
      { EOF }
  | _
      { failwith "lexical error" }
```

FIGURE 3.1 – Analyse lexicale d'expressions arithmétiques avec `ocamllex`.

reconnue par l'expression régulière avec `as s`. Les autres lexèmes sont traités de façon similaire. Le lexème `EOF` représente la fin de l'entrée et on utilise pour cela l'expression régulière particulière `eof`.

Outre la construction des lexèmes, il convient de traiter aussi deux autres cas de figure. D'une part, il faut ignorer les blancs. Pour cela, on utilise l'expression régulière `[' ' '\t' '\n']+` qui reconnaît une séquence de caractères espaces, tabulations et retours chariot. Pour ignorer ces caractères, on rappelle l'analyseur lexical `token` récursivement, en lui passant en argument une variable `lexbuf` qui est implicite dans le code `ocamllex`. Elle représente la structure de données sur laquelle on est en train de réaliser l'analyse lexicale (une chaîne, un fichier, etc.). D'autre part, il faut penser aux caractères illégaux. On reconnaît un caractère quelconque avec l'expression régulière `_` puis on signale l'erreur lexicale en levant une exception. En pratique, il faudrait faire un effort supplémentaire pour signaler la nature et la position de cette erreur lexicale.

Un tel source `ocamllex` est écrit dans un fichier portant le suffixe `.mll`, par exemple `lexer.mll`. On le compile avec l'outil `ocamllex`, comme ceci :

```
> ocamllex lexer.mll
```

Ceci a pour effet de produit un fichier OCaml `lexer.ml` qui contient la définition du type `token` écrite dans le prélude et une fonction `token` :

```
type token = ...
val token : Lexing.lexbuf -> token
```

La structure de données sur laquelle on réalise l'analyse lexicale a le type `Lexing.lexbuf`, provenant du module `Lexing` de la bibliothèque standard d'OCaml. Ce module fournit plusieurs moyens de construire une valeur de ce type, comme par exemple une fonction `from_channel` lorsqu'on souhaite analyser le contenu d'un fichier.

Raccourcis. Il est possible de définir des raccourcis pour des expressions régulières, avec la construction `let` d'`ocamllex`. Ainsi, on peut avantageusement écrire un analyseur lexical contenant des identificateurs, des constantes littérales entières et flottantes de la manière suivante :

```
  let letter = ['a'-'z' 'A'-'Z']
  let digit = ['0'-'9']
  let decimals = '.' digit*
  let exponent = ['e' 'E'] ['+' '-']?  digit+

  rule token = parse
    | letter (letter | digit | '_')*          { ... }
    | digit+                                  { ... }
    | digit+ (decimals | decimals? exponent)  { ... }
    | ...
```

Reconnaissance des mots-clés. Lorsque le langage analysé contient des mots-clés, on prend soin de les indiquer *avant* les identificateurs :

```
  rule token = parse
  | "fun"         { FUN     }
  | letter+ as s { IDENT s }
```

En effet, à longueur égale, la première règle qui s'applique sera utilisée, comme expliqué plus haut. Sur le mot `fun`, il faut construire le lexème `FUN` et non pas un identificateur. Si le langage contient de très nombreux mots-clés, on peut les regrouper dans une table et écrire alors plutôt quelque chose comme

```
  rule token = parse
  | letter+ as s { try Hashtbl.find table s with Not_found -> IDENT s }
```

En particulier, l'automate construit par `ocamllex` sera beaucoup plus petit.

Plusieurs analyseurs lexicaux. L'outil `ocamllex` permet de définir simultanément, de manière mutuellement récursive, plusieurs analyseurs lexicaux. Ainsi, on peut définir par exemple un analyseur `token` servant de point d'entrée principal et un autre analyseur `string` pour reconnaître les chaînes de caractères.

```
rule token = parse
| ...
| '"'    { string lexbuf }
and string = parse
| '"'    { ... }
| "\\n"  { ...; string lexbuf }
| _ as c { ...; string lexbuf }
| eof    { raise UnterminatedString }
```

Cela permet en particulier de traiter les séquences d'échappement (comme `\n`) de façon particulière ou encore d'échouer avec une erreur spécifique sur une chaîne non terminée. Si on avait choisi au contraire de reconnaître une chaîne avec une unique expression régulière, cela aurait été beaucoup moins facile à réaliser.

Un analyseur lexical secondaire permet également de reconnaître les commentaires imbriqués. Pour des commentaires de la forme `(*...*)`, comme en OCaml, on peut ainsi se donner les règles suivantes :

```
rule token = parse
  | "(*"  { comment lexbuf; token lexbuf }
  | ...

and comment = parse
  | "*)"  { () }
  | "(*"  { comment lexbuf; comment lexbuf }
  | _     { comment lexbuf }
  | eof   { raise UnterminatedComment }
```

Le principe est ici que l'analyseur `comment` reconnaît un commentaire jusqu'à son terme, sans rien renvoyer. Ainsi, il peut s'appeler récursivement pour reconnaître un commentaire imbriqué. Nul besoin de compteur ici ; c'est la pile d'appels qui compte pour nous.

Notes bibliographiques. Les langages réguliers font l'objet du premier chapitre du livre d'Olivier Carton [5]. La construction de l'automate fini à partir de l'expression régulière présentée dans ce chapitre est due à Berry et Sethi [3].

4 Analyse syntaxique

L'objectif de l'analyse syntaxique est de reconnaître les phrases appartenant à la syntaxe du langage. Son entrée est le flot des lexèmes construits par l'analyse lexicale (chapitre précédent) et sa sortie est un arbre de syntaxe abstraite (section 2.1). En particulier, l'analyse syntaxique doit détecter les erreurs de syntaxe et les signaler précisément.

Pour réaliser l'analyse syntaxique, on va utiliser des outils analogues aux expressions régulières et automates finis utilisés pour l'analyse lexicale, à savoir des *grammaires non contextuelles* pour décrire la syntaxe (section 4.2, des *automates à pile* pour les reconnaître (sections 4.3 et 4.4) et des outils pour automatiser tout cela (section 4.5). On va néanmoins commencer ce chapitre par quelque chose de plus élémentaire.

4.1 Analyse syntaxique élémentaire

Avant de décrire les techniques et les outils d'analyse syntaxique, nous allons chercher à réaliser un analyseur syntaxique par des moyens élémentaires. Il y a en effet beaucoup à apprendre d'une telle tentative. Pour rester simple, mais pas trop simple, contentons-nous d'analyser des expressions arithmétiques écrites à partir de constantes littérales, d'additions, de multiplications et de parenthèses, avec la convention usuelle que la multiplication a priorité sur l'addition. En cas de succès, on souhaite construire l'arbre de syntaxe abstraite correspondant, dont le type est le suivant :

```
type expr =
| Const of int
| Add   of expr * expr
| Mul   of expr * expr
```

Ainsi, l'analyse d'une chaîne telle que `2 + 5 * (4 + 4)` doit réussir et donner l'arbre de syntaxe abstraite `Add (Const 2, Mul (Const 5, Add (Const 4, Const 4)))`. En revanche, l'analyse d'une chaîne comme `1+*2` ou encore comme `(3+4` doit signaler une erreur de syntaxe.

Le grand chercheur qu'était Gilles Kahn m'a dit un jour que pour écrire un analyseur syntaxique il fallait commencer par écrire une fonction d'impression. Suivons son conseil. Nous allons utiliser ici le module `Format` de la bibliothèque standard d'OCaml et écrire une fonction d'impression générale de type

```
val print: Format.formatter -> expr -> unit
```

Le premier argument contient toute la machinerie nécessaire à l'impression, telle que par exemple le fichier dans lequel on est en train d'écrire. Le module `Format` fournit notamment une fonction `fprintf` pour écrire à l'aide d'un tel `formatter`.

On peut bien sûr écrire une fonction d'impression très grossière qui parenthèse toute sous-expression pour éviter toute ambiguïté, de la manière suivante [1] :

```
let rec print fmt = function
  | Const n      -> fprintf fmt "%d" n
  | Add (e1, e2) -> fprintf fmt "(%a + %a)" print e1 print e2
  | Mul (e1, e2) -> fprintf fmt "(%a * %a)" print e1 print e2
```

C'est une solution correcte, mais plutôt décevante. Outre le fait qu'on utilise des parenthèses inutilement, on ne fait aucun effort pour insérer des retours chariot ou pour montrer la structure. Cherchons donc à écrire plutôt une fonction d'impression *élégante* (les anglo-saxons parlent de *pretty-printer*).

Pour ce qui est d'insérer des retours chariot et de montrer la structure, la bibliothèque `Format` nous offre tout ce qui est nécessaire. Elle définit en effet une notion de boîtes, au sens de la typographie, et diverses stratégies pour les combiner, ainsi qu'une notion d'espace sécable. On les utilise ainsi :

```
let rec print fmt = function
  | Const n      -> fprintf fmt "%d" n
  | Add (e1, e2) -> fprintf fmt "(@[%a +@ %a@])" print e1 print e2
  | Mul (e1, e2) -> fprintf fmt "(@[%a *@ %a@])" print e1 print e2
```

Les directives `@[` et `@]` ouvre et ferme une boîte respectivement et la direction `@ ` indique un espace sécable. On a choisi ici d'associer une boîte à chaque expression parenthésée. Si d'aventure une expression parenthésée ne tient pas sur la ligne, et qu'un retour chariot est inséré, alors l'impression se poursuivra sur la colonne qui suit la parenthèse ouvrante, de la manière suivante :

```
(2 + (3 * ((5 + 6) *
           (7 * 8))))
```

Bien sûr, c'est un choix tout personnel d'utilisation des boîtes. D'autres utilisations sont possibles, avec des résultats différents.

Reste le problème des parenthèses inutiles. Si on reprend l'exemple de l'expression `2 + 5 * (4 + 4)`, une seule paire de parenthèses est nécessaire pour son impression. Elle est rendue nécessaire par le fait qu'un argument de la multiplication est une addition, à savoir `4+4`, c'est-à-dire une opération de priorité inférieure. On pourrait être donc tenté d'écrire une fonction d'impression prenant un argument supplémentaire représentant la priorité de l'opération en cours d'impression. C'est une solution possible. Il y en a cependant une autre, équivalente mais plus élégante, consistant à écrire plusieurs fonctions d'impression, une pour chaque niveau de priorité. Ainsi, on peut écrire une fonction `print_expr` pour imprimer les expressions de plus faible priorité, c'est-à-dire les sommes, puis une fonction `print_term` pour imprimer les expressions de la priorité suivante, c'est-à-dire les produits,

1. La conversion `%a` de la bibliothèque `Format` permet de passer une fonction d'impression et un argument pour cette fonction. Ici, c'est notre propre fonction `print` que l'on passe récursivement.

puis enfin une troisième fonction `print_factor` pour imprimer les expressions de la plus forte priorité, c'est-à-dire les expressions parenthésées.

La première de ces fonctions imprime les sommes, c'est-à-dire les expressions de la forme `Add`. Elle le fait sans utiliser de parenthèses.

```
let rec print_expr fmt = function
  | Add (e1, e2) -> fprintf fmt "%a +@ %a" print_expr e1 print_expr e2
```

Si en revanche l'expression n'est pas de la forme `Add`, elle passe la main à la deuxième fonction, `print_term` :

```
  | e            -> print_term fmt e
```

On procède de même dans `print_term`, en affichant les expressions de la forme `Mul`, le cas échéant, et en passant sinon la main à la troisième fonction.

```
and print_term fmt = function
  | Mul (e1, e2) -> fprintf fmt "%a *@ %a" print_term e1 print_term e2
  | e            -> print_factor fmt e
```

La troisième fonction se charge du cas restant, à savoir les constantes littérales.

```
and print_factor fmt = function
  | Const n -> fprintf fmt "%d" n
```

Si en revanche l'expression n'est pas une constante, alors elle l'imprime entre parenthèses, avec `print_expr`.

```
  | e       -> fprintf fmt "(@[%a@])" print_expr e
```

L'intégralité du code est donné figure 4.1. L'effet obtenu est le bon, en particulier parce qu'un argument de `Mul` de type `Add` ne sera pas traité par `print_term` mais passé à `print_factor`, qui l'imprimera entre parenthèses. Sur l'exemple donné plus haut, on obtient `2 + 3 * (5 + 6) * 7 * 8`, ce qui est le résultat attendu.

Venons-en maintenant au problème proprement dit de l'analyse syntaxique. On suppose donné un analyseur lexical tel que celui présenté à la fin du chapitre précédent (figure 3.1 page 52), c'est-à-dire un type `token` pour les lexèmes et une fonction

```
val token: Lexing.lexbuf -> token
```

que l'on appelle chaque fois qu'on veut obtenir le lexème suivant. On se donne un peu de machinerie pour lire les lexèmes, avec une référence `tok` contenant le prochain lexème à examiner et une fonction `next` pour lire le lexème suivant.

```
let tok = ref EOF
let next () = tok := token lb
```

On suppose que `lb` représente ici la structure de type `Lexing.lexbuf` sur laquelle l'analyse lexicale est effectuée (typiquement une chaîne ou un fichier). On commence par initialiser `tok` avec le tout premier lexème.

```
let () = next ()
```

Enfin, on se donne une fonction pour signaler toute erreur de syntaxe.

```
let error () = failwith "syntax error"
```

```
type expr =
  | Const of int
  | Add   of expr * expr
  | Mul   of expr * expr

open Format

let rec print_expr fmt = function
  | Add (e1, e2) -> fprintf fmt "%a +@ %a" print_expr e1 print_expr e2
  | e            -> print_term fmt e

and print_term fmt = function
  | Mul (e1, e2) -> fprintf fmt "%a *@ %a" print_term e1 print_term e2
  | e            -> print_factor fmt e

and print_factor fmt = function
  | Const n -> fprintf fmt "%d" n
  | e       -> fprintf fmt "(@[%a@])" print_expr e
```

FIGURE 4.1 – Fonction d'impression élégante d'expressions arithmétiques.

La fonction d'impression que nous venons d'écrire nous a mis sur la bonne voie en distinguant les sommes, les produits et les facteurs. Écrivons donc trois fonctions d'analyse syntaxique `parse_expr`, `parse_term` et `parse_factor` sur la même idée. Ces trois fonctions ont le type `unit -> expr`. Notre invariant est que chacune de ces trois fonctions consomme tous les lexèmes qui composent l'expression reconnue. La fonction `parse_expr` doit reconnaître une somme. Elle commence par reconnaître un premier terme en appelant la fonction `parse_term`.

```
  let rec parse_expr () =
    let e = parse_term () in
```

Puis elle examine le lexème suivant. S'il s'agit de `PLUS`, c'est-à-dire du symbole `+`, alors on le consomme avec `next` et on poursuit la lecture d'une somme avec un autre appel à `parse_expr`. Sinon, la lecture de la somme est terminée et on renvoie `e`.

```
    if !tok = PLUS then begin next (); Add (e, parse_expr ()) end else e
```

La fonction `parse_term` procède de façon similaire, en reconnaissant un produit de facteurs séparés par le lexème `TIMES`.

```
  and parse_term () =
    let e = parse_factor () in
    if !tok = TIMES then begin next (); Mul (e, parse_term ()) end else e
```

Enfin, la fonction `parse_factor` doit reconnaître les constantes littérales et les expressions parenthésées. Il n'y a aucune difficulté pour les premières. Il faut juste ne pas oublier de consommer le lexème.

```
  and parse_factor () = match !tok with
    | CONST n -> next (); Const n
```

Pour les expressions parenthésées, c'est plus subtil. On commence par consommer le lexème correspondant à la parenthèse ouvrante puis on reconnaît une expression avec `parse_expr`.

```
    | LEFTPAR ->
        next ();
        let e = parse_expr () in
```

Ensuite, il convient de vérifier qu'on trouve bien une parenthèse fermante juste derrière l'expression `e`. Si ce n'est pas le cas, on signale une erreur de syntaxe. Sinon, on consomme la parenthèse fermante et on renvoie `e`.

```
        if !tok <> RIGHTPAR then error ();
        next (); e
```

Si le premier lexème n'est ni une constante, ni une parenthèse ouvrante, alors on signale une erreur de syntaxe.

```
    | _ ->
        error ()
```

Pour reconnaître l'expression toute entière, il suffit d'appeler la fonction `parse_expr`. En effet, si l'expression n'est pas une somme, elle se réduira à un unique terme reconnu par `parse_term`. Si de même ce terme n'est pas un produit, il se réduira à un unique facteur reconnu par `parse_factor`. Il faut cependant penser à vérifier que toute l'entrée a bien été consommée. On le fait de la manière suivante :

```
  let e = parse_expr ()
  let () = if !tok <> EOF then error ()
```

Sans cette dernière vérification, une entrée comme `1+2 3` serait acceptée. L'intégralité du code est donné figure 4.2.

4.2 Grammaire

On a déjà utilisé informellement la notion de grammaire dans la section 2.1, pour définir la syntaxe abstraite. On donne maintenant une définition formelle de ce qu'est une grammaire.

Définition 8 (grammaire)**.** Une grammaire non contextuelle (ou hors contexte) est un quadruplet (N, T, S, R) où

— N est un ensemble fini de *symboles non terminaux* ;
— T est un ensemble fini de *symboles terminaux* ;
— $S \in N$ est le symbole de départ (dit *axiome*) ;
— $R \subseteq N \times (N \cup T)^\star$ est un ensemble fini de *règles de production*. □

Exemple. Voici un exemple de grammaire très simple pour des expressions arithmétiques constituées de constantes entières, d'additions, de multiplications et de parenthèses.

$$N = \{E\}$$
$$T = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{int}\}$$
$$S = E$$
$$R = \{\, (E, E\texttt{+}E), (E, E\texttt{*}E), (E, \texttt{(}E\texttt{)}), (E, \texttt{int})\,\}$$

```
let lb = Lexing.from_channel stdin
let tok = ref EOF              (* le prochain lexème à examiner *)
let next () = tok := token lb
let () = next ()

let error () = failwith "syntax error"

let rec parse_expr () =
  let e = parse_term () in
  if !tok = PLUS then begin next (); Add (e, parse_expr ()) end else e
and parse_term () =
  let e = parse_factor () in
  if !tok = TIMES then begin next (); Mul (e, parse_term ()) end else e
and parse_factor () = match !tok with
  | CONST n ->
      next (); Const n
  | LEFTPAR ->
      next (); let e = parse_expr () in
      if !tok <> RIGHTPAR then error (); next (); e
  | _ ->
      error ()

let e = parse_expr ()
let () = if !tok <> EOF then error ()
```

FIGURE 4.2 – Analyse syntaxique élémentaire d’expressions arithmétiques.

En pratique, on note les règles sous une forme plus agréable à lire, comme ceci :

$$
\begin{array}{rcl}
E & \rightarrow & E \texttt{ + } E \\
 & | & E \texttt{ * } E \\
 & | & \texttt{(} E \texttt{)} \\
 & | & \texttt{int}
\end{array}
\tag{4.1}
$$

Dans notre contexte, les terminaux de la grammaire sont les lexèmes produits par l'analyse lexicale. Ici `int` désigne ici le lexème correspondant à une constante entière. □

Notre objectif est maintenant de définir le langage des mots acceptés par cette grammaire. Pour cela, on commence par introduire la notion de dérivation.

Définition 9 (dérivation)**.** Un mot $u \in (N \cup T)^\star$ se *dérive* en un mot $v \in (N \cup T)^\star$, et on note $u \rightarrow v$, s'il existe une décomposition

$$u = u_1 X u_2$$

avec $X \in N$, $X \rightarrow \beta \in R$ et

$$v = u_1 \beta u_2.$$

Une suite $w_1 \rightarrow w_2 \rightarrow \cdots \rightarrow w_n$ est appelée une dérivation. On parle de *dérivation gauche* (resp. *droite*) si le non terminal réduit est systématiquement le plus à gauche, *i.e.*, $u_1 \in T^\star$ (resp. le plus à droite, *i.e.*, $u_2 \in T^\star$). On note $\rightarrow^\star$ la clôture réflexive transitive de $\rightarrow$. □

Avec la grammaire ci-dessus, on a par exemple la dérivation gauche suivante :

$$
\begin{array}{rcl}
E & \rightarrow & E \texttt{ * } E \\
 & \rightarrow & \texttt{int * } E \\
 & \rightarrow & \texttt{int * (} E \texttt{)} \\
 & \rightarrow & \texttt{int * (} E \texttt{ + } E \texttt{)} \\
 & \rightarrow & \texttt{int * (int + } E \texttt{)} \\
 & \rightarrow & \texttt{int * (int + int)}
\end{array}
$$

Le langage défini par une grammaire vient alors sans surprise :

Définition 10. Le *langage* défini par une grammaire non contextuelle $G = (N, T, S, R)$ est l'ensemble des mots de $T^\star$ dérivés de l'axiome, *i.e.*,

$$L(G) = \{ w \in T^\star \mid S \rightarrow^\star w \}.$$

□

Toujours avec la même grammaire, on a donc établi plus haut que

$$\texttt{int * (int + int)} \in L(G).$$

Définition 11 (arbre de dérivation)**.** Pour une grammaire donnée, un *arbre de dérivation* est un arbre dont les nœuds sont étiquetés par des symboles de la grammaire, de la manière suivante :

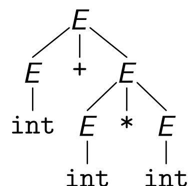
— la racine est l'axiome S ;
— tout nœud interne X est un non terminal dont les fils sont étiquetés par $\beta \in (N \cup T)^\star$ avec $X \rightarrow \beta$ une règle de la grammaire. □

Pour un arbre de dérivation dont les feuilles forment le mot w dans l'ordre infixe, il est clair qu'on a $S \rightarrow^\star w$. Inversement, à toute dérivation $S \rightarrow^\star w$, on peut associer un arbre de dérivation dont les feuilles forment le mot w dans l'ordre infixe (preuve par récurrence sur la longueur de la dérivation).

Exemple. À la dérivation gauche

$$E \rightarrow E + E \rightarrow \texttt{int} + E \rightarrow \texttt{int} + E * E \rightarrow \texttt{int} + \texttt{int} * E \rightarrow \texttt{int} + \texttt{int} * \texttt{int}$$

correspond l'arbre de dérivation suivant :



Mais cet arbre de dérivation correspond également à la dérivation droite

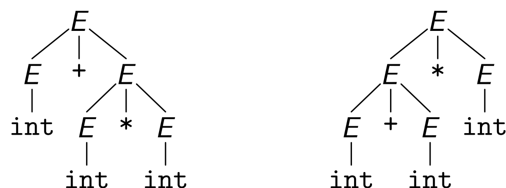
$$E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * \texttt{int} \rightarrow E + \texttt{int} * \texttt{int} \rightarrow \texttt{int} + \texttt{int} * \texttt{int}$$

ainsi qu'à d'autres dérivations encore qui ne sont ni gauche ni droite.

La notion d'arbre de dérivation nous permet d'identifier tout un ensemble de dérivations qui sont « moralement les mêmes ». Si en revanche un même mot admet plusieurs arbres de dérivation, c'est alors le symptôme d'une ambiguïté dans la grammaire.

Définition 12. (ambiguïté) Une grammaire est dite *ambiguë* si un mot au moins admet plusieurs arbres de dérivation.

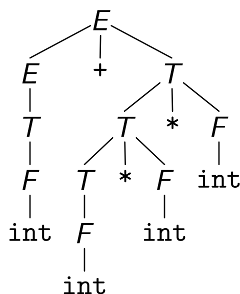
Exemple. La grammaire utilisée en exemple jusqu'à présent est ambiguë. En effet, le mot `int + int * int` admet les deux arbres de dérivations suivants :



Il est néanmoins possible de proposer une autre grammaire, non ambiguë, qui définit le même langage. Voici une solution (mais ce n'est pas la seule) :

$$\begin{array}{rcl} E & \rightarrow & E + T \\ & | & T \\ T & \rightarrow & T * F \\ & | & F \\ F & \rightarrow & (\ E\ ) \\ & | & \texttt{int} \end{array} \tag{4.2}$$

Cette nouvelle grammaire traduit la priorité de la multiplication sur l'addition et le choix d'une associativité à gauche pour ces deux opérations. Ainsi, le mot `int + int * int * int` n'a plus qu'un seul arbre de dérivation, à savoir



correspondant à la dérivation gauche

$$\begin{aligned} E &\to E + T \to T + T \to F + T \to \texttt{int} + T \to \texttt{int} + T * F \\ &\to \texttt{int} + T * F * F \to \texttt{int} + F * F * F \to \texttt{int} + \texttt{int} * F * F \\ &\to \texttt{int} + \texttt{int} * \texttt{int} * F \to \texttt{int} + \texttt{int} * \texttt{int} * \texttt{int} \end{aligned}$$

Exercice 18. Montrer que cette nouvelle grammaire reconnaît bien le même langage que la précédente. Solution □

Exercice 19. Voici une grammaire possible pour les termes du λ-calcul (en supposant que l'on utilise ce que l'on appelle des *indices de de Bruijn*) :

$$\begin{array}{rcl} T & \to & \texttt{nat} \\ & | & T\ T \\ & | & \texttt{lam}\ T \end{array}$$

Montrer que cette grammaire est ambiguë. Proposer une autre grammaire qui reconnaît le même langage et qui n'est pas ambiguë. Solution □

Malheureusement, déterminer si une grammaire est ou non ambiguë n'est *pas décidable*[2]. On va donc utiliser des *critères décidables suffisants* pour garantir qu'une grammaire est non ambiguë, pour lesquels on sait en outre décider l'appartenance au langage efficacement (avec un automate à pile déterministe). Les classes de grammaires définies par ces critères portent des noms étranges comme LL(1), LR(0), SLR(1), LALR(1) ou encore LR(1), que nous introduirons dans les sections suivantes. Mais pour cela, il nous faut quelques notions supplémentaires sur les grammaires.

La première de ces notions est la propriété pour un non terminal X de se dériver en le mot vide, c'est-à-dire $X \to^\star \epsilon$. On note cette propriété $\textsc{null}(X)$. On la définit de manière plus générale pour un mot quelconque.

Définition 13 (NULL)**.** Soit $\alpha \in (T \cup N)^\star$. La propriété $\textsc{null}(\alpha)$ est vraie si et seulement si on peut dériver ϵ à partir de α *i.e.* $\alpha \to^\star \epsilon$.

La deuxième notion caractérise les symboles terminaux qui peuvent apparaître en première position d'un mot dérivé par la grammaire.

Définition 14 (FIRST)**.** Soit $\alpha \in (T \cup N)^\star$. On définit $\textsc{first}(\alpha)$ comme l'ensemble de tous les premiers terminaux des mots dérivés de α, *i.e.* $\textsc{first}(\alpha) = \{a \in T \mid \exists w.\, \alpha \to^\star aw\}$.

Enfin, la troisième notion caractérise les symboles terminaux qui peuvent apparaître après un symbole non terminal dans une dérivation.

Définition 15 (FOLLOW)**.** Soit $X \in N$. On définit $\textsc{follow}(X)$ comme l'ensemble de tous les terminaux qui peuvent apparaître après X dans une dérivation d'un mot reconnu, *i.e.* $\textsc{follow}(X) = \{a \in T \mid \exists u, w.\, S \to^\star uXaw\}$.

Ces notions sont analogues de celles que nous avions introduites sur les expressions régulières page 48. Il reste à montrer comment les calculer.

2. On rappelle que « décidable » veut dire qu'on peut écrire un programme qui, pour toute entrée, termine et répond oui ou non.

Calcul de NULL, FIRST et FOLLOW. Pour calculer $\text{NULL}(\alpha)$, il suffit de déterminer $\text{NULL}(X)$ pour chaque $X \in N$. En effet, $\text{NULL}(X)$ est vrai si et seulement si il existe une production $X \rightarrow \epsilon$ dans la grammaire ou s'il existe une production $X \rightarrow Y_1 \dots Y_m$ avec $\text{NULL}(Y_i)$ pour tout i. Il s'agit donc d'un ensemble d'équations mutuellement récursives, qui définissent les valeurs des $\text{NULL}(X)$ en fonction d'elles-mêmes. Dit autrement, si $N = \{X_1, \dots, X_n\}$ et si on pose $\vec{V} = (\text{NULL}(X_1), \dots, \text{NULL}(X_n))$, on cherche la *plus petite solution* d'une équation de la forme

$$\vec{V} = F(\vec{V})$$

Fort heureusement, nous sommes dans un cas de figure où il existe un résultat qui va nous permettre de résoudre une telle équation facilement, à savoir une forme simple du théorème de Knaster–Tarski.

Théorème 1 (existence d'un plus petit point fixe)**.** Soit A un ensemble fini muni d'une relation d'ordre $\leq$ et d'un plus petit élément ε. Toute fonction $f : A \rightarrow A$ croissante, *i.e.* telle que $\forall x, y.\, x \leq y \Rightarrow f(x) \leq f(y)$, admet *un plus petit point fixe*, c'est-à-dire qu'il existe $a_0 \in A$ tel que

$$f(a_0) = a_0$$

et tel que pour tout autre point fixe $b = f(b)$, on a $a_0 \leq b$.

PREUVE. Comme ε est le plus petit élément, on a $\varepsilon \leq f(\varepsilon)$. La fonction f étant croissante, on a donc $f^k(\varepsilon) \leq f^{k+1}(\varepsilon)$ pour tout k. L'ensemble A étant fini, il existe donc un plus petit k_0 tel que $f^{k_0}(\varepsilon) = f^{k_0+1}(\varepsilon)$. On vient de trouver un point fixe $a_0 = f^{k_0}(\varepsilon)$ de f.

Soit b un autre point fixe de f. On a $\varepsilon \leq b$ et donc $f^k(\varepsilon) \leq f^k(b)$ pour tout k. En particulier $a_0 = f^{k_0}(\varepsilon) \leq f^{k_0}(b) = b$. Le point fixe a_0 est donc le plus petit point fixe de f. □

Dans le cas du calcul de NULL, on a $A = \text{BOOL} \times \dots \times \text{BOOL}$ avec $\text{BOOL} = \{\texttt{false}, \texttt{true}\}$. On peut munir BOOL de l'ordre $\texttt{false} \leq \texttt{true}$ et A de l'ordre point à point, c'est-à-dire

$$(x_1, \dots, x_n) \leq (y_1, \dots, y_n) \quad \text{si et seulement si} \quad \forall i.\, x_i \leq y_i.$$

Le théorème s'applique alors en prenant

$$\varepsilon = (\texttt{false}, \dots, \texttt{false})$$

car la fonction calculant $\text{NULL}(X)$ à partir des $\text{NULL}(X_i)$ est croissante. Dit autrement, il suffit de considérer initialement que tous les $\text{NULL}(X)$ sont faux, puis d'appliquer la fonction de calcul de NULL jusqu'à ce que la valeur des $\text{NULL}(X)$ ne bouge plus. Prenons l'exemple de cette grammaire :

$$\begin{array}{rcl} E & \rightarrow & T\,E' \\ E' & \rightarrow & \texttt{+}\;T\,E' \\ & | & \epsilon \\ T & \rightarrow & F\,T' \\ T' & \rightarrow & \texttt{*}\;F\,T' \\ & | & \epsilon \\ F & \rightarrow & \texttt{(}\;E\;\texttt{)} \\ & | & \texttt{int} \end{array} \tag{4.3}$$

Il s'agit d'une troisième variante de la grammaire des expressions arithmétiques. Le calcul va converger en deux étapes, de la manière suivante :

E	E'	T	T'	F
`false`	`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`false`

On en déduit que le mot vide peut être dérivé des symboles non terminaux E' et T' mais pas des symboles E, T et F. Ce n'était pas totalement évident *a priori*, car la règle $E \to T\,E'$ aurait pu conduire à $\textsc{null}(E)$ si on avait eu aussi $\textsc{null}(T)$.

Exercice 20. Justifier que l'on cherche un *plus petit* point fixe pour calculer les valeurs de NULL. Solution □

Venons-en maintenant au calcul de FIRST. Pour calculer $\textsc{first}(\alpha)$ pour un mot quelconque, il suffit de savoir calculer $\textsc{first}(X)$ pour chaque non terminal X. En effet, on a

$$\begin{array}{rcl}\textsc{first}(\epsilon) & = & \emptyset \\ \textsc{first}(a\beta) & = & \{a\}, \quad \text{si } a \in T \\ \textsc{first}(X\beta) & = & \textsc{first}(X), \quad \text{si } \neg\textsc{null}(X) \\ \textsc{first}(X\beta) & = & \textsc{first}(X) \cup \textsc{first}(\beta), \quad \text{si } \textsc{null}(X)\end{array}$$

Et pour calculer $\textsc{first}(X)$, il suffit de considérer toutes les productions pour X dans la grammaire :

$$\textsc{first}(X) = \bigcup_{X \to \beta} \textsc{first}(\beta)$$

On se retrouve donc de nouveau avec des équations récursives définissant les ensembles $\textsc{first}(X)$. On peut là encore se servir du théorème de Knaster–Tarski, cette fois sur le produit cartésien $A = \mathcal{P}(T) \times \cdots \times \mathcal{P}(T)$ muni, point à point, de l'inclusion comme relation d'ordre. Le plus petit élément est $\varepsilon = (\emptyset, \ldots, \emptyset)$.

Si on reprend l'exemple de la grammaire (4.3), on converge en quatre étapes, de la manière suivante :

E	E'	T	T'	F
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	{`+`}	$\emptyset$	{`*`}	{`(`, `int`}
$\emptyset$	{`+`}	{`(`, `int`}	{`*`}	{`(`, `int`}
{`(`, `int`}	{`+`}	{`(`, `int`}	{`*`}	{`(`, `int`}
{`(`, `int`}	{`+`}	{`(`, `int`}	{`*`}	{`(`, `int`}

Il nous reste à calculer FOLLOW. De façon évidente, l'ensemble $\textsc{first}(\beta)$ fait partie de $\textsc{follow}(X)$ si on a une production de la forme $Y \to \alpha X \beta$. Plus subtilement, si $\textsc{null}(\beta)$, il faut aussi ajouter à $\textsc{follow}(X)$ tous les éléments de $\textsc{follow}(Y)$. On a donc

$$\textsc{follow}(X) = \bigcup_{Y \to \alpha X \beta} \textsc{first}(\beta) \;\cup \bigcup_{Y \to \alpha X \beta,\, \textsc{null}(\beta)} \textsc{follow}(Y)$$

Là encore, il s'agit d'équations mutuellement récursives pour lesquelles on va procéder à un calcul de point fixe, sur le même domaine que pour FIRST. En pratique, il nous sera utile d'ajouter un symbole terminal particulier, noté #, pour représenter la fin de l'entrée. On peut l'ajouter directement dans $\textsc{follow}(S)$. Si on reprend l'exemple de la grammaire (4.3), on converge en quatre étapes, de la manière suivante :

E	E'	T	T'	F
$\{\#\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{\#,)\}$	$\{\#\}$	$\{+, \#\}$	$\emptyset$	$\{*\}$
$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#\}$	$\{*, +, \#\}$
$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#,)\}$	$\{*, +, \#,)\}$
$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#,)\}$	$\{*, +, \#,)\}$

Exercice 21. Voici une grammaire possible pour les termes du λ-calcul :

$$\begin{array}{rcl} S & \to & T\ \# \\ T & \to & \texttt{nat} \\ & | & \texttt{(}\ T\ T\ \texttt{)} \\ & | & \texttt{lam}\ T \end{array} \tag{4.4}$$

Calculer NULL, FIRST et FOLLOW pour cette grammaire. Solution □

Exercice 22. Voici la grammaire du langage LISP :

$$\begin{array}{rcl} S & \to & E\ \# \\ E & \to & \texttt{sym} \\ & | & \texttt{(}\ L\ \texttt{)} \\ L & \to & \epsilon \\ & | & E\ L \end{array} \tag{4.5}$$

Calculer NULL, FIRST et FOLLOW pour cette grammaire. Solution □

4.3 Analyse descendante

L'idée de l'*analyse descendante* (en anglais *top-down parsing*) consiste à procéder par expansions successives du non terminal le plus à gauche. On construit donc une dérivation gauche. On part du symbole de départ S et on se sert d'une *table* indiquant, pour un non terminal X à expanser et les k premiers caractères de l'entrée, l'expansion $X \to \beta$ à effectuer. Supposons $k = 1$ par la suite et notons $T(X, c)$ cette table, pour un non terminal X et un terminal c donnés. On suppose aussi qu'un symbole terminal # dénote la fin de l'entrée et que la table T indique donc également l'expansion à effectuer pour ce caractère.

L'analyse descendante utilise une pile, qui est un mot de $(N \cup T)^\star$. Initialement la pile est réduite au symbole de départ S. À chaque instant, on examine le sommet de la pile et le premier caractère c de l'entrée, en effectuant les actions suivantes :

- si la pile est vide, on s'arrête ; il y a succès si et seulement si c est #.
- si le sommet de la pile est un terminal a, alors a doit être égal à c, on dépile a et on consomme c ; sinon on échoue.
- si le sommet de la pile est un non terminal X, alors on remplace X par le mot $\beta = T(X, c)$ en sommet de pile, le cas échéant, en empilant les caractères de β en partant du dernier ; sinon, on échoue.

Illustrons ce fonctionnement sur un exemple. On reprend l'exemple de la grammaire (4.3) pour les expressions arithmétiques et on se donne une table d'expansion (nous expliquerons plus loin comment la construire).

pile	entrée
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T$`+`	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F$`*`	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	`#`
E'	`#`
ϵ	`#`

FIGURE 4.3 – Analyse descendante du mot `int+int*int`.

$$
\begin{array}{rcl}
E & \rightarrow & T\,E' \\
E' & \rightarrow & +\,T\,E' \\
 & | & \epsilon \\
T & \rightarrow & F\,T' \\
T' & \rightarrow & *\,F\,T' \\
 & | & \epsilon \\
F & \rightarrow & (\,E\,) \\
 & | & \texttt{int}
\end{array}
$$

	`+`	`*`	`(`	`)`	`int`	`#`
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		`int`	

Analysons le mot `int+int*int` en nous servant de cette table. Les différentes étapes sont illustrées figure 4.3. Initialement, la pile contient le symbole de départ E et l'entrée contient le mot à analyser suivi du terminal #. Notre première décision implique le symbole E au sommet de la pile et le symbole `int` au début de l'entrée. Comme E est un non terminal, on consulte la table, qui indique de faire l'expansion $E \rightarrow TE'$. On dépile donc le symbole E et on empile TE', en commençant par la fin. C'est donc T qui se retrouve en sommet de pile. On consulte à nouveau ta table, cette fois avec T et `int`. Elle indique de procéder à l'expansion $T \rightarrow FT'$. On dépile donc T pour empiler T' puis F. Une troisième consultation de la table indique l'expansion $F \rightarrow$ `int`. On dépile donc F pour empiler `int`. Cette fois, on se retrouve avec un symbole non terminal, `int`, en sommet de pile. On vérifie qu'il coïncide avec le début de l'entrée et, puisque c'est le cas, les deux sont supprimés. Le sommet de pile devient T' et le début de l'entrée devient `+`. La table indique de faire l'expansion $T' \rightarrow \epsilon$, ce qui a pour effet de dépiler T. Et ainsi de suite. On se retrouve au final avec une pile vide et une entrée réduite à #, ce qui achève notre analyse sur un succès.

Un analyseur descendant se programme très facilement en introduisant une fonction pour chaque non terminal de la grammaire. Chaque fonction examine l'entrée et, selon

le cas, la consomme ou appelle récursivement les fonctions correspondant à d'autres non terminaux, selon la table d'expansion. Sur notre exemple, nous aurions cinq fonctions E, E', T, T' et F mutuellement récursives. Sans le savoir, nous avions réalisé dans la section 4.1 une analyse descendante pour cette grammaire, avec seulement trois fonctions `parse_expr`, `parse_term` et `parse_factor` correspondant à E, T et F. Les fonctions correspondant à E' et T' étaient directement expansées dans les fonctions `parse_expr` et `parse_term`. La table d'expansion n'était pas explicite : elle était dans le code de chaque fonction. La pile non plus n'était pas explicite : elle était réalisée par la pile d'appels. Bien sûr, on aurait pu les rendre explicites.

Construction de la table d'expansion. Reste à expliquer comment construire la table d'expansion à partir de la grammaire. L'idée est simple : pour décider si on réalise l'expansion $X \rightarrow \beta$ lorsque le premier caractère de l'entrée est c, on détermine si c fait partie des *premiers* caractères des mots reconnus par β, c'est-à-dire si $c \in \text{FIRST}(\beta)$. Une difficulté se pose pour une production telle que $X \rightarrow \epsilon$. Il faut alors considérer aussi l'ensemble des caractères qui peuvent *suivre* X, c'est-à-dire $\text{FOLLOW}(X)$. On définit donc la table T de la manière suivante : pour chaque production $X \rightarrow \beta$,

- on pose $T(X, a) = \beta$ pour tout $a \in \text{FIRST}(\beta)$;
- si $\text{NULL}(\beta)$, on pose aussi $T(X, a) = \beta$ pour tout $a \in \text{FOLLOW}(X)$.

Exercice 23. Calculer la table d'expansion de la grammaire (4.3) et vérifier qu'on retrouve bien la table donnée page 67. □

Bien entendu, rien ne nous empêche de nous retrouver avec une table d'expansion contenant plusieurs expansions différentes dans une même case. Ce n'est pas nécessairement le symptôme d'une grammaire ambiguë, mais seulement d'une grammaire qui ne se prête pas à une telle analyse descendante. C'est pourquoi on introduit la définition suivante.

Définition 16 (grammaire LL(1))**.** Une grammaire est dite LL(1) si, dans la table précédente, il y a au plus une production dans chaque case. □

Dans cet acronyme, LL signifie « Left to right scanning, Leftmost derivation », ce qui signifie que l'entrée est analysée de la gauche vers la droite et que l'on construit une dérivation gauche. Le chiffre 1 signifie que l'on examine uniquement un caractère au début de l'entrée.

Exercice 24. La grammaire du λ-calcul de l'exercice 21 est-elle LL(1) ? Solution □

Exercice 25. La grammaire de LISP de l'exercice 22 est-elle LL(1) ? Solution □

Il faut souvent transformer les grammaires pour les rendre LL(1). En particulier, une grammaire *récursive gauche*, *i.e.* contenant une production de la forme $X \rightarrow X\alpha$ ne sera jamais LL(1). Il faut alors supprimer la récursion gauche (directe ou indirecte). De même, il faut factoriser les productions qui commencent par le même terminal (factorisation gauche).

En conclusion, les analyseurs LL(1) sont relativement simples à écrire mais ils nécessitent d'écrire des grammaires peu naturelles. On va se tourner vers une autre solution.

4.4 Analyse ascendante

L'idée est toujours de lire l'entrée de gauche à droite, mais on cherche maintenant à reconnaître des membres droits de productions pour construire l'arbre de dérivation de bas en haut, d'où le nom d'analyse ascendante (en anglais *bottom-up parsing*).

Comme l'analyse descendante, l'analyse ascendante lit l'entrée de la gauche vers la droite et manipule une pile qui est un mot de $(T \cup N)^\star$. À chaque instant, deux actions sont possibles :

— une opération de *lecture* (*shift* en anglais), qui consiste à lire le premier terminal de l'entrée et à l'empiler ;

— une opération de *réduction* (*reduce* en anglais), qui consiste à reconnaître en sommet de pile le membre droit β d'une production $X \to \beta$ et à remplacer β par X en sommet de pile.

Dans l'état initial, la pile est vide. Lorsqu'il n'y a plus d'action possible, l'entrée est reconnue si elle a été entièrement lue et si la pile est réduite à l'axiome S.

On illustre à droite un exemple d'analyse ascendante avec la grammaire

$$\begin{array}{lcl} E & \to & E + T \\ & | & T \\ T & \to & T * F \\ & | & F \\ F & \to & (\ E\ ) \\ & | & \texttt{int} \end{array}$$

et le mot `int+int*int`. La colonne de droite indique l'opération qui est effectuée à chaque étape. On a ici une séquence d'opérations qui mène à un succès et le mot est donc reconnu par la grammaire.

pile	entrée	action
ϵ	`int+int*int`	lecture
`int`	`+int*int`	réduction $F \to$ `int`
F	`+int*int`	réduction $T \to F$
T	`+int*int`	réduction $E \to T$
E	`+int*int`	lecture
E+	`int*int`	lecture
E+`int`	`*int`	réduction $F \to$ `int`
E+F	`*int`	réduction $T \to F$
E+T	`*int`	lecture
E+T*	`int`	lecture
E+T*`int`		réduction $F \to$ `int`
E+T*F		réduction $T \to T$*F
E+T		réduction $E \to E$+T
E		succès

Bien sûr, on aurait pu prendre d'autres décisions, comme par exemple effectuer une seconde opération de lecture après la toute première et échouer alors dans l'analyse. Afin de pouvoir utiliser en pratique l'analyse ascendante, il nous faut un moyen de mécaniser cette décision, c'est-à-dire un moyen de choisir à chaque étape entre lecture et réduction. Comme pour l'analyse descendante, nous allons prendre notre décision en examinant uniquement le premier caractère de l'entrée (mais l'idée se généralise à un nombre arbitraire de caractères).

L'idée consiste à construire un automate fini à partir de la grammaire et à intercaler sur la pile de l'analyse ascendante des états de cet automate entre les différents symboles. La pile prend donc la forme

$$s_0\, x_1\, s_1\ \ldots\ x_n\, s_n$$

où s_i est un état de l'automate et $x_i \in T \cup N$ comme auparavant. On se donne également une table d'actions indexée par un état de l'automate et un caractère. À chaque étape, on considère le premier caractère a de l'entrée et l'action indiquée par la table pour ce caractère et l'état s_n situé en sommet de pile.

— si la table indique un succès ou un échec, on s'arrête ;

— si la table indique une lecture, alors il doit exister une transition $s_n \xrightarrow{a} s$ dans l'automate et on empile successivement le non terminal a et l'état s ;
— si la table indique une réduction $X \to \alpha$, avec α de longueur p, alors on doit trouver α en sommet de pile

$$s_0\, x_1\, s_1\ \ldots\ x_{n-p}\, s_{n-p}\, |\alpha_1\, s_{n-p+1} \ldots\ \alpha_p\, s_n.$$

Par ailleurs, il doit exister une transition $s_{n-p} \xrightarrow{X} s$ dans l'automate. On dépile alors α et on empile $X\, s$, la pile devenant donc

$$s_0\, x_1\, s_1\ \ldots\ x_{n-p}\, s_{n-p}\, X\, s.$$

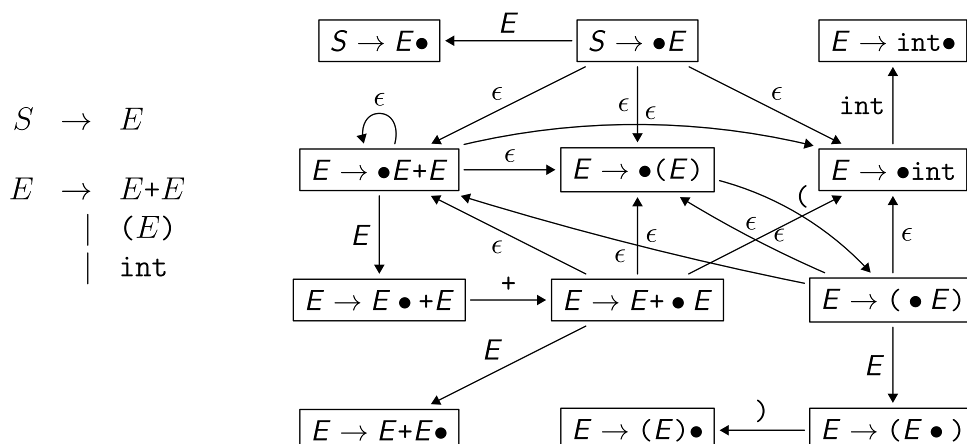
Montrons maintenant comment construire l'automate et la table. Pour expliquer le processus, on commence par construire un automate *asynchrone*, c'est-à-dire un automate contenant des transitions spontanées appelées *ϵ-transitions* et notées $s_1 \xrightarrow{\epsilon} s_2$. Une telle transition signifie que l'on peut passer de l'état s_1 à l'état s_2 spontanément, sans lire de caractère. Les *états* de cet automate asynchrone ont la forme

$$[X \to \alpha \bullet \beta]$$

où $X \to \alpha\beta$ est une production de la grammaire. Un tel état est appelé un *item*. L'intuition d'un tel état est « je cherche à reconnaître X, j'ai déjà lu α et je dois encore lire β ». Les *transitions* de l'automate sont étiquetées par $T \cup N$ et sont de trois formes :

$$\begin{array}{rcll} [Y \to \alpha \bullet a\beta] & \xrightarrow{a} & [Y \to \alpha a \bullet \beta] & \text{pour}\ \ a \in T, \\ [Y \to \alpha \bullet X\beta] & \xrightarrow{X} & [Y \to \alpha X \bullet \beta] & \text{pour}\ \ X \in N, \\ [Y \to \alpha \bullet X\beta] & \xrightarrow{\epsilon} & [X \to \bullet\gamma] & \text{pour toute production}\ X \to \gamma. \end{array}$$

On démarre la construction avec les états $[S \to \bullet\beta]$ pour toute production $S \to \beta$ de la grammaire. Voici un exemple sur une grammaire très simple (et ambiguë) pour les expressions arithmétiques :



L'étape suivante consiste à *déterminiser* cet automate. Pour cela, on regroupe les états reliés (transitivement) par des ϵ-transitions et les états deviennent donc des ensembles d'*items*. Ainsi, l'état $[S \to \bullet E]$, duquel sortaient trois transitions spontanées, devient l'ensemble suivant de quatre *items* :

$$\begin{array}{|l|}\hline S \to \bullet E \\ E \to \bullet E\texttt{+}E \\ E \to \bullet\texttt{(}E\texttt{)} \\ E \to \bullet\texttt{int} \\ \hline\end{array}$$

D'une manière générale, on obtient les états de l'automate déterministe en les saturant avec les ϵ-transitions dans le sens suivant : pour un état s de l'automate déterministe et un item i appartenant à s, si i est relié par ϵ à un item j dans l'automate non-déterministe, alors j appartient également à s.

Au final, on obtient l'automate fini déterministe donné au milieu de la figure 4.4. (On a ajouté à la fin de la production $S \to E$ un nouveau symbole terminal # pour représenter la fin de l'entrée, comme pour l'analyse descendante.) On peut maintenant construire la table d'actions à partir de cet automate. En pratique, on travaille avec deux tables. On a d'une part une table d'*actions* ayant pour lignes les états et pour colonnes les terminaux, la case $\texttt{action}(s, a)$ indiquant

— `shift` s' pour une lecture et un nouvel état s' ;
— `reduce` $X \to \alpha$ pour une réduction ;
— un succès ;
— un échec.

On a d'autre part une table de *déplacements* ayant pour lignes les états et pour colonnes les non terminaux, la case $\texttt{goto}(s, X)$ indiquant l'état résultat d'une réduction de X. On construit ainsi ces deux tables. Pour la table `action`, on pose

— $\texttt{action}(s, \#)$ = succès si $[S \to E \bullet \#] \in s$;
— $\texttt{action}(s, a) = \texttt{shift}\ s'$ si on a une transition $s \xrightarrow{a} s'$;
— $\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ si $[X \to \beta\bullet] \in s$, pour tout a ;
— échec dans tous les autres cas.

Pour la table `goto`, on pose

— $\texttt{goto}(s, X) = s'$ si et seulement si on a une transition $s \xrightarrow{X} s'$.

Sur notre exemple, on obtient la table donnée figure 4.4 (en bas). La table ainsi construite peut contenir plusieurs actions possibles dans une même case. On appelle cela un *conflit*. Il y a deux sortes de conflits :

— un conflit *lecture/réduction* (en anglais *shift/reduce*), si dans un état s on peut effectuer une lecture mais aussi une réduction ;
— un conflit *réduction/réduction* (en anglais *reduce/reduce*), si dans un état s deux réductions différentes sont possibles.

Lorsqu'il n'y a pas de conflit, on dit que la grammaire est LR(0).

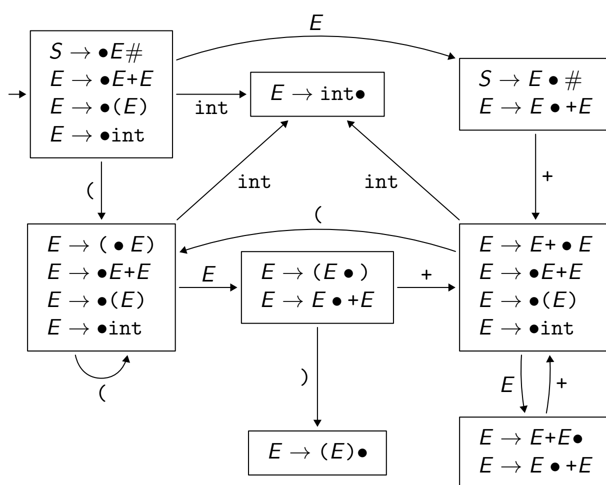
Définition 17 (grammaire LR(0))**.** Une grammaire est dite LR(0) si la table ainsi construite ne contient pas de conflit. (L'acronyme LR signifie « Left to right scanning, Rightmost derivation ».)

Dans notre exemple, il y a un conflit dans l'état 8 (dernière ligne). Si le caractère en entrée est +, on peut tout autant lire ce caractère que réduire la production $E \to E + E$. L'état 8 est l'état en bas à droite de l'automate, *i.e.* celui qui contient à la fois $E \to E\texttt{+}E\bullet$ et $E \to E \bullet \texttt{+}E$. Ce conflit illustre précisément l'ambiguïté de la grammaire sur un mot tel que `int+int+int`, après avoir lu `int+int`. On peut résoudre le conflit de deux façons : soit on favorise la *lecture*, en traduisant alors une associativité à droite ; soit on favorise la *réduction*, en traduisant alors une associativité à gauche. La figure 4.5 illustre

une grammaire

$$
\begin{array}{rcl}
S & \rightarrow & E \\
\\
E & \rightarrow & E\texttt{+}E \\
 & | & \texttt{(}E\texttt{)} \\
 & | & \texttt{int}
\end{array}
$$

son automate LR(0)



sa table LR(0)

	action					*goto*
état	`(`	`)`	`+`	`int`	#	E
1	`shift 4`			`shift 2`		3
2	`reduce` $E \rightarrow$ `int`					
3			`shift 6`		succès	
4	`shift 4`			`shift 2`		5
5		`shift 7`	`shift 6`			
6	`shift 4`			`shift 2`		8
7	`reduce` $E \rightarrow$ `(`E`)`					
8			`shift 6`			
	`reduce` $E \rightarrow E$`+`E					

FIGURE 4.4 – Analyse LR(0).

pile	entrée	action
1	`int+int+int`	s2
1 `int` 2	`+int+int`	$E \to$ `int`, g3
1 E 3	`+int+int`	s6
1 E 3 + 6	`int+int`	s2
1 E 3 + 6 `int` 2	`+int`	$E \to$ `int`, g8
1 E 3 + 6 E 8	`+int`	$E \to E$+E, g3
1 E 3	`+int`	s6
1 E 3 + 6	`int`	s2
1 E 3 + 6 `int` 2	#	$E \to$ `int`, g8
1 E 3 + 6 E 8	#	$E \to E$+E, g3
1 E 3	#	succès

FIGURE 4.5 – Analyse du mot `int+int+int`.

le fonctionnement de l'analyse sur le mot `int+int+int` en supposant qu'on a choisi cette seconde option.

Exercice 26. Montrer que la grammaire du λ-calcul, donnée dans l'exercice 21 page 66, est LR(0). Solution □

Analyse SLR(1). La construction LR(0) engendre très facilement des conflits. On va donc chercher à limiter les réductions. Une idée très simple consiste à poser `action`$(s, a) =$ `reduce` $X \to \beta$ si et seulement si

$$[X \to \beta\bullet] \in s \quad \text{et} \quad a \in \text{FOLLOW}(X).$$

Définition 18 (classe SLR(1))**.** Une grammaire est dite SLR(1) si la table ainsi construite ne contient pas de conflit. (SLR signifie en anglais *Simple LR.*) □

Exercice 27. Montrer que la grammaire

$$\begin{array}{rcl} S & \to & E\# \\ E & \to & E + T \\ & | & T \\ T & \to & T * F \\ & | & F \\ F & \to & (\ E\ ) \\ & | & \texttt{int} \end{array}$$

est SLR(1). (L'automate contient 12 états.) Solution □

Exercice 28. La grammaire de LISP, donnée dans l'exercice 22 page 66, est-elle SLR(1) ? Solution □

Exercice 29. Montrer que la grammaire

$$\begin{array}{rcl} S & \rightarrow & E\# \\ E & \rightarrow & G = D \\ & | & D \\ G & \rightarrow & * D \\ & | & \texttt{id} \\ D & \rightarrow & G \end{array}$$

n'est pas SLR(1). Solution □

Analyse LR(1). En pratique, la classe SLR(1) reste trop restrictive, comme le montre l'exemple de l'exercice 29. C'est là un exemple réaliste, issue de la grammaire du langage C. On introduit donc une classe de grammaires encore plus large, LR(1), au prix de tables encore plus grandes. Les *items* ont maintenant la forme

$$[X \rightarrow \alpha \bullet \beta, a]$$

avec la signification « je cherche à reconnaître X, j'ai déjà lu α et je dois encore lire β puis vérifier que le caractère suivant est a ». Les transitions de l'automate LR(1) non déterministe sont

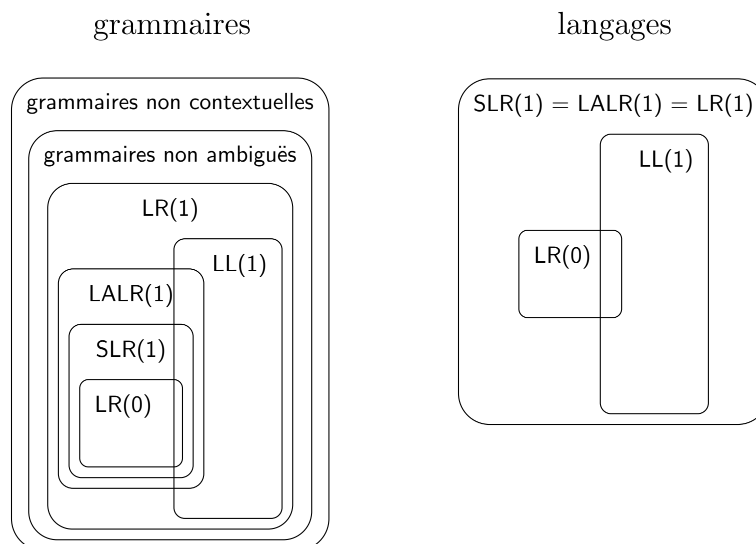
$$\begin{array}{rcll} [Y \rightarrow \alpha \bullet a\beta, b] & \xrightarrow{a} & [Y \rightarrow \alpha a \bullet \beta, b] & \text{pour} \quad a \in T \\ [Y \rightarrow \alpha \bullet X\beta, b] & \xrightarrow{X} & [Y \rightarrow \alpha X \bullet \beta, b] & \text{pour} \quad X \in N \\ [Y \rightarrow \alpha \bullet X\beta, b] & \xrightarrow{\epsilon} & [X \rightarrow \bullet\gamma, c] & \text{pour} \quad Y \rightarrow \beta \in R \text{ et } c \in \textsc{first}(\beta b) \end{array}$$

L'état initial est celui qui contient $[S \rightarrow \bullet\alpha, \#]$. Comme précédemment, on peut déterminiser l'automate et construire la table correspondante. On introduit une action de réduction pour (s, a) seulement lorsque s contient un item de la forme $[X \rightarrow \alpha\bullet, a]$.

Définition 19 (classe LR(1))**.** Une grammaire est dite LR(1) si la table ainsi construite ne contient pas de conflit.

Exercice 30. Montrer que la grammaire de l'exercice 29 est LR(1). Solution □

Autres classes. La construction LR(1) pouvant être coûteuse, il existe des approximations. La classe LALR(1) (pour *lookahead LR*) est une telle approximation, utilisée notamment dans les outils de la famille `yacc` dont nous parlerons dans la section suivante. On peut chercher à comparer les différentes classes de grammaires que nous avons introduites jusqu'ici. On peut le faire également en interprétant une classe de grammaires comme une classe de langages, *i.e.* en disant par exemple qu'un langage est LL(1) s'il est reconnu par une grammaire LL(1). Les comparaisons de ces classes de grammaires et de langages, en termes ensemblistes, sont les suivantes :



Il s'agit là d'inclusion strictes. On note en particulier que la classe LR(1) est strictement plus expressive que la classe LL(1), autant en termes de grammaires qu'en terme de langages.

4.5 L'outil `menhir`

L'analyse ascendante est puissante mais le calcul des tables est complexe. C'est pourquoi il est automatisé par des outils. C'est la grande famille de `yacc` (YACC signifie *Yet Another Compiler Compiler*), dans laquelle on trouve `bison`, `ocamlyacc`, `cup`, `menhir`, etc. On choisir ici de présenter l'outil `menhir`, un analyseur LR(1) pour OCaml. L'outil `menhir` s'utilise conjointement avec un analyseur lexical, typiquement construit par `ocamllex`.

La figure 4.6 contient un exemple de source `menhir` pour l'analyse syntaxique d'expressions arithmétiques. Les deux premières lignes déclarent les lexèmes, à savoir `PLUS` (pour `+`), `TIMES` (pour `*`), `LEFTPAR` (pour `(`), `RIGHTPAR` (pour `)`), `CONST` (pour une constante littérale) et `EOF` pour la fin de l'entrée. La déclaration du lexème `CONST` s'accompagne du type de la valeur de ce lexème, à savoir `int`. Ces lexèmes sont en tout point identiques à ceux de l'analyseur lexical de la figure 3.1.

Les règles de grammaire apparaissent après la ligne `%%`. Il y a deux non terminaux, `phrase` et `expression`. Les points-virgules sont optionnels ; ils ne sont là que pour aider à la lecture. Il faut lire cette grammaire comme ceci :

$$
\begin{array}{rcl}
\mathit{phrase} & \to & \mathit{expression}\ \texttt{EOF} \\
\mathit{expression} & \to & \mathit{expression}\ \texttt{PLUS}\ \mathit{expression} \\
 & | & \mathit{expression}\ \texttt{TIMES}\ \mathit{expression} \\
 & | & \texttt{LEFTPAR}\ \mathit{expression}\ \texttt{RIGHTPAR} \\
 & | & \texttt{CONST}
\end{array}
$$

À chaque production est associée une action, qui est un code OCaml arbitraire écrit entre accolades. Ici, ce code calcule la valeur entière de l'expression qui est reconnue. Dans un exemple plus réaliste, ce code construirait un arbre de syntaxe abstraite pour l'expression arithmétique. La syntaxe `e1 = ...` dans les règles de grammaire permet de récupérer la

```
%token PLUS TIMES LEFTPAR RIGHTPAR EOF
%token <int> CONST

%left PLUS
%left TIMES

%start <int> phrase

%%

phrase:
| e = expression; EOF  { e }

expression:
| e1 = expression; PLUS;  e2 = expression { e1 + e2 }
| e1 = expression; TIMES; e2 = expression { e1 * e2 }
| LEFTPAR; e = expression; RIGHTPAR       { e }
| i = CONST                               { i }
```

FIGURE 4.6 – Analyse syntaxique d'expressions arithmétiques avec `menhir`.

valeur d'un terminal (dans le cas de `CONST` ici) ou la valeur construite récursivement par un non terminal de la grammaire (ici `expression`). On peut ainsi construire la valeur d'une expression arithmétique de bas en haut, en récupérant les valeurs de ses sous-expressions.

Si un tel source `menhir` est contenu dans un fichier `arith.mly`, on le compile avec la commande

```
> menhir -v arith.mly
```

et on obtient du code OCaml dans deux fichiers `arith.ml` et `arith.mli`. Ce module exporte la déclaration d'un type `token`

```
type token = RPAR | PLUS | LPAR | INT of int | EOF
```

et une fonction `phrase` du type

```
val phrase: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int
```

Le premier est construit à partir des déclarations `%token` et le second à partir de la déclaration `%start`. Plusieurs déclarations `%start` sont possibles et donnent alors lieu à plusieurs fonctions exportées.

Lorsque la grammaire n'est pas LR(1), `menhir` annonce les conflits et les présente à l'utilisateur dans deux fichiers. Le fichier `.automaton` contient une description de l'automate LR(1) et les conflits y sont mentionnés. Le fichier `.conflicts` contient une explication des conflits, sous la forme d'une séquence de lexèmes conduisant à deux arbres de dérivation. En présence de conflits, `menhir` fait un choix arbitraire entre lecture et réduction mais l'utilisateur peut indiquer comment choisir entre lecture et réduction. Pour cela, on donne des priorités aux lexèmes et aux productions et des règles d'associativité. Par défaut, la priorité d'une production est celle de son lexème le plus à droite (mais elle peut être spécifiée explicitement). Si la priorité de la production est supérieure à celle du

lexème à lire, alors la réduction est favorisée. Inversement, si la priorité du lexème est supérieure, alors la lecture est favorisée. En cas d'égalité, l'associativité est consultée : un lexème associatif à gauche favorise la réduction et un lexème associatif à droite la lecture. Dans notre exemple, on a déclaré `PLUS` et `TIMES` comme associatifs à gauche avec la déclaration `%left`. De plus, on a déclaré `TIMES` comme plus prioritaire que `PLUS` en faisant apparaître `%left TIMES` plus bas que `%left PLUS`. Dès lors, tous les conflits sont résolus.

L'outil `menhir` offre de nombreux avantages par rapport aux outils traditionnels de la famille `yacc` comme `ocamlyacc`. On renvoie au manuel de `menhir` pour plus de détails [24].

Exercice 31. Écrire un source `menhir` pour la grammaire suivante :

$$
\begin{array}{rcl}
S & \rightarrow & E\ \texttt{EOF} \\
E & \rightarrow & \texttt{IF}\ E\ \texttt{THEN}\ E \\
 & | & \texttt{IF}\ E\ \texttt{THEN}\ E\ \texttt{ELSE}\ E \\
 & | & \texttt{CONST}
\end{array}
$$

Cette grammaire présente un conflit. L'expliquer. Proposer une manière de le résoudre.

Solution □

Notes bibliographiques. L'analyse LR a été inventée par Donald Knuth [16]. Elle est décrite en détail dans la section 4.7 de *Compilateurs : principes techniques et outils* de A. Aho, R. Sethi, J. Ullman (dit « le dragon ») [1, 2].

5 Typage statique

Si on écrit une expression telle que `"5" + 37` dans un langage de programmation, doit-on obtenir une erreur à la compilation (comme en OCaml), une erreur à l'exécution (comme en Python), l'entier 42 (comme en Visual Basic ou en PHP), la chaîne `"537"` (comme en Java) ou encore autre chose ? Et si on additionne non pas `"5"` et `37` mais deux expressions arbitraires, comment déterminer si cela est légal et ce que l'on doit faire le cas échéant ? Une réponse est le *typage*, une analyse qui associe un *type* à chaque valeur ou expression, dans le but de rejeter les programmes incohérents.

Certains langages sont *typés dynamiquement*, c'est-à-dire que les types sont associés aux valeurs et utilisés pendant l'exécution du programme, comme par exemple Lisp, PHP, Python, Julia, etc. D'autres langages sont *typés statiquement*, c'est-à-dire que les types sont associés aux expressions et utilisés pendant la compilation du programme, comme par exemple C, C++, Java, OCaml, Rust, Go, etc. Il existe des langages utilisant à la fois le typage statique et le typage dynamique. C'est le cas notamment des langages à objets, comme C++ ou Java. Nous y reviendrons dans le chapitre 8 relatif à la compilation des langages à objets.

Dans ce chapitre, on s'intéresse uniquement au typage statique. Le typage statique s'accompagne d'un slogan que l'on doit à Robin Milner :

well typed programs do not go wrong

Cela signifie que les programmes acceptés par le typage s'exécuteront sans échec, au sens de la sémantique opérationnelle présentée dans la section 2.2. Par exemple, un programme bien typé ne doit pas aboutir à l'utilisation d'un entier comme une fonction. Au delà de la propriété de sûreté, le typage doit aussi avoir la propriété d'être décidable. Il ne serait en effet pas raisonnable qu'un compilateur ne termine pas à cause du typage, de même qu'il ne serait pas raisonnable qu'il réponde « je ne sais pas ». Bien entendu, il suffirait de rejeter tous les programmes pour avoir automatiquement la propriété de sûreté mais cela ne constituerait pas un langage très intéressant. Le typage se doit donc d'être également *expressif*, en ne rejetant pas trop de programmes non-absurdes. Il y a une tension indéniable entre sûreté et expressivité du typage. Ainsi, le langage OCaml rejette un programme comme

```
(fun x -> x x) (fun x -> x x)
```

$$
\begin{array}{rcll}
e & ::= & x & \text{identificateur} \\
  & \mid & c & \text{constante } (1,\ 2,\ \ldots,\ \mathit{true},\ \ldots) \\
  & \mid & \mathit{op} & \text{primitive } (+,\ \times,\ \mathit{fst},\ \ldots) \\
  & \mid & \texttt{fun}\ x \rightarrow e & \text{fonction anonyme} \\
  & \mid & e\ e & \text{application} \\
  & \mid & (e,\ e) & \text{paire} \\
  & \mid & \texttt{let}\ x = e\ \texttt{in}\ e & \text{liaison locale}
\end{array}
$$

FIGURE 5.1 – Syntaxe abstraite de Mini-ML.

comme étant mal typé, bien que ce dernier ne pose en réalité aucune difficulté au moteur d'exécution d'OCaml.

Dans ce chapitre, on choisit d'illustrer le typage statique sur l'exemple du langage Mini-ML, déjà étudié au chapitre 2, dont on redonne la syntaxe abstraite dans la figure 5.1.

5.1 Typage simple de Mini-ML

On va commencer par un typage relativement simple de Mini-ML, avec ce que l'on appelle justement des *types simples*. Un type est noté τ et sa syntaxe abstraite est

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \text{types de base} \\
 & \mid & \tau \rightarrow \tau & \text{type d'une fonction} \\
 & \mid & \tau \times \tau & \text{type d'une paire}
\end{array}
$$

Ainsi, $\texttt{int} \rightarrow \texttt{int} \times \texttt{int}$ est un type, à savoir le type des fonctions qui reçoivent un entier en argument et renvoient une paire d'entiers. On ne précise pas ici ce que sont exactement les types de bases, car ils dépendent des constantes et des primitives que l'on a choisies pour Mini-ML. Pour donner un type à une expression quelconque, on va introduire la notion de *jugement de typage*. Il s'agit d'une relation ternaire entre un *environnement de typage* Γ, une expression e et un typage τ, que l'on note

$$\Gamma \vdash e : \tau$$

et qui se lit « dans l'environnement Γ, l'expression e a le type τ ». L'environnement Γ est une fonction des variables vers les types. On pourra ainsi donner un type à une expression e non nécessairement close, toute variable x libre dans e étant supposée avoir le type $\Gamma(x)$ donné par l'environnement.

On définit la relation $\Gamma \vdash e : \tau$ par un ensemble de règles d'inférence, données figure 5.2. Dans la règle de typage des constructions `fun` et `let`, on a utilisé la notation $\Gamma + x : \tau$ pour désigner une extension Γ' de l'environnement Γ définie par

$$\Gamma'(y) = \begin{cases} \tau & \text{si } y = x, \\ \Gamma(y) & \text{sinon.} \end{cases}$$

Avec ces règles de typage, on peut montrer que l'expression $\texttt{let}\ f = \texttt{fun}\ x \rightarrow +(x, 1)\ \texttt{in}\ f\ 2$ est bien typée de type `int` dans un environnement vide, avec la dérivation suivante (où

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \mathtt{int}}\ \text{etc.} \qquad \frac{}{\Gamma \vdash + : \mathtt{int} \times \mathtt{int} \to \mathtt{int}}\ \text{etc.}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_2\ e_1 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

FIGURE 5.2 – Typage de Mini-ML avec des types simples.

quelques parties ont été omises faute de place) :

$$\dfrac{\dfrac{\dfrac{\vdots \qquad \dfrac{\vdots}{x : \mathtt{int} \vdash (x,1) : \mathtt{int} \times \mathtt{int}}}{x : \mathtt{int} \vdash +(x,1) : \mathtt{int}}}{\emptyset \vdash \mathtt{fun}\ x \to +(x,1) : \mathtt{int} \to \mathtt{int}} \qquad \dfrac{\dfrac{}{\ldots \vdash f : \mathtt{int} \to \mathtt{int}} \quad \dfrac{}{\ldots \vdash 2 : \mathtt{int}}}{f : \mathtt{int} \to \mathtt{int} \vdash f\ 2 : \mathtt{int}}}{\emptyset \vdash \mathtt{let}\ f = \mathtt{fun}\ x \to +(x,1)\ \mathtt{in}\ f\ 2 : \mathtt{int}}$$

En revanche, on ne peut pas typer le programme 1 2. En effet, il nous faudrait conclure par la règle d'application, c'est-à-dire

$$\frac{\Gamma \vdash 1 : \tau' \to \tau \qquad \Gamma \vdash 2 : \tau'}{\Gamma \vdash 1\ 2 : \tau}$$

et il n'existe pas de règle pour donner à l'expression 1 un type de fonction. De même, il n'est pas possible de typer le programme $\mathtt{fun}\ x \to x\ x$. Il faudrait pour cela une dérivation de la forme

$$\dfrac{\dfrac{\Gamma + x : \tau_1 \vdash x : \tau_3 \to \tau_2 \qquad \Gamma + x : \tau_1 \vdash x : \tau_3}{\Gamma + x : \tau_1 \vdash x\ x : \tau_2}}{\Gamma \vdash \mathtt{fun}\ x \to x\ x : \tau_1 \to \tau_2}$$

avec $\tau_3 = \tau_1$ et $\tau_1 = \tau_3 \to \tau_2$. Or, il n'a pas de solution à ces deux égalités, car les types sont finis (ce qui est implicite dans leur définition par une syntaxe abstraite). Nous verrons plus loin un système de types qui permet de typer cette expression.

Certaines expressions admettent plusieurs types. Ainsi, l'expression $\mathtt{fun}\ x \to x$ admet aussi bien le type $\mathtt{int} \to \mathtt{int}$ que le type $\mathtt{bool} \to \mathtt{bool}$. Ce n'est pas pour autant du *polymorphisme* (dont nous parlerons en détail plus loin). En effet, pour une occurrence donnée de $\mathtt{fun}\ x \to x$ il faut *choisir* un type. Ainsi, le terme $\mathtt{let}\ f = \mathtt{fun}\ x \to x\ \mathtt{in}\ (f\ 1, f\ \mathtt{true})$ n'est pas typable, car il n'y a pas de type τ tel que

$$f : \tau \to \tau \vdash (f\ 1,\ f\ \mathtt{true}) : \tau_1 \times \tau_2$$

Exercice 32. Montrer qu'on peut donner un type à $((\mathtt{fun}\ x \to x)\ (\mathtt{fun}\ x \to x))\ 42$.

Solution □

En particulier, on ne peut pas donner un type satisfaisant à une primitive comme `fst` ; il faudrait choisir entre une infinité de types possibles :

$$\begin{array}{l} \texttt{int} \times \texttt{int} \to \texttt{int} \\ \texttt{int} \times \texttt{bool} \to \texttt{int} \\ \texttt{bool} \times \texttt{int} \to \texttt{bool} \\ (\texttt{int} \to \texttt{int}) \times \texttt{int} \to \texttt{int} \to \texttt{int} \\ \text{etc.} \end{array}$$

Il en va de même pour les primitives *opif* et *opfix*. Si on ne peut pas donner un type satisfaisant à *opfix*, on pourrait en revanche donner une règle de typage pour une construction `let rec` qui serait primitive :

$$\frac{\Gamma + x : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec } x = e_1 \texttt{ in } e_2 : \tau_2}$$

Et si on souhaite exclure les *valeurs* récursives, on peut la modifier ainsi :

$$\frac{\Gamma + (f : \tau \to \tau_1) + (x : \tau) \vdash e_1 : \tau_1 \qquad \Gamma + (f : \tau \to \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec } f\ x = e_1 \texttt{ in } e_2 : \tau_2}$$

Exercice 33. Donner la dérivation de typage de l'expression

$$\texttt{let rec } \mathit{fact}\ n \ = \ \texttt{if } {=}(n,\, 0) \texttt{ then } 1 \texttt{ else } \times(n,\, \mathit{fact}\ ({+}(n,\, -1))) \texttt{ in } \mathit{fact}\ 2.$$

Solution □

Algorithme de typage. Pour l'instant, nous avons défini une *relation* de typage, entre une expression et un type, mais nous ne disposons pas d'un *algorithme* de typage, qui décide si une expression est ou non bien typée. En particulier, un algorithme de typage doit choisir un type à donner à x quand il tombe sur une expression de la forme $\texttt{fun } x \to e$. Considérons une approche simple où ce type est donné par l'utilisateur, par exemple sous la forme $\texttt{fun } x : \tau \to e$. C'est ainsi qu'on procède dans de nombreux langages typés statiquement, comme C ou Java par exemple, où les paramètres formels d'une fonction sont accompagnés de leur type. Dès lors, il suffit de suivre les règles de la figure 5.2 pour calculer le type d'une expression par récurrence sur la structure de cette expression. En notant $T(\Gamma, e)$ cet algorithme de calcul de type, on a :

$$\begin{array}{rcl} T(\Gamma,\, x) & = & \Gamma(x) \\ T(\Gamma,\, n) & = & \texttt{int} \\ T(\Gamma,\, +) & = & \texttt{int} \times \texttt{int} \to \texttt{int} \\ T(\Gamma,\, \texttt{fun } x : \tau_1 \to e) & = & \tau_1 \to T(\Gamma + x : \tau_1,\, e) \\ T(\Gamma,\, (e_1,\, e_2)) & = & T(\Gamma,\, e_1) \times T(\Gamma,\, e_2) \\ T(\Gamma,\, \texttt{let } x = e_1 \texttt{ in } e_2) & = & T(\Gamma + x : T(\Gamma,\, e_1),\, e_2) \\ T(\Gamma,\, e_2\ e_1) & = & \tau_2 \quad \text{si } T(\Gamma,\, e_2) = T(\Gamma,\, e_1) \to \tau_2 \end{array}$$

Parce que l'algorithme de typage procède récursivement sur la syntaxe des expressions, sans qu'on ait besoin de faire de choix ou de retour en arrière, on dit qu'il est *dirigé par la syntaxe* (en anglais *syntax-directed*). On note que seul le dernier cas implique une vérification, à savoir que le type de e_2 est bien un type d'une fonction attendant un argument du type de e_1. Dans tous les autres cas, l'expression est bien typée dès lors que ses sous-expressions sont bien typées.

Exercice 34. Programmer l'algorithme de typage ci-dessus en OCaml. Solution □

En pratique, un compilateur a besoin de conserver les types de toutes les expressions du programme pour les besoins de ses phases ultérieures. Pour cette raison, on n'écrit pas une fonction qui calcule le type d'une expression, comme ici, mais une fonction qui renvoie de nouveaux arbres de syntaxe abstraite, dans lesquels chaque sous-expression est décorée par son type. Une façon efficace de procéder en OCaml consiste à introduire deux types mutuellement récursifs, à savoir un type enregistrement pour la décoration proprement dite et un type algébrique pour les différentes constructions de la syntaxe abstraite.

```
type expression = { node: node; typ: typ; }
and node =
  | Var of string
  | App of expression * expression
  | ...
```

5.2 Sûreté du typage

Maintenant que nous avons défini la relation de typage pour Mini-ML, nous pouvons chercher à vérifier si le slogan de Robin Milner, à savoir que les programmes bien typés ne plantent pas, est valide pour Mini-ML. Pour cela, on va se servir de la sémantique opérationnelle à petits pas définie dans la section 2.2.2 et montrer le résultat suivant :

Si $\emptyset \vdash e : \tau$, alors l'évaluation de e est infinie ou se termine sur une valeur.

La preuve de ce théorème s'appuie sur deux lemmes, dits de *progrès* et de *préservation*. Le premier dit qu'une expression bien typée qui n'est pas une valeur peut progresser dans son calcul, en effectuant au moins un pas de réduction. Le second dit que le type d'une expression est préservé par un pas de réduction. Commençons par le lemme de progrès.

Lemme 4 (progrès)**.** Si $\emptyset \vdash e : \tau$, alors soit e est une valeur, soit il existe e' tel que $e \to e'$.

Preuve. On procède par récurrence sur la dérivation de typage $\emptyset \vdash e : \tau$. Supposons par exemple que $e = e_2\ e_1$; on a donc

$$\frac{\emptyset \vdash e_2 : \tau_1 \to \tau_2 \quad \emptyset \vdash e_1 : \tau_1}{\emptyset \vdash e_2\ e_1 : \tau_2}$$

On applique l'hypothèse de récurrence à e_2 :
- si $e_2 \to e_2'$, alors $e_2\ e_1 \to e_2'\ e_1$ par le lemme 1 de passage au contexte (page 31) ;
- si e_2 est une valeur, supposons $e_2 = \texttt{fun}\ x \to e_3$ (il y a aussi + etc.). On applique l'hypothèse de récurrence à e_1 :
  - si $e_1 \to e_1'$, alors $e_2\ e_1 \to e_2\ e_1'$ (même lemme) ;
  - si e_1 est une valeur, alors $e_2\ e_1 \to e_3[x \leftarrow e_1]$.

Les autres cas sont laissés en exercice. □

Venons-en maintenant au lemme de préservation. Pour le montrer, on va avoir besoin de plusieurs lemmes sur les dérivations de typage. Le premier affirme que l'ordre des ajouts dans Γ n'est pas important.

Lemme 5 (permutation)**.** Si $\Gamma{+}x : \tau_1{+}y : \tau_2 \vdash e : \tau$ et $x \neq y$ alors $\Gamma{+}y : \tau_2{+}x : \tau_1 \vdash e : \tau$ et la dérivation a la même hauteur.

Preuve. Par récurrence immédiate sur la dérivation de typage. □

Le lemme suivant affirme qu'on peut ajouter à Γ des hypothèses inutiles sans modifier pour autant la relation de typage.

Lemme 6 (affaiblissement)**.** Si $\Gamma \vdash e : \tau$ et $x \notin dom(\Gamma)$, alors $\Gamma + x : \tau' \vdash e : \tau$ et la dérivation a la même hauteur.

Preuve. Là encore, par récurrence immédiate sur la dérivation de typage. □

Pour ces deux lemmes, on a pris soin d'ajouter « et la dérivation a la même hauteur » au résultat, ce qui nous permettra par la suite d'appliquer des hypothèses de récurrence à des dérivations obtenues grâce à ces lemmes. On continue par un *lemme clé* qui montre que le typage est préservé par certaines substitutions.

Lemme 7 (préservation par substitution)**.** Si $\Gamma + x : \tau' \vdash e : \tau$ et $\Gamma \vdash e' : \tau'$ alors $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Preuve. On procède par récurrence sur la dérivation $\Gamma + x : \tau' \vdash e : \tau$.
- cas d'une variable $e = y$:
  - si $x = y$ alors $e[x \leftarrow e'] = e'$ et $\tau = \tau'$;
  - si $x \neq y$ alors $e[x \leftarrow e'] = e$ et $\tau = \Gamma(y)$.
- cas d'une abstraction $e = \texttt{fun}\ y \rightarrow e_1$:
  On peut supposer $y \neq x$, $y \notin dom(\Gamma)$ et y non libre dans e', quitte à renommer y. On a $\Gamma + x : \tau' + y : \tau_2 \vdash e_1 : \tau_1$ et donc $\Gamma + y : \tau_2 + x : \tau' \vdash e_1 : \tau_1$ (permutation). D'autre part $\Gamma \vdash e' : \tau'$ et donc $\Gamma + y : \tau_2 \vdash e' : \tau'$ (affaiblissement). Par hypothèse de récurrence, on a donc $\Gamma + y : \tau_2 \vdash e_1[x \leftarrow e'] : \tau_1$ et donc $\Gamma \vdash (\texttt{fun}\ y \rightarrow e_1)[x \leftarrow e'] : \tau_2 \rightarrow \tau_1$, c'est-à-dire $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Les autres cas sont laissés en exercice. □

On est maintenant en mesure de montrer le lemme de préservation.

Lemme 8 (préservation)**.** Si $\emptyset \vdash e : \tau$ et $e \rightarrow e'$ alors $\emptyset \vdash e' : \tau$.

Preuve. On procède par récurrence sur la dérivation $\emptyset \vdash e : \tau$.
- cas $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$:

$$\frac{\emptyset \vdash e_1 : \tau_1 \qquad x : \tau_1 \vdash e_2 : \tau_2}{\emptyset \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

  - si $e_1 \rightarrow e_1'$, par hypothèse de récurrence on a $\emptyset \vdash e_1' : \tau_1$ et donc $\emptyset \vdash \texttt{let}\ x = e_1'\ \texttt{in}\ e_2 : \tau_2$;
  - si e_1 est une valeur et $e' = e_2[x \leftarrow e_1]$, alors on applique le lemme de préservation par substitution.
- cas $e = e_1\ e_2$:
  - si $e_1 \rightarrow e_1'$ ou si e_1 valeur et $e_2 \rightarrow e_2'$ alors on utilise l'hypothèse de récurrence ;
  - si $e_1 = \texttt{fun}\ x \rightarrow e_3$ et e_2 valeur alors $e' = e_3[x \leftarrow e_2]$ et on applique là encore le lemme de substitution.

Les autres cas sont laissés en exercice. □

Le théorème de sûreté du typage est une conséquence facile des lemmes de progrès et de préservation. Plutôt que de l'énoncer sous la forme « Si $\emptyset \vdash e : \tau$, alors l'évaluation de e est infinie ou se termine sur une valeur » donnée plus haut, on choisit une autre formulation, équivalente.

Théorème 2 (sûreté du typage)**.** Si $\emptyset \vdash e : \tau$ et $e \xrightarrow{\star} e'$ avec e' irréductible, alors e' est une valeur.

PREUVE. On a $e \rightarrow e_1 \rightarrow \cdots \rightarrow e'$ et par applications répétées du lemme de préservation, on a donc $\emptyset \vdash e' : \tau$. Par le lemme de progrès, e' se réduit ou est une valeur. C'est donc une valeur. □

5.3 Polymorphisme paramétrique

Il est contraignant de donner un type unique à $\texttt{fun}\ x \rightarrow x$ dans une expression comme $\texttt{let}\ f = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ e$ car on aimerait pouvoir se servir plusieurs fois de la fonction f dans l'expression e, sur des arguments de types différents. De même, on aimerait pouvoir donner plusieurs types aux primitives telles que *fst* ou *snd*. Une solution à ce problème s'appelle le *polymorphisme paramétrique*. Elle consiste à étendre l'algèbre des types de la façon suivante :

$$\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \dots & \text{types de base} \\
 & \mid & \tau \rightarrow \tau & \text{type d'une fonction} \\
 & \mid & \tau \times \tau & \text{type d'une paire} \\
 & \mid & \alpha & \text{variable de type} \\
 & \mid & \forall \alpha . \tau & \text{type polymorphe}
\end{array}$$

Aux trois constructions de types existantes, on a rajouté deux nouvelles constructions : des variables de type, dénotées dans la suite par des lettres grecques, et des types universellement quantifiés. Intuitivement, une variable de type α représente un type quelconque et un type universellement quantifié $\forall\alpha.\,\tau$ est le type d'une valeur, dite polymorphe, qui peut prendre le type τ quel que soit le type α. La construction $\forall\alpha.\,\tau$ est un lieur, qui lie la variable α dans le type τ. En particulier, on peut la renommer à loisir, exactement comme une variable de programme introduite par `fun` ou `let`. Sans surprise, il vient donc une notion de variable libre dans un type, analogue à celle de variable libre dans une expression.

Définition 20 (variables libres d'un type)**.** On note $\mathcal{L}(\tau)$ l'ensemble des variables de types *libres* dans τ, défini par

$$\begin{array}{rcl}
\mathcal{L}(\texttt{int}) & = & \emptyset \\
\mathcal{L}(\alpha) & = & \{\alpha\} \\
\mathcal{L}(\tau_1 \rightarrow \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\forall \alpha.\tau) & = & \mathcal{L}(\tau) \setminus \{\alpha\}
\end{array}$$

Pour un environnement de typage Γ, on pose

$$\mathcal{L}(\Gamma) = \bigcup_{x \in dom(\Gamma)} \mathcal{L}(\Gamma(x)).$$

□

Si une variable de type est libre dans un type, on peut chercher à lui substituer un autre type, de manière analogue à la substitution d'une variable par une valeur dans une expression. Comme pour cette dernière, on prend soin d'arrêter la substitution lorsque l'on rencontre une variable liée portant le même nom que la variable à remplacer.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \text{ etc.} \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}} \text{ etc.}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_2\ e_1 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\tau} \qquad \frac{\Gamma \vdash e : \forall \alpha.\tau}{\Gamma \vdash e : \tau[\alpha \leftarrow \tau']}$$

FIGURE 5.3 – Typage de Mini-ML avec le système F.

Définition 21 (substitution de type)**.** On note $\tau[\alpha \leftarrow \tau']$ la substitution de α par τ' dans τ, définie par

$$\begin{array}{rcl}
\texttt{int}[\alpha \leftarrow \tau'] & = & \texttt{int} \\
\alpha[\alpha \leftarrow \tau'] & = & \tau' \\
\beta[\alpha \leftarrow \tau'] & = & \beta \quad \text{si } \beta \neq \alpha \\
(\tau_1 \rightarrow \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \rightarrow \tau_2[\alpha \leftarrow \tau'] \\
(\tau_1 \times \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \times \tau_2[\alpha \leftarrow \tau'] \\
(\forall \alpha.\tau)[\alpha \leftarrow \tau'] & = & \forall \alpha.\tau \\
(\forall \beta.\tau)[\alpha \leftarrow \tau'] & = & \forall \beta.\tau[\alpha \leftarrow \tau'] \quad \text{si } \beta \neq \alpha
\end{array}$$

□

Les règles de typage dans ce nouveau système de types sont données figure 5.3. Ce sont *exactement* les mêmes règles qu'auparavant (figure 5.2 page 81), auxquelles on adjoint deux nouvelles règles (sur la dernière ligne). La première dit que le type d'une expression peut être *généralisé* par rapport à une variable de type α dès lors que cette variable n'apparaît pas dans le contexte Γ.

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\tau}$$

La seconde dit qu'un type polymorphe peut être *spécialisé* par n'importe quel type.

$$\frac{\Gamma \vdash e : \forall \alpha.\tau}{\Gamma \vdash e : \tau[\alpha \leftarrow \tau']}$$

Le système ainsi obtenu s'appelle le *système F*. On note que ce système n'est pas dirigé par la syntaxe. En effet, trois règles s'appliquent maintenant potentiellement à chaque expression : la règle de l'ancien système, la règle de généralisation et la règle de spécialisation.

On peut maintenant typer l'expression $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$ qu'il n'était pas possible de typer auparavant avec les types simples.

$$\dfrac{\dfrac{\dfrac{x : \alpha \vdash x : \alpha}{\vdash \texttt{fun}\ x \to x : \alpha \to \alpha}}{\vdash \texttt{fun}\ x \to x : \forall\alpha.\alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\dots \vdash f : \forall\alpha.\alpha \to \alpha}{\dots \vdash f : \texttt{int} \to \texttt{int}} \quad \vdots}{\dots \vdash f\ 1 : \texttt{int}} \qquad \dfrac{\vdots}{\dots \vdash f\ \texttt{true} : \texttt{bool}}}{f : \forall\alpha.\alpha \to \alpha \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}}{\vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}$$

Pour y parvenir, on a donné à la fonction $\texttt{fun}\ x \to x$ le type polymorphe $\forall\alpha.\alpha \to \alpha$, ce qui nous a permis de l'utiliser à la fois sur un entier, en spécialisant avec le type `int`, et sur un booléen, en spécialisant avec le type `bool`. On peut maintenant donner un type satisfaisant aux primitives de Mini-ML :

$$\begin{array}{rl} \mathit{fst} : & \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha \\ \mathit{snd} : & \forall\alpha.\forall\beta.\alpha \times \beta \to \beta \\ \mathit{opif} : & \forall\alpha.\texttt{bool} \times \alpha \times \alpha \to \alpha \\ \mathit{opfix} : & \forall\alpha.(\alpha \to \alpha) \to \alpha \end{array}$$

Exercice 35. Donner une dérivation de typage pour l'expression $\texttt{fun}\ x \to x\ x$.
Solution □

Remarque. La condition $\alpha \notin \mathcal{L}(\Gamma)$ dans la règle de généralisation est cruciale. Sans elle, on pourrait typer $\texttt{fun}\ x \to x$ avec le type $\alpha \to \forall\alpha.\alpha$, de la manière suivante

$$\dfrac{\dfrac{\Gamma + x : \alpha \vdash x : \alpha}{\Gamma + x : \alpha \vdash x : \forall\alpha.\alpha}}{\Gamma \vdash \texttt{fun}\ x \to x : \alpha \to \forall\alpha.\alpha}$$

et typer avec succès l'expression $(\texttt{fun}\ x \to x)\ 1\ 2$, c'est-à-dire un programme dont l'exécution aboutit à l'utilisation d'un entier comme une fonction. La sûreté du typage ne serait donc pas garantie. □

On peut montrer que le système F est un système de types sûr pour Mini-ML, comme nous l'avons fait plus haut pour les types simples, même si la preuve est plus complexe. Malheureusement, on n'en déduit pas pour autant facilement un algorithme de typage. En effet, il s'avère que le problème de l'*inférence de type* (étant donné e, existe-t-il τ tel que $\vdash e : \tau$?) tout comme celui de la vérification (étant donnés e et τ, a-t-on $\vdash e : \tau$?) ne sont pas décidables. Pour obtenir une inférence de types décidable, on va restreindre la puissance du système F.

Système de Hindley-Milner. Une solution est apportée par le système de Hindley-Milner. Elle consiste à limiter la quantification universelle $\forall$ en tête des types ; on parle alors de quantification prénexe, comme en logique. Pour le faire, on distingue les types sans quantification, toujours notés τ, des types pouvant être universellement quantifiés,

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \text{ etc.} \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}} \text{ etc.}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \; e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma + x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma} \qquad \frac{\Gamma \vdash e : \forall \alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau]}$$

FIGURE 5.4 – Typage de Mini-ML avec Hindley-Milner.

appelés *schémas* et notés σ.

$$\begin{array}{lcll} \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \text{types de base} \\ & \mid & \tau \rightarrow \tau & \text{type d'une fonction} \\ & \mid & \tau \times \tau & \text{type d'une paire} \\ & \mid & \alpha & \text{variable de type} \\ \\ \sigma & ::= & \tau & \text{schémas} \\ & \mid & \forall \alpha.\sigma & \end{array}$$

Dans le système de Hindley-Milner, les types suivants sont toujours acceptés

$$\begin{array}{l} \forall \alpha.\alpha \rightarrow \alpha \\ \forall \alpha.\forall \beta.\alpha \times \beta \rightarrow \alpha \\ \forall \alpha.\texttt{bool} \times \alpha \times \alpha \rightarrow \alpha \\ \forall \alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha \end{array}$$

mais plus les types tels que

$$(\forall \alpha.\alpha \rightarrow \alpha) \rightarrow (\forall \alpha.\alpha \rightarrow \alpha).$$

Un environnement de typage Γ associe à des variables des schémas de types et la relation de typage a maintenant la forme $\Gamma \vdash e : \sigma$. Les règles du système de Hindley-Milner sont données figure 5.4. On note que la spécialisation remplace une variable de type α par un type τ et non un schéma, sans quoi le schéma obtenu serait mal formé. On note également que seule la construction `let` permet d'introduire un type polymorphe dans l'environnement. En particulier, on peut toujours typer

$$\texttt{let } f = \texttt{fun } x \rightarrow x \texttt{ in } (f\ 1,\ f\ \texttt{true})$$

avec $f : \forall \alpha.\alpha \rightarrow \alpha$ dans le contexte pour typer $(f\ 1,\ f\ \texttt{true})$. En revanche, la règle de typage

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

n'introduit pas un type polymorphe, car sinon $\tau_1 \rightarrow \tau_2$ serait mal formé. En particulier, on ne peut plus typer $\texttt{fun } x \rightarrow x\ x$.

Remarque. Dans un langage comme OCaml, qui implémente le système de Hindley-Milner, la quantification universelle est implicite car elle est nécessairement prénexe. Ainsi, un type présenté comme

```
List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

doit être lu comme $\forall\alpha.\forall\beta.(\alpha \to \beta \to \alpha) \to \alpha \to \beta\ list \to \alpha$.

5.4 Inférence de types

Cherchons à programmer une inférence de types pour le système de Hindley-Milner. Naturellement, on va chercher à procéder par récurrence sur la structure du programme. Or, pour une expression donnée, trois règles peuvent s'appliquer : la règle du système monomorphe, la règle de généralisation

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma}$$

ou encore la règle de spécialisation

$$\frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau]}$$

Il semble difficile de choisir entre les trois, car on ne peut pas préjuger de l'usage qui va être fait de l'expression une fois typée. Et on ne souhaite pas procéder par essais et erreurs, car le typage risquerait d'être trop coûteux au final. Pour parvenir à nos fins, nous allons modifier la présentation du système de Hindley-Milner pour qu'il soit *dirigé par la syntaxe*, *i.e.* pour que, pour toute expression, au plus une règle ne s'applique. Les règles auront la même puissance d'expression : tout terme clos est typable dans un système si et seulement si il est typable dans l'autre.

Pour définir cette nouvelle présentation, nous avons besoin de deux ingrédients nouveaux. On introduit d'une part la relation $\tau \leq \sigma$ dont le sens est « τ est une spécialisation du schéma σ ». Formellement, cette relation est définie par

$$\tau \leq \forall\alpha_1...\alpha_n.\tau' \quad \text{ssi} \quad \exists\tau_1...\exists\tau_n.\ \tau = \tau'[\alpha_1 \leftarrow \tau_1, ..., \alpha_n \leftarrow \tau_n].$$

On a par exemple $\texttt{int} \times \texttt{bool} \to \texttt{int} \leq \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$. On introduit d'autre part une opération de *généralisation*, notée $Gen(\tau, \Gamma)$, qui construit un schéma de types en quantifiant τ par rapport à toutes ses variables libres n'apparaissant pas dans Γ, c'est-à-dire

$$Gen(\tau, \Gamma) \stackrel{\text{def}}{=} \forall\alpha_1...\forall\alpha_n.\tau \quad \text{où} \quad \{\alpha_1, ..., \alpha_n\} = \mathcal{L}(\tau)\backslash\mathcal{L}(\Gamma).$$

Muni de ces deux ingrédients, on peut définir le système de Hindley-Milner dirigé par la syntaxe. Ses règles sont données figure 5.5. La différence par rapport au système précédent se situe dans la généralisation et la spécialisation uniquement. Plutôt que de les proposer sous la forme de deux nouvelles règles, elles sont placées à des endroits stratégiques. D'une part, la spécialisation se fait au niveau des axiomes, lorsqu'on utilise une variable ou une primitive. On suppose ici qu'une fonction *type* donne le type d'une primitive. Ainsi, on a $type(fst) = \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$ et donc en particulier $\Gamma \vdash fst : \texttt{int} \times \texttt{bool} \to \texttt{int}$ avec cette règle. D'autre part, la généralisation se fait au niveau du `let`, la seule construction à même d'introduire un schéma dans l'environnement, en généralisant au maximum.

$$\frac{\tau \le \Gamma(x)}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \ \text{etc.} \qquad \frac{\tau \le \mathit{type}(op)}{\Gamma \vdash op : \tau}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \mathit{Gen}(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

FIGURE 5.5 – Typage de Mini-ML avec Hindley-Milner, dirigé par la syntaxe.

On peut toujours typer l'expression $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1,\ f\ \texttt{true})$, mais cette fois la dérivation est entièrement dirigée par la syntaxe :

$$\dfrac{\dfrac{\dfrac{\alpha \le \alpha}{x : \alpha \vdash x : \alpha}}{\vdash \texttt{fun}\ x \to x : \alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\texttt{int} \to \texttt{int} \le \forall\alpha.\alpha \to \alpha}{\Gamma \vdash f : \texttt{int} \to \texttt{int}} \quad \vdots}{\Gamma \vdash f\ 1 : \texttt{int}} \qquad \dfrac{\dfrac{\texttt{bool} \to \texttt{bool} \le \forall\alpha.\alpha \to \alpha}{\Gamma \vdash f : \texttt{bool} \to \texttt{bool}} \quad \vdots}{\Gamma \vdash f\ \texttt{true} : \texttt{bool}}}{\Gamma \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}}{\vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}$$

en posant $\Gamma = f : \mathit{Gen}(\alpha \to \alpha,\ \emptyset) = f : \forall\alpha.\alpha \to \alpha$. Si cette dérivation est dirigée par la syntaxe, elle contient toujours une part de magie : pour typer $\texttt{fun}\ x \to x$ il faut choisir un type à donner à x (ici on a choisi α) et pour typer f il faut choisir une certaine instance de $\forall\alpha.\alpha \to \alpha$ (ici on a choisi $\texttt{int} \to \texttt{int}$ la première fois et $\texttt{bool} \to \texttt{bool}$ la seconde). Ces deux problèmes se posent de manière générale, pour toute construction `fun` d'une part et pour toute utilisation d'une variable (ou d'une primitive polymorphe) d'autre part. Le problème n'est pas trivial, car il y a une distance arbitraire entre l'endroit où l'on type une fonction et l'endroit où on l'utilise, tout comme entre l'endroit où l'on type une variable et l'endroit où le type choisi intervient. Fort heureusement, il existe une solution à ces deux problèmes, connue sous le nom *d'algorithme W*.

L'algorithme W. L'idée de l'algorithme W consiste à différer le choix des types jusqu'au seul moment où ce choix a une réelle incidence, c'est-à-dire au moment de l'application, lorsque l'on vérifie que l'argument passé à une fonction a bien le type attendu par cette fonction. Pour cela, on utilise de *nouvelles variables de types* pour représenter des types encore inconnus, à savoir le type de x dans $\texttt{fun}\ x \to e$ d'une part et les types avec lesquels spécialiser le schéma $\Gamma(x)$ lors de l'utilisation d'une variable x d'autre part. Ce ne sont pas des variables de types au sens usuel, mais des *inconnues*, susceptibles de recevoir plus tard des valeurs plus précises que des variables de types. Leur résolution se fait au moment du typage des applications, en unifiant le type de l'argument et le type attendu par la fonction. Un tel problème d'*unification* peut être posé en les termes suivants : étant donnés deux types τ_1 et τ_2 contenant des variables de types $\alpha_1, \dots, \alpha_n$, existe-t-il une spécialisation θ, c'est-à-dire une fonction des variables α_i vers des types, telle que $\theta(\tau_1) = \theta(\tau_2)$? Ainsi, le problème d'unification

$$\alpha \times \beta \to \texttt{int} \stackrel{?}{=} \texttt{int} \times \texttt{bool} \to \gamma$$

$$
\begin{aligned}
\mathit{unifier}(\tau, \tau) &= \text{succès} \\
\mathit{unifier}(\tau_1 \rightarrow \tau_1', \tau_2 \rightarrow \tau_2') &= \mathit{unifier}(\tau_1, \tau_2)\ ;\ \mathit{unifier}(\tau_1', \tau_2') \\
\mathit{unifier}(\tau_1 \times \tau_1', \tau_2 \times \tau_2') &= \mathit{unifier}(\tau_1, \tau_2)\ ;\ \mathit{unifier}(\tau_1', \tau_2') \\
\mathit{unifier}(\alpha, \tau) &= \begin{array}{l}\text{si } \alpha \notin \mathcal{L}(\tau),\ \text{remplacer } \alpha \text{ par } \tau \text{ partout} \\ \text{sinon, échec}\end{array} \\
\mathit{unifier}(\tau, \alpha) &= \mathit{unifier}(\alpha, \tau) \\
\mathit{unifier}(\tau_1, \tau_2) &= \text{échec dans tous les autres cas}
\end{aligned}
$$

FIGURE 5.6 – Pseudo-code de l'algorithme d'unification.

admet une solution, à savoir $\{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{bool}, \gamma \mapsto \texttt{int}\}$. De même, le problème d'unification

$$\alpha \times \texttt{int} \rightarrow \alpha \times \texttt{int} \stackrel{?}{=} \gamma \rightarrow \gamma$$

admet une solution, à savoir $\{\gamma \mapsto \alpha \times \texttt{int}\}$. En revanche, le problème

$$\alpha \rightarrow \texttt{int} \stackrel{?}{=} \beta \times \gamma$$

n'admet pas de solution, car un type $\rightarrow$ ne peut être égal à un type $\times$, de même que le problème

$$\alpha \rightarrow \texttt{int} \stackrel{?}{=} \alpha$$

car les types sont finis. Le pseudo-code d'un algorithme d'unification est donné figure 5.6. Ce code est volontairement écrit sous une forme impérative, au sens où un appel à $\mathit{unifier}(\tau_1, \tau_2)$ conduit à une résolution globale, par effet de bord, de certaines variables de types. On parle d'*unification destructive*. Bien d'autres façons de l'écrire sont possibles, y compris dans un style purement applicatif. Cet algorithme nous donne l'*unificateur le plus général* (en anglais *most general unifier* ou mgu), au sens où toute spécialisation θ telle que $\theta(\tau_1) = \theta(\tau_2)$ est une instance du résultat de $\mathit{unifier}(\tau_1, \tau_2)$.

Avant de décrire l'algorithme W, commençons par en illustrer l'idée sur un exemple, à savoir l'expression $\texttt{fun}\ x \rightarrow +(\mathit{fst}\ x, 1)$. Comme il s'agit d'une fonction, on donne à x le type α_1, une nouvelle variable de type. Dans l'environnement $x : \alpha_1$, on cherche maintenant à typer l'expression $+(\mathit{fst}\ x, 1)$. La primitive $+$ a le type $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$. Typons son argument, à savoir l'expression $(\mathit{fst}\ x, 1)$. La primitive fst a pour type le schéma $\forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \alpha$, qu'il faut donc spécialiser. On lui donne donc le type $\alpha_2 \times \beta_1 \rightarrow \alpha_2$, pour deux nouvelles variables de types α_2 et β_1. L'application $\mathit{fst}\ x$ impose d'unifier α_1 et $\alpha_2 \times \beta_1$, ce qui donne $\{\alpha_1 \mapsto \alpha_2 \times \beta_1\}$. L'expression $(\mathit{fst}\ x, 1)$ a donc le type $\alpha_2 \times \texttt{int}$. Ensuite, l'application $+(\mathit{fst}\ x, 1)$ unifie les $\texttt{int} \times \texttt{int}$ et $\alpha_2 \times \texttt{int}$, ce qui donne $\{\alpha_2 \mapsto \texttt{int}\}$. Au final, on obtient le type $\texttt{int} \times \beta_1 \rightarrow \texttt{int}$, c'est-à-dire

$$\vdash \texttt{fun}\ x \rightarrow +(\mathit{fst}\ x, 1) : \texttt{int} \times \beta \rightarrow \texttt{int}$$

La variable de type restante, β_1, n'est plus une inconnue, mais une vraie variable de type, qu'on a ici renommée en β. Le pseudo-code de l'algorithme W est donné figure 5.7. On

$W(\Gamma,\ e) \stackrel{\text{def}}{=}$

- — si e est une variable x,
  renvoyer une instance triviale de $\Gamma(x)$
- — si e est une constante c,
  renvoyer une instance triviale de son type
- — si e est une primitive op,
  renvoyer une instance triviale de son type
- — si e est une paire $(e_1,\ e_2)$,
  calculer $\tau_1 = W(\Gamma,\ e_1)$
  calculer $\tau_2 = W(\Gamma,\ e_2)$
  renvoyer $\tau_1 \times \tau_2$
- — si e est une fonction $\texttt{fun}\ x \to e_1$,
  soit α une nouvelle variable
  calculer $\tau = W(\Gamma + x : \alpha,\ e_1)$
  renvoyer $\alpha \to \tau$
- — si e est une application $e_1\ e_2$,
  calculer $\tau_1 = W(\Gamma,\ e_1)$
  calculer $\tau_2 = W(\Gamma,\ e_2)$
  soit α une nouvelle variable
  $\mathit{unifier}(\tau_1,\ \tau_2 \to \alpha)$
  renvoyer α
- — si e est $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$,
  calculer $\tau_1 = W(\Gamma,\ e_1)$
  renvoyer $W(\Gamma + x : \mathit{Gen}(\tau_1,\ \Gamma),\ e_2)$

FIGURE 5.7 – Pseudo-code de l'algorithme W.

note que le cas d'une constante c implique une instance triviale. On peut en effet avoir des constantes polymorphes, comme par exemple la liste vide.

Théorème 3. L'algorithme W est correct, au sens où

$$\text{si } W(\emptyset, e) = \tau \text{ alors } \emptyset \vdash e : \tau,$$

et il détermine le type « le plus général possible », dit aussi *type principal*, au sens où

$$\text{si } \emptyset \vdash e : \tau \text{ alors } \tau \leq Gen(W(\emptyset, e),\ \emptyset).$$

□

Par ailleurs, on peut montrer que le système de Hindley-Milner est sûr, au sens du théorème 2, c'est-à-dire que si $\emptyset \vdash e : \tau$, alors la réduction de e est infinie ou se termine sur une valeur.

Le problème des références polymorphes. Supposons que l'on veuille ajouter à Mini-ML des références, comme en OCaml, avec une construction *ref* e pour construire une nouvelle référence, une construction $!e$ pour accéder au contenu d'une référence et une construction $e_1 := e_2$ pour modifier le contenu d'une référence. On pourrait penser qu'il suffit de voir ces trois constructions comme autant de nouvelles primitives, auxquelles on donnerait les types polymorphes suivants

$$\begin{aligned} \mathit{ref} :&\ \forall\alpha.\alpha \rightarrow \alpha\ \mathit{ref} \\ ! :&\ \forall\alpha.\alpha\ \mathit{ref} \rightarrow \alpha \\ := :&\ \forall\alpha.\alpha\ \mathit{ref} \rightarrow \alpha \rightarrow \mathit{unit} \end{aligned}$$

où τ *ref* est le type d'une référence contenant une valeur de type τ. Cependant, il s'avère que c'est incorrect, au sens où la sûreté du typage n'est plus garantie. Un contre-exemple est le programme suivant :

$$\begin{array}{l} \texttt{let}\ r = \mathit{ref}\ (\texttt{fun}\ x \rightarrow x)\ \texttt{in} \\ \texttt{let}\ _ = r := (\texttt{fun}\ x \rightarrow x + 1)\ \texttt{in} \\ !r\ (\texttt{fun}\ y \rightarrow y) \end{array}$$

Il est bien typé dans le système Hindley-Milner. En effet, la première ligne donne à r le type polymorphe $\forall\alpha.(\alpha \rightarrow \alpha)\ \mathit{ref}$. Ensuite, l'affectation est acceptée, car le type polymorphe de r peut être spécialisé en $(\texttt{int} \rightarrow \texttt{int})\ \mathit{ref}$. Enfin, la dernière ligne est également bien typée, car le type de r peut être spécialisé de nouveau, cette fois avec le type $((\texttt{int} \rightarrow \texttt{int}) \rightarrow (\texttt{int} \rightarrow \texttt{int}))\ \mathit{ref}$. Mais ce programme ne s'exécute pas correctement, car il aboutit à l'addition de la fonction `fun` $y \rightarrow y$ et de l'entier 1.

Ce problème, dit des *références polymorphes*, admet plusieurs solutions. L'une des plus simples consiste à ne généraliser le type de e_1 dans `let` $x = e_1$ `in` e_2 que lorsque e_1 est syntaxiquement une valeur [1]. Dès lors, on ne peut plus écrire

$$\texttt{let}\ r = \mathit{ref}\ (\texttt{fun}\ x \rightarrow x)\ \texttt{in}\ \dots$$

1. On appelle cela la *value restriction* en anglais.

car *ref* (fun $x \to x$) n'est pas une valeur, mais on peut écrire en revanche

$$(\texttt{fun}\ r \to \dots)\ (\mathit{ref}\ (\texttt{fun}\ x \to x))$$

où le type de r n'est pas généralisé. En pratique, on peut continuer d'écrire `let` r = *ref* ... `in` ... mais sans généraliser le type de r dans ce cas. C'est notamment ce que fait OCaml.

```
# let x = ref (fun x -> x);;
val x : ('_a -> '_a) ref
```

Ici, `'_a` désigne un type qu'on ne connaît pas encore et non pas une variable de type comme `'a` qui serait quantifiée. Si par exemple on applique `!x` à un entier, on détermine alors `'_a` comme étant `int` et on ne peut plus appliquer `!x` à autre chose que des entiers.

Exercice 36. Expliquer pourquoi le programme OCaml suivant est mal typé :

```
  let map_id = List.map (fun x -> x)
  let l1 = map_id [1; 2; 3]
  let l2 = map_id [true; false; true]
```

Proposer une solution. Solution □

Notes bibliographiques. Le type de preuve réalisée dans la section 5.2 a été introduit dans *A Syntactic Approach to Type Soundness* [29]. Le livre de Benjamin Pierce *Types and Programming Languages* [22] en contient plusieurs exemples. Le système F est dû indépendamment à J.-Y. Girard et J. C. Reynolds. Le résultat d'indécidabilité du typage dans le système F a été montré en 1994 par J. B. Wells [28]. L'algorithme W est dû à Damas et Milner [6]. Le problème des références polymorphes a été détecté en 1990 [26] et a nécessité des rectifications de versions de SML qui présentaient ce problème. La solution mise en place dans OCaml est relativement sophistiquée [9], notamment pour permettre d'indiquer quelles sont les variables de types d'un type abstrait qui peuvent être généralisées.

Troisième partie

Partie arrière du compilateur

6 Passage des paramètres

Dans ce chapitre, nous nous intéressons à la façon dont un appel de fonction est réalisé et notamment à la façon dont les paramètres sont transmis par l'appelant à l'appelé. Nous illustrons différentes stratégies en la matière, en prenant des exemples parmi des langages existants tels que C, OCaml, Java ou encore C++. Commençons par un peu de vocabulaire. Dans la *déclaration* d'une fonction `f` [1]

```
function f(x1, ..., xn) =
  ...
```

les variables `x1`, ..., `xn` sont appelées *paramètres formels* de `f`. Dans un *appel* à cette fonction

```
f(e1, ..., en)
```

les expressions `e1`, ..., `en` sont appelées *paramètres effectifs* de `f`. Si le langage comprend des modifications en place, une affectation

```
e1 := e2
```

modifie un emplacement mémoire désigné par l'expression `e1`, pour lui affecter la valeur de l'expression `e2`. Le plus souvent, l'expression `e1` est limitée à certaines constructions, car des affectations comme `42 := 17` ou encore `true := false` n'auraient pas de sens. On parle de *valeur gauche* (en anglais *left value*) pour désigner les expressions légales à gauche d'une affectation.

6.1 Stratégie d'évaluation

La stratégie d'évaluation d'un langage spécifie l'ordre dans lequel les calculs sont effectués. On peut la définir à l'aide d'une sémantique formelle, comme nous l'avons fait dans le chapitre 2, et le compilateur se doit de respecter cette stratégie. En particulier, la stratégie d'évaluation *peut* spécifier à quel moment les paramètres effectifs d'un appel sont évalués, d'une part, et l'ordre d'évaluation des opérandes et des paramètres effectifs, d'autre part. Certains aspects de l'évaluation peuvent cependant rester volontairement

1. Peu importe le langage pour le moment.

non spécifiés. Cela laisse alors de la latitude au compilateur pour effectuer des optimisations en ordonnançant stratégiquement les calculs.

Parmi les différentes stratégies d'évaluation, on distingue notamment l'*évaluation stricte* et l'*évaluation paresseuse*. Avec une évaluation stricte, les opérandes et paramètres effectifs sont évalués *avant* l'opération ou l'appel. Des langages comme C, C++, Java, OCaml ou encore Python ont adopté une évaluation stricte. Avec une évaluation paresseuse, au contraire, les opérandes et paramètres effectifs ne sont évalués que si nécessaire. En particulier, ils ne sont pas évalués avant l'appel. Des langages comme Haskell ou Clojure ont fait ce choix.

Un langage impératif adopte systématiquement une évaluation stricte, pour garantir une séquentialité des effets de bord qui coïncide avec le texte source. Par exemple, le code OCaml

```
let r = ref 0
let id x = r := !r + x; x
let f x y = !r
let () = print_int (f (id 40) (id 2))
```

affiche `42` car les deux arguments de `f` ont été évalués, même si la fonction `f` n'utilise aucun de ses deux arguments. Il serait difficile de comprendre l'effet de l'expression `f (id 40) (id 2)` s'il faut commencer par déterminer quels sont ceux de ses arguments que `f` utilise.

Dans les langages impératifs, une exception est faite pour les connectives logiques `&&` et `||`, qui n'évaluent leur seconde opérande que si nécessaire. Plus précisément, la connective `&&` (resp. `||`) n'évalue pas sa seconde opérande lorsque la première est fausse (resp. vraie), quand bien même cette seconde opérande aurait des effets. On peut ainsi écrire des morceaux de code tels que

```
while 0 < !j && v < a.(!j-1)
```

qui assurent qu'on n'évalue pas `a.(!j-1)` lorsque `!j` est nul.

Il est important de comprendre que la non-terminaison est également un effet. Ainsi, le programme OCaml

```
let rec loop () = loop ()
let f x y = x + 1
let v = f 41 (loop ())
```

ne termine pas, bien que l'argument `y` de `f` n'est pas utilisé.

Un langage purement applicatif, en revanche, peut adopter la stratégie d'évaluation de son choix, car une expression aura toujours la même valeur. On parle de *transparence référentielle*. En particulier, il peut faire le choix d'une évaluation paresseuse. Ainsi, le programme Haskell

```
loop () = loop ()
f x y = x
main = putChar (f 'a' (loop ()))
```

termine (après avoir affiché `a`).

Passage des paramètres. La stratégie d'évaluation d'un langage précise également le *mode de passage* des paramètres lors d'un appel. On distingue notamment l'*appel par*

valeur (en anglais *call by value*), l'*appel par référence* (en anglais *call by reference*), l'*appel par nom* (en anglais *call by name*) et l'*appel par nécessité* (en anglais *call by need*). On parle aussi parfois de *passage* par valeur, par référence, etc.

Dans l'appel par valeur, de *nouvelles* variables représentant les paramètres formels reçoivent les *valeurs* des paramètres effectifs [2]. Ainsi, un programme comme

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)
```

affiche `41`, car le paramètre `x` de `f` est une nouvelle variable, recevant la valeur `41`, et c'est cette variable qui est incrémentée. La variable `v`, quant à elle, reste égale à `41`, d'où le résultat.

Dans l'appel par référence, en revanche, les paramètres formels désignent les *mêmes* valeurs gauches que les paramètres effectifs. Ainsi, le même programme ci-dessus affiche `42` en appel par référence, car `x` désigne la même variable que `v`, qui se retrouve donc incrémentée.

Dans l'appel par nom, les paramètres effectifs sont *substitués* aux paramètres formels, textuellement, et donc évalués seulement si nécessaire. Ainsi, le programme

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0))
```

affiche `25`, en évaluant deux fois l'expression `1+2` et deux fois l'expression `2+2`. En revanche, l'expression `1/0` n'est jamais évaluée, ce qui explique que le programme n'échoue pas.

Enfin, dans l'appel par nécessité, les paramètres effectifs ne sont évalués que si nécessaire, mais *au plus une fois*. Ainsi, le même programme ci-dessus affiche toujours `25`, mais en évaluant une seule fois l'expression `1+2` et une seule fois l'expression `2+2`.

6.2 Comparaison des langages Java, OCaml, C et C++

Bien comprendre un langage de programmation, notamment pour bien l'utiliser, nécessite de bien comprendre sa stratégie d'évaluation. Dans cette section, nous détaillons les modes de passages des quatre langages que sont Java, OCaml, C et C++. Nous les présentons dans cet ordre car les modèles d'exécution de Java et OCaml sont plus simples que ceux de C et C++.

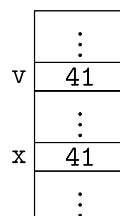
Java. Java est muni d'une stratégie d'évaluation stricte, avec appel *par valeur*. L'ordre d'évaluation est spécifié comme étant de la gauche vers la droite, c'est-à-dire dans l'ordre

2. Nous avons déjà évoqué l'appel par valeur dans le chapitre 2.

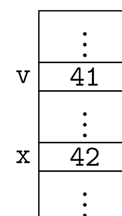
où sont écrits les paramètres[3]. Une valeur est soit d'un type primitif (booléen, caractère, entier machine, etc.), soit un pointeur vers un objet alloué sur le tas. Si on passe un entier à une fonction qui incrémente son argument, cet incrémentation ne sera pas observée par l'appelant. On le comprend en matérialisant l'appel par valeur sur une illustration comme celle-ci :

```
void f(int x) {
  x += 1;
}
int main() {
  int v = 41;
  f(v);
  // v vaut toujours 41
}
```

au début de l'appel



à la fin de l'appel

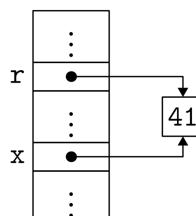


Ici, on a matérialisé les variables `v` et `x` comme allouées sur la pile, dans les tableaux d'activations respectifs des fonctions `main` et `f`[4]. Ces variables pourraient tout aussi bien être allouées dans des registres. Cela ne changerait en rien cet exemple.

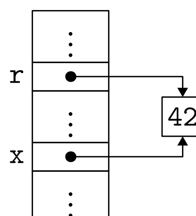
Lorsqu'un objet est passé en argument à une fonction, c'est toujours un appel par valeur, mais d'une valeur qui est maintenant un pointeur, même si celui-ci n'est pas explicite en Java. En voici une illustration :

```
class C { int f; }

void incr(C x) {
  x.f += 1;
}
void main () {
  C r = new C();
  r.f = 41;
  incr(r);
  // r.f vaut maintenant 42
}
```

au début de l'appel



à la fin de l'appel

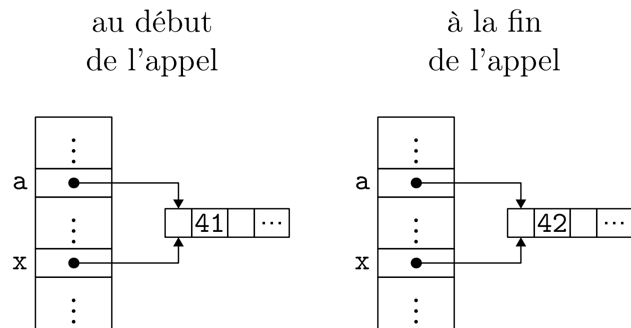


Cette fois, l'incrémentation est bien observée par l'appelant, car même si une nouvelle variable `x` a été créée, elle contenait seulement un pointeur vers le même objet que celui désigné par `r` et c'est dans cet objet que la modification a été effectuée. En Java, un tableau est un objet (presque) comme un autre, ce qui amène à une situation comparable :

3. Il est amusant de lire à ce propos dans *The Java Language Specification* la phrase « It is recommended that code not rely crucially on this specification ».

4. La pile croissant vers le bas, la variable `x` apparaît plus bas que la variable `v`.

```
void incr(int[] x) {
  x[1] += 1;
}
void main () {
  int[] a = new int[17];
  a[1] = 41;
  incr(a);
  // a[1] vaut maintenant 42
}
```



On peut *simuler l'appel par nom* en Java, en remplaçant les arguments par des fonctions sans argument [5]. Ainsi, la fonction

```
int f(int x, int y) {
  if (x == 0) return 42; else return y + y;
}
```

peut être réécrite sous la forme

```
int f(Supplier<Integer> x, Supplier<Integer> y) {
  if (x.get() == 0)
    return 42;
  else
    return y.get() + y.get();
}
```

et appelée comme ceci

```
int v = f(() -> 0, () -> { throw new Error(); });
```

Ce code n'échouera pas, car on ne cherche pas à appliquer la fonction `y` dans le cas où `x.get()` renvoie zéro. On a bien simulé un appel par nom.

Plus subtilement, on peut aussi *simuler l'appel par nécessité* en Java. On peut le faire avec un objet qui mémorise la valeur donnée par la fonction la première fois qu'elle est appelée.

5. Une fonction sans argument n'est ici qu'un objet de type `Supplier`, avec une méthode `get` pour l'appliquer.

```
class Lazy<T> implements Supplier<T> {
  private T cache = null;
  private Supplier<T> f;

  Lazy(Supplier<T> f) { this.f = f; }

  public T get() {
    if (this.cache == null) {
      this.cache = this.f.get();
      this.f = null; // permet au GC de récupérer f
    }
    return this.cache;
  }
}
```

Ce n'est rien d'autre que de la mémoïsation, sur une fonction n'ayant qu'une seule valeur. On utilise alors ainsi cette classe `Lazy` :

```
  int w = f(new Lazy<Integer>(() -> 1),
            new Lazy<Integer>(() -> { ...gros calcul... }));
```

OCaml. OCaml est muni d'une stratégie d'évaluation stricte, avec appel *par valeur*. L'ordre d'évaluation n'est pas spécifié. Une valeur est soit d'un type primitif (booléen, caractère, entier machine, etc.), soit un pointeur vers un bloc mémoire (tableau, enregistrement, constructeur non constant, etc.) alloué sur le tas en général [6]. Les valeurs gauches sont les éléments de tableaux et les champs d'enregistrements déclarés comme mutables. Une variable mutable, appelée une *référence* en OCaml, n'est qu'une valeur d'un type enregistrement `ref` prédéfini comme

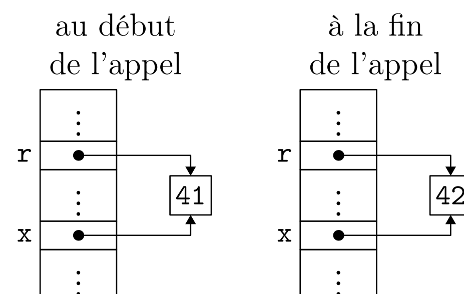
```
  type 'a ref = { mutable contents: 'a }
```

sur lequel sont définies des opérations d'accès ! et d'affectation :=, de la manière suivante :

```
  let (!)  r   = r.contents
  let (:=) r v = r.contents <- v
```

Lorsqu'on passe une référence en argument à une fonction, c'est une valeur qui est un pointeur qui est passée *par valeur*. En voici une illustration :

```
let incr x =
  x := !x + 1

let main () =
  let r = ref 41 in
  incr r
  (* !r vaut maintenant 42 *)
```



6. Il est frappant de noter à quel point les modèles d'exécution d'OCaml et Java sont proches, même si leurs langages de surface sont très différents.

Cette situation est tout à fait analogue à celle vue plus haut du passage d'un objet en Java. Il en va de même pour le passage d'un tableau en OCaml, la valeur d'un tableau étant un pointeur vers un bloc mémoire contenant les éléments du tableau.

Comme on l'a fait en Java, on peut *simuler l'appel par nom* en OCaml, en remplaçant les arguments par des fonctions. Ainsi, la fonction

```
let f x y =
  if x = 0 then 42 else y + y
```

peut être réécrite en

```
let f x y =
  if x () = 0 then 42 else y () + y ()
```

et appelée comme ceci

```
let v = f (fun () -> 0) (fun () -> failwith "oups")
```

On peut aussi *simuler l'appel par nécessité* en OCaml, en commençant par introduire un type pour représenter les calculs paresseux

```
type 'a value = Value  of 'a
              | Frozen of (unit -> 'a)

type 'a by_need = 'a value ref
```

et une fonction qui évalue un tel calcul si ce n'est pas déjà fait

```
let force l = match !l with
  | Value  v -> v
  | Frozen f -> let v = f () in l := Value v; v
```

On définit alors la fonction `f` légèrement différemment, pour forcer l'évaluation de ses arguments

```
let f x y =
  if force x = 0 then 42 else force y + force y
```

et on l'appelle ainsi

```
let v = f (ref (Frozen (fun () -> 1)))
          (ref (Frozen (fun () -> ...gros calcul...)))
```

La construction lazy d'OCaml fait quelque chose de semblable, mais un peu plus subtilement.

C. Le langage C est un langage impératif relativement bas niveau, notamment parce que la notion de pointeur, et d'arithmétique de pointeur, y est explicite. On peut le considérer inversement comme un assembleur de haut niveau. C'est là une excellente définition du langage C. Le langage C est muni d'une stratégie d'évaluation stricte, avec appel *par valeur*. L'ordre d'évaluation n'est pas spécifié.

Parmi les types du C, on trouve des types de base tels que `char`, `int`, `float`, etc. Il n'y a pas de type de booléens : toute valeur scalaire peut être utilisée comme un booléen et elle est vraie si et seulement si elle est non nulle. On trouve aussi un type τ`*` des pointeurs vers des valeurs de type τ. Si p est un pointeur de type τ`*`, alors $*p$ désigne la valeur

pointée par p, de type τ. Inversement, si e est une valeur gauche de type τ, alors `&`e est un pointeur sur l'emplacement mémoire correspondant, de type τ`*`. Enfin, on trouve des enregistrements, appelés *structures*, tels que

```
struct L { int head; struct L *next; };
```

Si `e` a le type `struct L`, on note `e.head` l'accès au champ.

En C, une valeur gauche est de trois formes possibles : une variable `x`, le déréférencement d'un pointeur `*e` ou l'accès à un champ de structure `e.x` si `e` est elle-même une valeur gauche. Par ailleurs, `t[e]` est du sucre syntaxique pour `*(t+e)` et `e->x` du sucre syntaxique pour `(*e).x`. Ce sont donc, syntaxiquement du moins, deux autres formes de valeurs gauches.

Le passage d'un entier par valeur ne diffère pas de celui en Java :

```
void f(int x) {
  x += 1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut toujours 41
}
```

	au début de l'appel		à la fin de l'appel
	⋮		⋮
v	41	v	41
	⋮		⋮
x	41	x	42
	⋮		⋮

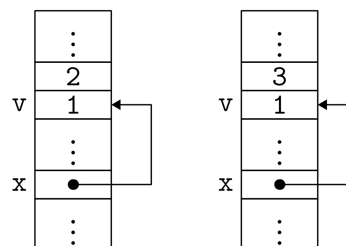
Le passage d'une structure, en revanche, nous donne une situation inédite jusqu'à présent, car une structure C *est une valeur* et est donc copiée lors d'un appel. En voici une illustration :

```
struct S { int a; int b; };

void f(struct S x) {
  x.b += 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b vaut toujours 2
}
```

	au début de l'appel		à la fin de l'appel
	⋮		⋮
	2		2
v	1	v	1
	⋮		⋮
	2		3
x	1	x	1
	⋮		⋮

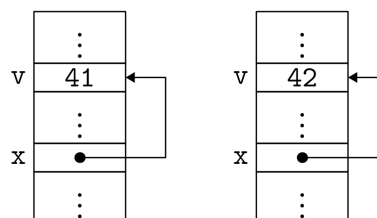
Les structures sont également copiées lorsqu'elles sont renvoyées avec `return` et lors d'affectations de structures, c'est-à-dire d'affectations de la forme `x = y`, où `x` et `y` ont le type `struct S`. Pour éviter le coût de telles copies, on manipule le plus souvent des pointeurs sur des structures, comme ceci :

```
struct S { int a; int b; };

void f(struct S *x) {
  x->b += 1;
}

int main() {
  struct S v = { 1, 2 };
  f(&v);
  // v.b vaut maintenant 3
}
```



C'est une façon de *simuler* un passage par référence, mais cela reste un passage par valeur d'une valeur qui est un pointeur. On peut faire de même avec un entier :

```
void incr(int *x) {
  *x += 1;
}

int main() {
  int v = 41;
  incr(&v);
  // v vaut maintenant 42
}
```



Une telle manipulation explicite de pointeurs peut être dangereuse. Considérons par exemple le programme

```
int* p() {
  int x;
  ...
  return &x;
}
```

Il renvoie un pointeur obtenu en prenant l'adresse de la variable locale `x`, c'est-à-dire un pointeur qui correspond à un emplacement sur la pile qui vient justement de disparaître (à savoir le tableau d'activation de `p`) et qui sera très probablement réutilisé rapidement par un autre tableau d'activation. On parle de référence fantôme (en anglais *dangling reference*).

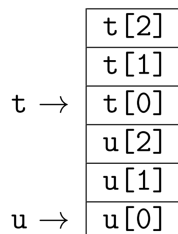
En C, les tableaux ne constituent pas vraiment un type en soi. Si on déclare un tableau de dix entiers, avec

```
int t[10];
```

la notation `t[i]` n'est que du sucre syntaxique pour `*(t+i)` où `t` désigne un pointeur sur le début d'une zone contenant 10 entiers et `+` désigne une opération d'*arithmétique de pointeur* (qui consiste à ajouter à `t` la quantité $4i$ pour un tableau d'entiers 32 bits). Le

premier élément du tableau est donc `t[0]` c'est-à-dire `*t`. Quand on passe un tableau en paramètre, on ne fait que passer le pointeur, toujours par valeur. On ne peut affecter des tableaux, seulement des pointeurs. Ainsi, on ne peut pas écrire

```
void p() {
  int t[3];
  int u[3];
  t = u; // refusé
}
```



car `t` et `u` sont des tableaux alloués sur la pile et l'affectation de tableaux n'est pas autorisée.

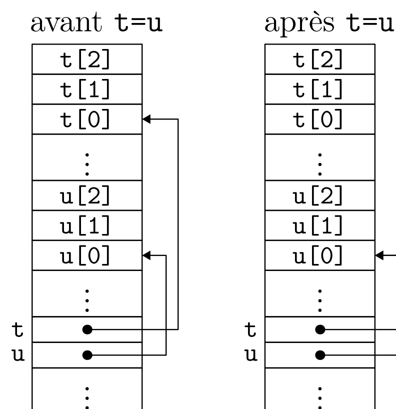
En revanche on peut écrire

```
void q(int t[3], int u[3]) {
  t = u;
}
```

car c'est exactement la même chose que

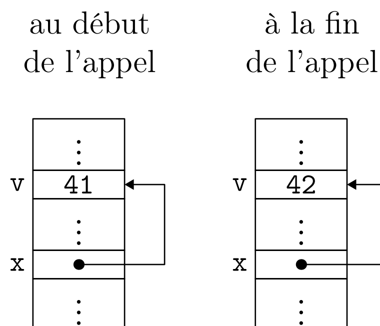
```
void q(int *t, int *u) {
  t = u;
}
```

et l'affectation de pointeurs est autorisée.



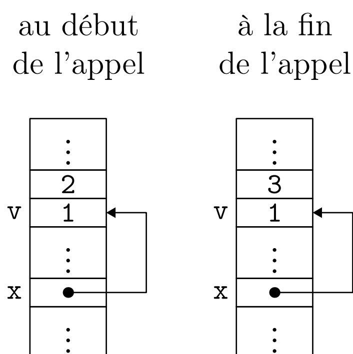
C++. En C++, on trouve (entre autres) les types et constructions du C, avec une stratégie d'évaluation stricte. Le mode de passage est *par valeur* par défaut. Mais on trouve aussi un passage *par référence* indiqué par le symbole `&` au niveau de l'argument formel. Ainsi, on peut écrire

```
void f(int &x) {
  x += 1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut maintenant 42
}
```



En particulier, c'est le compilateur qui a pris l'adresse de `v` au moment de l'appel, d'une part, et a déréférencé l'adresse `x` dans la fonction `f`, d'autre part. De même, on peut passer une structure par référence, ce qui est une alternative à l'utilisation explicite de pointeurs comme en C.

```
struct S { int a; int b; };

void f(struct S &x) {
  x.b += 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b vaut maintenant 3
}
```



On peut aussi passer un pointeur par référence. Un exemple pertinent est celui de l'insertion d'un élément dans un arbre.

```
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
  if       (t == NULL ) t = create(NULL, x, NULL);
  else if (x < t->elt) add(t->left,  x);
  else if (x > t->elt) add(t->right, x);
}
```

Ainsi, on traite élégamment le cas de l'insertion dans un arbre vide.

Résumé. Résumons les différentes situations que nous avons rencontrées avec ces quatre langages au regard du passage d'un entier à une fonction :

	v 41 … x 41	v 41 … x • (vers v)	r • … x • (vers 41)
C	entier par valeur	pointeur par valeur	pointeur par valeur
OCaml	entier par valeur	—	pointeur par valeur (par ex. type `ref`)
Java	entier par valeur	—	pointeur par valeur (objet)
C++	entier par valeur	pointeur par valeur entier par référence	pointeur par valeur ou par référence

On note en particulier que la situation illustrée dans la colonne centrale, correspondant au passage d'un pointeur vers l'emplacement d'une variable, n'a pas de réalité dans les langages OCaml et Java.

Notes bibliographiques. L'ouvrage *Le langage C* de Brian Kernighan et Dennis Ritchie [15, 14] reste une référence pour apprendre ce langage.

```
E  ::= n                              expression arithmétique
     | x
     | E + E | E - E
     | E * E | E / E
     | - E

C  ::= E = E | E <> E                 expression booléenne
     | E < E | E <= E
     | E > E | E >= E
     | C and C
     | C or C
     | not C

S  ::= x:= E                          instruction
     | if C then S
     | if C then S else S
     | while C do S
     | p(E, ..., E)
     | B

B  ::= begin S; ...; S end            bloc

D  ::= var x, ..., x: integer;        déclaration (locale ou globale)
     | procedure p(F; ...; F);
         D ... D B;

F  ::= x: integer                     paramètre formel
     | var x: integer

P  ::= program x; D ... D B.          programme
```

FIGURE 6.1 – Syntaxe abstraite de mini Pascal.

6.3 Compilation d'un mini Pascal

Pour bien comprendre le passage des paramètres, d'un point de vue de la compilation, écrivons un compilateur pour un petit fragment du langage Pascal. L'intérêt de ce langage, ici, est de proposer à la fois des procédures imbriquées et du passage par valeur et par référence. La syntaxe abstraite de ce mini Pascal est donnée figure 6.1. Les variables y sont notées x et les procédures p. Les expressions sont limitées à des expressions entières, de l'unique type `integer`. Un programme est formé d'une suite de déclarations, suivie d'un bloc qui constitue le point d'entrée. Une déclaration est soit la déclaration de plusieurs variables de type `integer`, non initialisées, soit la déclaration d'une procédure. Une procédure est constituée d'une suite de déclarations locales, suivie d'un bloc qui constitue le corps de la procédure. En particulier, une procédure peut être déclarée localement à une autre procédure.

```
program fib;
var f : integer;

procedure fib(n : integer);
   procedure somme();
   var tmp : integer;
   begin fib(n-2); tmp := f;
         fib(n-1); f := f + tmp end;
begin
   if n <= 1 then f := n else somme()
end;

begin fib(3); writeln(f) end.
```

FIGURE 6.2 – Un programme mini Pascal.

La stratégie d'évaluation est stricte, sauf pour les constructions `and` et `or`. Le mode de passage est par valeur pour un paramètre déclaré avec la syntaxe x : `integer` et par référence pour un paramètre déclaré avec la syntaxe `var` x : `integer`. On suppose l'existence d'une procédure prédéfinie `writeln` pour afficher un entier. Les figures 6.2 et 6.3 contiennent deux programmes mini Pascal, qui calculent respectivement la suite de Fibonacci et la suite de Syracuse.

Les procédures pouvant être imbriquées, il convient de définir soigneusement la *portée* des variables et des procédures. On dit qu'une procédure p est le *parent* d'un identificateur y si y est déclaré dans p, soit comme un paramètre de p, soit comme une variable ou une procédure locale de p. On dit que p est un *ancêtre* de y si p est soit y soit le parent d'un ancêtre de y. On dit que la déclaration d *précède* la déclaration d' si elles ont le même parent (ou sont toutes les deux globales) et que d vient avant d' dans le programme. Par ailleurs, on définit le *niveau* d'une déclaration ou d'un bloc de code comme le nombre de procédures sous lesquelles elle est déclarée. En particulier, les déclarations globales et le programme principal ont le niveau 0.

On peut alors définir la portée de la manière suivante : si le corps d'une procédure p mentionne un identificateur alors celui-ci est soit une déclaration locale de p, soit un ancêtre de p (y compris p lui-même), soit une déclaration précédant un ancêtre de p. En particulier, une procédure peut être récursive mais il n'y a pas de procédures mutuellement récursives.

Venons-en maintenant à la compilation de mini Pascal. On adopte ici un schéma de compilation simple où toutes les variables sont sur la pile. Les paramètres sont situés en haut du tableau d'activation, placés là par l'appelant, et les variables locales plus bas, dans la partie allouée par l'appelé. La difficulté ici repose sur le fait qu'une variable x qui est utilisée n'est pas nécessairement située dans le tableau d'activation qui se trouve en sommet de pile. En effet, les règles de portée font que cette variable peut avoir été déclarée dans la procédure parent, ou plus généralement dans un ancêtre quelconque de la procédure en cours d'exécution. Il faut donc savoir retrouver le tableau d'activation où se trouve cette variable. Mais avant d'expliquer comment, persuadons-nous qu'il existe

```
program syracuse;

procedure syracuse(max : integer);
var i : integer;
   procedure length();
   var v,j : integer;
      procedure step();
      begin
        v := v+1; if j = 2*(j/2) then j := j/2 else j := 3*j+1
      end;
   begin
     v := 0; j := i; while j <> 1 do step(); writeln(v)
   end;
begin
   i := 1;
   while i <= max do begin length(); i := i+1 end
end;

begin syracuse(100) end.  { affiche 0 1 7 2 5 ... }
```

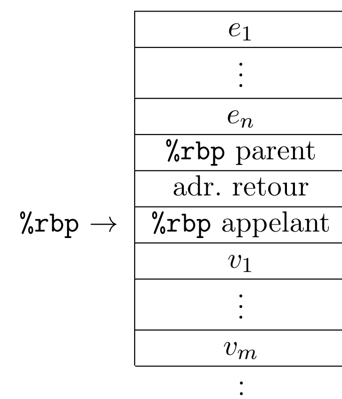
FIGURE 6.3 – Un autre programme mini Pascal.

bien. On dit qu'une procédure est *active* si on n'a pas encore fini d'exécuter le corps de cette procédure. Un résultat essentiel à la compilation de mini Pascal est le suivant :

Proposition 5. Lorsqu'une procédure p est active, alors tous les ancêtres de p sont des procédures actives.

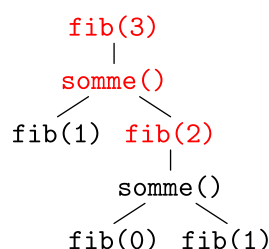
PREUVE. La preuve se fait par récurrence sur la profondeur de p dans l'arbre d'activation. C'est vrai pour le programme principal, qui n'a que lui-même comme ancêtre. Si p a été activée par une procédure q alors q est toujours active (par définition) et tous les ancêtres de q sont actifs par hypothèse de récurrence ; or les règles de visibilité impliquent que soit q est le parent de p, soit p précède un ancêtre de q, et dans les deux cas tous les ancêtres de p sont bien actifs. □

Ce résultat implique donc que toute variable utilisée se situe dans un tableau d'activation qui se trouve quelque part sur la pile. Pour être en mesure de le retrouver, on va placer dans chaque tableau d'activation un pointeur vers la tableau d'activation de la procédure parent. Ainsi, il suffira de suivre ces pointeurs, un nombre de fois égal à la différence de niveaux, pour retrouver le tableau d'activation d'une variable donnée. Le tableau d'activation correspondant à un appel $p(e_1, \dots, e_n)$ est illustré ci-contre. La partie supérieure est construite par l'appelant. Elle contient les valeurs des arguments $e_1, \dots, e_n$, ainsi que l'adresse vers le tableau d'activation de la procédure parent. Vient ensuite l'adresse de

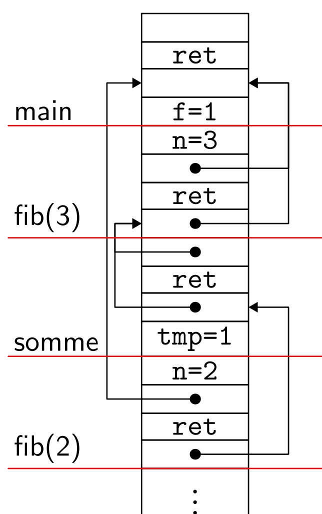


retour, placée par l'instruction `call`. Enfin, la partie inférieure est construite par l'appelé. Elle contient la sauvegarde du registre `%rbp` et les variables locales $v_1, \ldots, v_m$ de p.

Prenons l'exemple du programme `fib` de la figure 6.2 et supposons que l'on soit en train de calculer `fib(3)`. La procédure `fib` va faire un appel à `somme()`, qui va appeler successivement `fib(1)` et `fib(2)`. Dans le premier cas, on ne fera pas d'autre appel ; dans le second, on va appeler `somme()` de nouveau, qui appellera `fib(0)` puis `fib(1)`. On peut représenter ces différents appels par un *arbre d'activation*, où chaque nœud correspond à un appel de procédure $p(e_1, \ldots, e_n)$ et les sous-nœuds correspondent aux appels directement effectués par $p(e_1, \ldots, e_n)$. Voici l'arbre d'activation pour `fib(3)` :



À chaque instant, les tableaux d'activation sur la pile correspondent à un chemin depuis la racine dans l'arbre d'activation. Si on prend le chemin `fib(3)`—`somme()`—`fib(2)` ici colorié en rouge, alors on a quatre tableaux d'activation sur la pile, organisés de la manière suivante :



Les différents tableaux sont séparés par une ligne rouge. Le tableau le plus haut, tout au fond de la pile, correspond au programme principal qui a appelé `fib(3)`. Comme il n'a ni parent ni appelant, on a deux cases non significatives dans son tableau, ici représentées comme vides. La variable globale `f` est représentée comme une variable locale du programme principal et donc matérialisée dans ce premier tableau. Vient ensuite le tableau de `fib(3)`, avec l'argument `n` et deux pointeurs vers l'appelant et le parent qui coïncident ici. De même, le troisième tableau contient deux pointeurs égaux, ainsi qu'une variable locale `tmp`. Le quatrième tableau, celui de l'appel à `fib(2)`, est plus intéressant car le tableau de la procédure parent (`main`) ne coïncide plus cette fois avec celui de la procédure appelante (`somme`). En particulier, si le code de `fib` a besoin d'accéder à la

variable `f`, il la retrouve en suivant le pointeur vers le tableau de la procédure parent, comme on le constate ici.

Expliquons maintenant comment compiler les différentes constructions de mini Pascal vers l'assembleur x86-64. On suppose connaître le niveau x_l de chaque variable x, ainsi que la position x_o relative à `%rbp` où se trouve x dans son tableau d'activation. On suppose de même connaître le niveau p_l d'une procédure p. On suppose pour l'instant que tous les arguments sont passés par valeur. Commençons par les expressions, dont on va calculer la valeur sur 64 bits signés. La compilation d'une expression arithmétique E dépend du niveau l où se trouve cette expression et nous la notons $Comp_l(E)$. C'est un morceau d'assembleur dont l'exécution doit mettre la valeur de E dans le registre `%rdi`. Le cas d'une constante est immédiat :

$$Comp_l(n) = \texttt{movq \$}n\texttt{, \%rdi}$$

Le cas d'une variable x est plus intéressant. On trouve le tableau contenant x en suivant les pointeurs vers le tableau parent un nombre de fois égal à $l - x_l$. On écrit donc

$$\begin{array}{rl} Comp_l(x) = & \texttt{movq \%rbp, \%rsi} \\ & \text{répéter } l - x_l \text{ fois } \texttt{movq 16(\%rsi), \%rsi} \\ & \texttt{movq } x_o\texttt{(\%rsi), \%rdi} \end{array}$$

Dans le cas particulier où $x_l = l$, c'est-à-dire si x se trouve dans le tableau d'activation courant, on peut simplifier en `movq` x_o`(%rbp), %rdi`. Restent les opérations arithmétiques. On ne va pas chercher à faire d'allocation de registres ici. On se sert donc de la pile pour stocker les résultats intermédiaires. Par exemple, l'addition peut se compiler ainsi :

$$\begin{array}{rl} Comp_l(E_1 + E_2) = & Comp_l(E_1) \\ & \texttt{pushq \%rdi} \\ & Comp_l(E_2) \\ & \texttt{popq \%rsi} \\ & \texttt{addq \%rsi, \%rdi} \end{array}$$

Il en va de même pour les autres opérations arithmétiques. Bien entendu, c'est extrêmement naïf. La compilation d'une expression comme `1+2` requiert cinq instructions et utilise la pile, alors même qu'on dispose de seize registres. On peut imaginer de nombreuses façons de compiler plus efficacement, même sans chercher à faire de l'allocation de registres, mais ce n'est pas le propos ici [7]. On compile de même les expressions booléennes, par exemple vers les entiers 0 pour faux et 1 pour vrai. La seule subtilité se trouve dans le caractère paresseux des opérateurs `and` et `or`, qui ne doivent pas évaluer leur seconde opérande dans certains cas. Par exemple, on compile ainsi l'opérateur `and` :

$$\begin{array}{rl} Comp_l(C_1 \texttt{ and } C_2) = & Comp_l(C_1) \\ & \texttt{testq \%rdi, \%rdi} \\ & \texttt{jz } L \\ & Comp_l(C_2) \\ & L\texttt{:} \end{array}$$

7. Nous verrons plus loin, dans le chapitre 9, comment compiler efficacement les expressions arithmétiques.

Ici, L désigne une étiquette fraîche. Venons-en à la compilation des instructions. Certaines ne posent pas de difficulté particulière, telles que la conditionnelle, la boucle `while` ou encore un bloc d'instructions. La compilation d'une affectation réutilise le calcul de la position de la variable que nous avons fait plus haut :

$Comp_l(x$ `:=` $E) =$ $Comp_l(E)$
`movq %rbp, %rsi`
répéter $l - x_l$ fois `movq 16(%rsi), %rsi`
`movq %rdi,` x_o`(%rsi)`

La dernière instruction à compiler est le cas d'un appel de procédure. On commence par empiler les valeurs des n arguments, puis le pointeur vers le tableau de la procédure parent, toujours calculé de la même façon, avant de faire `call`. On termine en dépilant la place occupée par les n arguments et le pointeur, soit $8(n+1)$ octets.

$Comp_l(p(E_1, \ldots, E_n)) =$ $Comp_l(E_1)$ `pushq %rdi`
⋮
$Comp_l(E_n)$ `pushq %rdi`
`movq %rbp, %rsi`
répéter $l - p_l$ fois `movq 16(%rsi), %rsi`
`pushq %rsi`
`call` p
`addq $`$8(n+1)$`, %rsp`

Reste enfin à expliquer comment compiler les *déclarations* de toutes les procédures. On compile chaque procédure indépendamment et de la même façon, qu'elle soit déclarée globablement ou localement à une autre procédure. Une procédure p de niveau l est compilée ainsi :

$Comp(p(x_1, \ldots, x_n) \ldots B) =$ p:
`pushq %rbp`
`movq %rsp, %rbp`
`subq $`$8m$`, %rsp`
$Comp_{l+1}(B)$
`movq %rbp, %rsp`
`popq %rbp`
`ret`

Ici, l'entier m désigne le nombre de variables locales à la procédure p. On note que le corps B de la procédure p est une instruction compilée au niveau $l + 1$. Le programme principal peut être considéré comme une procédure de niveau -1, les variables globales étant alors autant de variables locales de cette procédure (comme illustré plus haut avec le programme `fib`).

Passage par référence. Pour l'instant, on a supposé tous les paramètres passés *par valeur*. Mais le qualificatif `var` au niveau d'un paramètre formel permet de spécifier un passage *par référence* et il faut alors modifier la compilation en conséquence. Dans le cas d'un passage par référence, le paramètre effectif doit être une valeur gauche. Dans mini Pascal, cela se limite donc à une variable[8]. Lors d'un appel de procédure $p(e_1, \ldots, e_n)$,

8. Dans un langage plus complet, cela pourrait être un élément de tableau, un champ de structure, etc.

si le paramètre i est passé par référence, il faut donc vérifier d'une part que e_i est bien une variable et compiler cet argument différemment lors de l'appel, pour passer l'adresse de cette variable et non plus sa valeur. Une façon élégante de procéder consiste à ajouter une construction de « calcul de valeur gauche » dans la syntaxe des expressions. Notons-la `&`x par analogie avec le langage C. On suppose que le typage a introduit cette nouvelle construction dans les appels de procédures, au niveau de chaque paramètre passé par référence. Commençons par expliquer comment compiler cette nouvelle construction :

$Comp_l($`&`$x) =$ `movq %rbp, %rdi`
répéter $l - x_l$ fois `movq 16(%rdi), %rdi`
`addq $`x_o`, %rdi`
si x passée par référence, ajouter `movq (%rdi), %rdi`

La dernière ligne tient compte du cas d'une variable x elle-même passée par référence, qu'on est donc en train de repasser par référence à une autre procédure. Il faut aussi modifier la compilation de l'accès à une variable pour traiter le cas d'une variable passée par référence :

$Comp_l(n) =$ `movq %rbp, %rsi`
répéter $l - x_l$ fois `movq 16(%rsi), %rsi`
`movq` x_o`(%rsi), %rdi`
si x passée par référence, ajouter `movq (%rdi), %rdi`

On constate qu'on a seulement ajouté la dernière ligne. Enfin, il faut modifier l'affectation, toujours pour traiter le cas d'une variable passée par référence :

$Comp_l(x$ `:=` $E) =$ $Comp_l(E)$
`movq %rbp, %rsi`
répéter $l - x_l$ fois `movq 16(%rsi), %rsi`
si x passée par référence, `movq (%rsi),%rsi` sinon `addq $`x_o`,%rsi`
`movq %rdi, (%rsi)`

En revanche, il n'y a rien à modifier dans l'appel de procédure, grâce à la nouvelle construction `&`. De même qu'il n'y a rien à modifier dans la compilation de la déclaration d'une procédure.

Exercice 37. Discuter de la possibilité d'avoir des procédures mutuellement récursives.
Solution □

7 Compilation des langages fonctionnels

Dans ce chapitre, on s'intéresse à divers aspects de la compilation en rapport avec les langages fonctionnels. Aujourd'hui, beaucoup d'idées venant des langages fonctionnels ont fait leur chemin dans des langages de paradigmes très différents *a priori*. Ainsi, on trouve maintenant des fonctions de première classe en Java et en C++. Dans tout ce chapitre, on prend des fragments du langage OCaml à des fins d'illustration, mais il faut donc garder à l'esprit que les concepts ne sont pas liés à un langage particulier.

7.1 Fonctions comme valeurs de première classe

Ce qui caractérise les langages de programmation fonctionnels, c'est la possibilité de manipuler les fonctions comme des valeurs de première classe, c'est-à-dire exactement comme des valeurs d'un autre type, par exemple des entiers. Ainsi, on peut recevoir une fonction en argument, la renvoyer comme un résultat, la stocker dans une liste ou un tableau, la faire transporter par une exception, etc.

Considérons le fragment simple d'OCaml donné figure 7.1, dans lequel les expressions contiennent des fonctions anonymes introduites par `fun`, des déclarations locales, éventuellement récursives, introduites par `let` et des conditionnelles. Un programme est une suite de déclarations, éventuellement récursives [1]. Ce fragment d'OCaml est très proche du langage Mini-ML que nous avons considéré dans les chapitres 2 et 5. Dans ce fragment, on peut écrire des programmes où les fonctions sont arbitrairement imbriquées, comme

```
let somme n =
  let f x = x * x in
  let rec boucle i = if i = n then 0 else f i + boucle (i+1) in
  boucle 0
```

Il ne s'agit pas encore de fonctions de première classe. D'ailleurs, imbriquer ainsi les fonctions n'est pas un trait des langages fonctionnels et existe depuis longtemps dans des langages comme Algol, Pascal ou Ada [2]. Sur cet exemple, il est important de noter que les

1. On ne considère pas ici de fonctions mutuellement récursives, mais cela n'ajouterait pas de vraie difficulté.

2. Le compilateur `gcc` propose une extension non standard où les fonctions peuvent être imbriquées.

$$
\begin{array}{rcl}
e & ::= & c \\
  & | & x \\
  & | & \texttt{fun}\ x \rightarrow e \\
  & | & e\ e \\
  & | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
  & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & & \\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
 & & \\
p & ::= & d\ \ldots\ d
\end{array}
$$

FIGURE 7.1 – Un noyau fonctionnel d'OCaml.

fonctions locales peuvent faire référence à des variables introduites précédemment, dans la portée desquelles elles se trouvent. Ainsi, la fonction `boucle` fait référence à `n`, l'argument de la fonction `somme`, et à la fonction `f` introduite juste avant.

Il est également possible de prendre des fonctions en argument, comme dans cet exemple

```
let carré f x =
  f (f x)
```

et d'en renvoyer, comme dans cet autre exemple

```
let f x =
  if x < 0 then fun y -> y - x else fun y -> y + x
```

Dans ce dernier cas, la valeur renvoyée par `f` est une fonction qui utilise `x` mais le tableau d'activation de `f` vient précisément de disparaître. Il va donc falloir compiler une telle fonction de manière à ce que la valeur de `x` survive à cet appel.

La solution consiste à utiliser une *fermeture* (en anglais *closure*). Il s'agit d'une structure de données allouée sur le tas (pour survivre aux appels de fonctions) contenant d'une part un *pointeur vers le code* de la fonction à appeler et d'autre part les valeurs des variables susceptibles d'être utilisées par ce code. Cette seconde partie de la fermeture s'appelle l'*environnement*. Pour une fermeture représentant $\texttt{fun}\ x \rightarrow e$, les variables dont il faut enregistrer la valeur dans l'environnement sont exactement les *variables libres* de $\texttt{fun}\ x \rightarrow e$ (voir définition 1 page 21).

Considérons l'exemple de cette fonction récursive `pow` qui calcule $\texttt{x}^{\texttt{i}}$ pour un flottant `x` et un entier `i`.

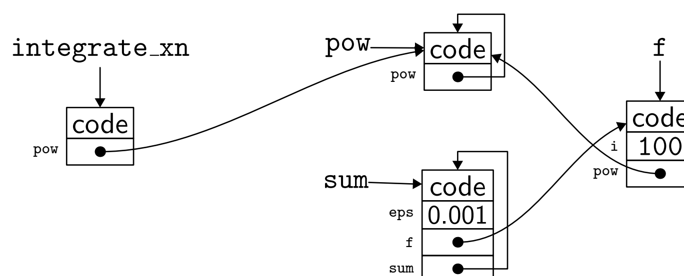
```
let rec pow = fun i -> fun x ->
  if i = 0 then 1. else x *. pow (i-1) x
```

Il y a deux constructions `fun`, correspondant donc à deux fermetures. Dans la première fermeture, `fun i ->`, l'environnement est réduit à $\{\texttt{pow}\}$, car les variables `i` et `x` sont liées. Dans la seconde fermeture, `fun x ->`, l'environnement est $\{\texttt{i}, \texttt{pow}\}$. Considérons une seconde fonction, `integrate_xn`, qui utilise la fonction `pow` pour calculer une approximation de $\int_0^1 x^n dx$.

```
let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum = fun x ->
    if x >= 1. then 0. else f x +. sum (x +. eps) in
  sum 0. *. eps
```

On a là encore deux fermetures. Pour la première, l'environnement est {`pow`} et pour la seconde, l'environnement est {`eps`, `f`, `sum`}.

Une fermeture peut être matérialisée par un unique bloc sur le tas [3], contenant l'adresse du code dans son premier champ et les valeurs de l'environnement dans les champs suivants. Avec cette représentation, les différentes fermetures en jeu pendant l'exécution d'un appel à `integrate_xn 100` ressemblent à ceci :



Les deux fonctions `integrate_xn` et `pow` sont des fermetures. Une fois `integrate_xn` appelée, l'application de `pow` à `100` renvoie une fonction, et donc une fermeture, stockée dans la variable `f`. Enfin, la fonction `sum` est une quatrième fermeture. On note en particulier que les fermetures des fonctions `pow` et `sum` contiennent leur propre valeur dans leur environnement, parce qu'il s'agit de fonctions récursives. Ainsi, l'exécution du code d'une fermeture retrouve dans l'environnement la valeur de toute variable, sans faire de cas particulier pour une fonction récursive. C'est pendant la construction de la fermeture qu'il faut prendre soin de la récursivité.

Expliquons maintenant comment mécaniser la construction des fermetures et comment les utiliser. Une façon relativement simple de compiler les fermetures consiste à procéder en deux temps. On commence par rechercher dans le code toutes les constructions `fun` $x \to e$ et on les remplace par une opération explicite de construction de fermeture

$$\texttt{clos}\ f\ [y_1, \ldots, y_n]$$

où les y_i sont les variables libres de `fun` $x \to e$ et f le nom donné à une déclaration globale de fonction de la forme

$$\texttt{letfun}\ f\ [y_1, \ldots, y_n]\ x = e'$$

où e' est obtenu à partir de e en y supprimant récursivement les constructions `fun`. Cette explicitation des fermetures s'appelle *closure conversion* en anglais. Sur notre exemple, l'explicitation des fermetures donne le programme suivant :

3. D'autres solutions sont possibles, comme par exemple un environnement alloué dans un second bloc ou encore sous forme de liste chaînée. Mais la solution utilisant un unique bloc est plus efficace en pratique, car sollicitant moins le GC.

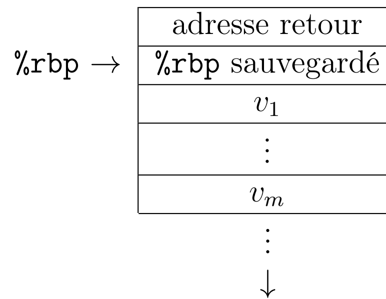
$$
\begin{array}{rcl}
e & ::= & c \\
  & | & x \\
  & | & \texttt{clos}\ f\ [x,\ldots,x] \\
  & | & e\ e \\
  & | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
  & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
  & & \\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
  & | & \texttt{letfun}\ f\ [x,\ldots,x]\ x = e \\
  & & \\
p & ::= & d\ \ldots\ d
\end{array}
$$

FIGURE 7.2 – Explicitation des fermetures.

```
letfun fun2 [i,pow] x =
  if i = 0 then 1. else x *. pow (i-1) x
letfun fun1 [pow] i =
  clos fun2 [i,pow]
let rec pow =
  clos fun1 [pow]
letfun fun3 [eps,f,sum] x =
  if x >= 1. then 0. else f x +. sum (x +. eps)
letfun fun4 [pow] n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum = clos fun3 [eps,f,sum] in
  sum 0. *. eps
let integrate_xn =
  clos fun4 [pow]
```

On note qu'on a toujours les deux déclarations globales de `pow` et `integrate_xn`, dont les valeurs sont des fermetures. Les quatre fonctions `fun1`, `fun2`, `fun3` et `fun4` introduites par cette traduction peuvent être placées arbitrairement dans le code, car elles sont indépendantes les unes des autres. Elles ne sont pas non plus récursives. En effet, une fermeture dispose dans son environnement de toute valeur dont elle a besoin. Ainsi, la fonction `fun2` trouve dans son environnement la valeur de `pow` pour procéder à un appel récursif. Le langage cible de cette transformation est donné figure 7.2. On note que chaque fonction introduite par `letfun` a exactement un argument.

Dans un second temps, on compile le code obtenu. Adoptons le schéma de compilation où chaque fonction introduite par `letfun` reçoit son unique argument dans `%rdi` et sa fermeture dans `%rsi`. Il n'y a pas d'autre argument et donc en particulier pas d'argument passé par la pile. Celle-ci est utilisée en revanche pour allouer les variables locales introduites par `let`. On a donc un tableau d'activation de la forme



où $v_1, \ldots, v_m$ sont les variables locales. Détaillons la compilation des diverses constructions. Pour compiler la construction

$$\texttt{clos}\ f\ [y_1, \ldots, y_n]$$

on procède ainsi. On alloue un bloc de taille $n + 1$ sur le tas, avec une fonction de type `malloc`. On stocke l'adresse de f dans le premier champ. Cette adresse est connue, comme celle de la fonction f dans le programme compilé. On stocke les valeurs des variables $y_1, \ldots, y_n$ dans les champs 1 à n du bloc. On expliquera un peu plus loin comment se fait l'accès à la valeur d'une variable. On renvoie l'adresse du bloc comme valeur de l'expression. On se repose sur un GC pour libérer ce bloc lorsque ce sera possible. Il est en effet difficile, et en toute généralité impossible, de décider à quel moment une fermeture peut être libérée.

Expliquons maintenant comment compiler un appel, c'est-à-dire une expression de la forme $e_1\ e_2$. On compile e_1 dans le registre `%rsi`. Puisqu'il s'agit d'une fonction (ce que le typage statique garantit), on sait que sa valeur est l'adresse d'une fermeture. On compile e_2 dans le registre `%rdi`. Enfin, on appelle la fonction dont l'adresse est contenue dans le premier champ de la fermeture, avec `call *(%rsi)`. Il s'agit donc d'un saut à une *adresse calculée*.

Expliquons maintenant comment accéder à la valeur d'une variable x. C'est plus subtil qu'il n'y paraît, car on doit distinguer quatre cas. S'il s'agit d'une variable globale, introduite par un `let` au niveau global, sa valeur se trouve par exemple dans le segment de donnée. S'il s'agit d'une variable locale, introduite par un `let-in`, sa valeur se trouve sur la pile, et on y accède par n(`%rbp`) pour un certain n. S'il s'agit d'une variable contenue dans la fermeture, on y accède par n(`%rsi`) pour un certain n. Enfin, s'il s'agit de l'argument d'une fonction `letfun`, sa valeur se trouve dans `%rdi`. Cette distinction de cas peut être déjà faite dans l'arbre de syntaxe abstraite à l'issue de l'explicitation des fermetures.

Enfin, on compile une déclaration de fonction `letfun` $f\ [y_1, \ldots, y_n]\ x = e$ de manière usuelle. On alloue le tableau d'activation, dans lequel on sauvegarde `%rbp` avant de le positionner. On évalue l'expression e dans `%rax`. Puis on restaure `%rbp` et on désalloue le tableau d'activation, avant de faire `ret`.

La compilation des autres constructions, à savoir une constante, une déclaration locale et une conditionnelle, est tout à fait classique et indépendante du reste.

Optimisations. Il est inutilement coûteux de créer des fermetures intermédiaires dans un appel où tous les arguments sont fournis. Un appel « traditionnel » peut être fait, où tous les arguments sont passés d'un coup. En revanche, une application partielle de la même fonction doit produire une fermeture. Le compilateur OCaml, par exemple, fait cette optimisation. Sur du code ne faisant pas usage de fonctions comme valeurs de première classe, on obtient donc la même efficacité qu'avec un langage non fonctionnel.

Une autre optimisation est possible lorsque l'on est sûr qu'une fermeture ne survivra pas à la fonction dans laquelle elle est créée. Elle peut être alors allouée sur la pile plutôt que sur le tas. C'est le cas par exemple de la fermeture pour `f` dans

```
let integrate_xn n =
  let f = ... in
```

Mais pour s'assurer que cette optimisation est possible, il faut effectuer une analyse statique non triviale, dite d'échappement (en anglais *escape analysis*).

Fermetures dans d'autres langages. On trouve aujourd'hui des fermetures dans des langages comme Java (depuis 2014 et Java 8) ou C++ (depuis 2011 et C++11). Dans ces langages, les fonctions anonymes sont appelées des *lambdas*, en écho à la notion de λ-abstraction. En Java, une fonction est un objet comme un autre, avec une méthode `apply`. On peut écrire par exemple

```
  LinkedList<B> map(LinkedList<A> l, Function<A, B> f) {
    ... f.apply(x) ...
  }
```

où `Function` est une interface prédéfinie pour un tel objet. Une fonction anonyme est introduite avec `->`

```
  map(l, x -> { System.out.print(x); return x+y; })
```

Le compilateur construit un objet fermeture, qui capture ici `y`, avec une méthode `apply`.

En C++, une fonction anonyme est introduite avec `[]` :

```
  for_each(v.begin(), v.end(), [y](int &x){ x += y; });
```

On spécifie les variables capturées dans la fermeture, ici `y`. Par défaut, les variables sont capturées par valeur, mais on peut spécifier une capture par référence (ici de `s`) :

```
  for_each(v.begin(), v.end(), [y,&s](int x){ s += y*x; });
```

Si un langage ne propose pas de fermetures, on peut les construire manuellement une fois qu'on en a compris le principe. Ainsi, avant Java 8, il était tout à fait possible de construire ponctuellement une classe représentant une fermeture, avec les valeurs de l'environnement dans des champs et une méthode de type `apply` pour le code de la fonction. La construction d'un tel objet est moins agréable qu'en utilisant la syntaxe fournie par Java 8 mais le résultat est identique. De même, le langage C ne propose pas de fermetures mais il est tout à fait possible d'introduire au cas par cas une structure contenant un pointeur de fonction et un environnement, accompagnée d'une fonction `apply`.

7.2 Optimisation des appels terminaux

Les langages fonctionnels encouragent un style de programmation récursive. Lorsque le langage est purement applicatif, c'est même la seule chose que l'on puisse faire pour écrire des boucles. Par conséquent, un soin particulier est apporté dans les compilateurs des langages fonctionnels pour que la récursivité soit aussi efficace que l'usage d'une boucle, et en particulier qu'elle ne provoque pas de débordement de pile, autant que possible. Pour cela, une optimisation de certains appels de fonctions, dits *terminaux*, est faite

par le compilateur. Aujourd'hui, une telle optimisation est faite dans une majorité de compilateurs, indépendamment de la nature du langage considéré.

Définition 22. On dit qu'un *appel* à une fonction f qui apparaît dans le corps d'une fonction g est *terminal* (en anglais *tail call*) si c'est la dernière chose que g calcule avant de renvoyer son résultat.

Par extension, on dit qu'une fonction est *récursive terminale* (en anglais *tail recursive function*) s'il s'agit d'une fonction récursive dont tous les appels récursifs sont des appels terminaux. □

Voici un exemple d'appel terminal à une fonction `f` dans une fonction `g` :

```
let g x =
  let y = x * x in f y
```

L'appel `f y` est bien la dernière chose que `g` calcule. Une fois le résultat de `f y` obtenu, il est directement transmis comme le résultat de `g`. Dans une fonction récursive, on peut avoir des appels récursifs terminaux et d'autres qui ne le sont pas, comme dans la très célèbre fonction 91 de McCarthy :

```
let rec f91 n =
  if n <= 100 then f91 (f91 (n + 11)) else n - 10
```

L'intérêt d'un appel terminal du point de vue de la compilation est que l'on peut détruire le tableau d'activation de la fonction g où se trouve l'appel *avant* de faire l'appel à f, puisqu'il ne servira plus ensuite. Mieux encore, on peut le réutiliser pour l'appel terminal que l'on doit faire. En particulier, l'adresse de retour qui s'y trouve est la bonne. Dit autrement, on peut faire un saut avec `jump` plutôt qu'un appel avec `call`. Considérons par exemple le programme suivant qui calcule la factorielle de `n` multipliée par `acc` :

```
let rec fact acc n =
  if n <= 1 then acc else fact (acc * n) (n - 1)
```

Une compilation classique donne le programme assembleur de gauche alors qu'en optimisant l'appel terminal, on obtient le programme de droite :

```
fact:
        cmpq    $1, %rsi
        jle     L0
        imulq   %rsi, %rdi
        decq    %rsi
        call    fact
        ret
L0:     movq    %rdi, %rax
        ret
```

```
fact:
        cmpq    $1, %rsi
        jle     L0
        imulq   %rsi, %rdi
        decq    %rsi
        jmp     fact

L0:     movq    %rdi, %rax
        ret
```

Le résultat est une *boucle*. Le code est en effet identique à ce qu'aurait donné la compilation d'un programme C tel que

```
while (n > 1) {
  acc = acc * n;
  n   = n - 1;
}
```

et ce, bien qu'on n'ait pas nécessairement de traits impératifs dans le langage considéré. On pourrait tout à fait être en train de compiler un langage purement applicatif.

Le programme obtenu avec cette optimisation est plus efficace. D'une part, on accède moins à la mémoire, car on n'utilise plus `call` et `ret` qui manipulent la pile. Mais surtout, l'espace de pile utilisé devient constant. En particulier, on évite ainsi tout débordement de pile (en anglais *stack overflow*) qui serait dû à un trop grand nombre d'appels imbriqués.

Il est important de noter que la notion d'appel terminal n'a rien à voir avec les langages fonctionnels. Sa compilation peut être optimisée dans tous les langages. Ainsi, `gcc` fait cette optimisation si on lui passe l'option de ligne de commande `-foptimize-sibling-calls` (incluse dans l'option `-O2`). Il est également important de retenir que la notion d'appel terminal n'est pas liée à la récursivité, même si c'est le plus souvent une fonction récursive qui fera déborder la pile et donc pour laquelle on souhaite une telle optimisation.

Application. Supposons que l'on cherche à écrire en OCaml une fonction qui calcule la hauteur d'un arbre, pour un type d'arbres binaires défini de cette façon :

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

Il est naturel d'écrire un code de la forme

```
let rec height = function
| Empty          -> 0
| Node (l, _, r) -> 1 + max (height l) (height r)
```

mais celui-ci va provoquer un débordement de pile sur un arbre de grande hauteur. En effet, les deux appels récursifs à `height` ne sont pas terminaux, car il faut encore calculer le maximum et ajouter un une fois ces deux appels terminés.

Pour éviter le débordement de pile, cherchons à écrire la fonction `height` en utilisant uniquement des appels terminaux. Au lieu de calculer la hauteur h de l'arbre, calculons $k(h)$ pour une fonction k quelconque, appelée *continuation*. La fonction `height` aura donc le type suivant :

```
val height: 'a tree -> (int -> 'b) -> 'b
```

On appelle cela la *programmation par continuations* (en anglais *continuation-passing style*, abrégé en CPS). Le programme voulu s'en déduira avec la continuation identité, c'est-à-dire `height t (fun h -> h)`. Le code prend alors la forme suivante

```
let rec height t k = match t with
| Empty ->
    k 0
| Node (l, _, r) ->
    height l (fun hl ->
    height r (fun hr ->
    k (1 + max hl hr)))
```

On constate que tous les deux appels à `height` et les deux appels à `k` sont *terminaux*. Le calcul de `height` se fait donc en espace de pile constant. On a remplacé l'espace sur la pile par de l'espace *sur le tas*. Il est occupé par les fermetures. La première fermeture capture `r` et `k`, la seconde `hl` et `k`.

Bien sûr, il y a d'autres solutions, ad hoc, pour calculer la hauteur d'un arbre sans faire déborder la pile, par exemple un parcours en largeur. De même qu'il y a d'autres solutions si le type d'arbres est plus complexe : arbres mutables, hauteur stockée dans le nœud, pointeurs parents, etc. Mais la solution à base de CPS a le mérite d'être mécanique. On trouvera plus de détails sur cet exemple dans l'article *Mesurer la hauteur d'un arbre* [8].

Exercice 38. Que faire si le langage optimise l'appel terminal mais ne propose pas de fonctions anonymes (par exemple, le langage C) ? Solution □

7.3 Filtrage

Dans les langages fonctionnels, on trouve généralement une construction appelée *filtrage* (en anglais *pattern matching*), utilisée dans les définitions de fonctions, comme

$$\texttt{function}\ p_1 \to e_1 \mid \dots \mid p_n \to e_n,$$

les conditionnelles généralisées, comme

$$\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid \dots \mid p_n \to e_n,$$

et les gestionnaires d'exceptions, comme

$$\texttt{try}\ e\ \texttt{with}\ p_1 \to e_1 \mid \dots \mid p_n \to e_n.$$

Le compilateur transforme ces constructions de haut niveau en séquences de *tests élémentaires* (tests de constructeurs et comparaisons de valeurs constantes) et d'accès à des champs de valeurs structurées. Nous allons expliquer ce processus de compilation. Dans ce qui suit, on considère la construction

$$\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \dots \mid p_n \to e_n$$

à laquelle il est aisé de se ramener avec un `let`. Commençons par définir ce que représentent les p_i ci-dessus.

Définition 23. Un *motif* (en anglais *pattern*) est défini par la syntaxe abstraite

$$p ::= x \mid C(p, \dots, p)$$

où x est une variable et C est un *constructeur*. □

Si on prend l'exemple du langage OCaml, un constructeur peut être une constante (telle que `false`, `true`, 0, 1, `"hello"`, etc.), un constructeur constant de type algébrique (tel que `[]` ou par exemple `Empty` déclaré par `type t = Empty | ...`), un constructeur d'arité $n \geq 1$ (tel que `::` ou par exemple `Node` déclaré par `type t = Node of t * t | ...`) ou encore un constructeur de n-uplet, avec $n \geq 2$.

Définition 24 (motif linéaire)**.** On dit qu'un motif p est *linéaire* si toute variable apparaît au plus une fois dans p. □

Ainsi, le motif (x, y) est linéaire, mais (x, x) ne l'est pas. Dans ce qui suit, on ne considère que des motifs linéaires[4]. Les valeurs filtrées sont construites à partir du même ensemble de constantes et de constructeurs que dans la définition des motifs, c'est-à-dire

$$v ::= C(v, \dots, v)$$

On commence par définir la notion de filtrage d'une valeur par un unique motif.

Définition 25 (filtrage)**.** On dit qu'une valeur v *filtre* un motif p s'il existe une substitution σ de variables par des valeurs telle que $v = \sigma(p)$. □

On peut supposer de plus que le domaine de σ, c'est-à-dire l'ensemble des variables x telles que $\sigma(x) \neq x$, est inclus dans l'ensemble des variables de p. Il est clair que toute valeur filtre $p = x$. D'autre part, on a le résultat suivant.

Proposition 6. Une valeur v filtre $p = C(p_1, \dots, p_n)$ si et seulement si v est de la forme $v = C(v_1, \dots, v_n)$ avec v_i qui filtre p_i pour tout $i = 1, \dots, n$.

PREUVE. Soit v qui filtre p. On a donc $v = \sigma(p)$ pour un certain σ, soit $v = C(\sigma(p_1), \dots, \sigma(p_n))$, et il suffit de poser $v_i = \sigma(p_i)$.

Réciproquement, si v_i filtre p_i pour tout i, alors il existe des σ_i telles que $v_i = \sigma_i(p_i)$. Comme p est linéaire, les domaines des σ_i sont deux à deux disjoints et on a donc $\sigma_i(p_j) = p_j$ si $i \neq j$. En posant $\sigma = \sigma_1 \circ \cdots \circ \sigma_n$, on a

$$\begin{aligned}\sigma(p_i) &= \sigma_1(\sigma_2(\dots\sigma_n(p_i))\dots) \\ &= \sigma_1(\sigma_2(\dots\sigma_i(p_i))\dots) \\ &= \sigma_1(\sigma_2(\dots v_i)\dots) \\ &= v_i\end{aligned}$$

et donc $\sigma(p) = C(\sigma(p_1), \dots, \sigma(p_n)) = C(v_1, \dots, v_n) = v$. □

On définit maintenant la sémantique de la construction `match`.

Définition 26 (filtrage à plusieurs cas)**.** Dans le filtrage

$$\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \dots \mid p_n \to e_n$$

si v est la valeur de x, on dit que v filtre le cas p_i si v filtre p_i et si v ne filtre aucun p_j pour tout $j < i$. Le résultat du filtrage est alors $\sigma(e_i)$, où σ est la substitution telle que $\sigma(p_i) = v$. Si en revanche v ne filtre aucun p_i, le filtrage conduit à une erreur[5]. □

Un premier algorithme. Pour écrire un algorithme de compilation du filtrage, on suppose disposer d'une fonction $constr(e)$ qui renvoie le constructeur de la valeur e et d'une fonction $\#_i(e)$ qui renvoie la i-ième composante de la valeur e. Autrement dit, si $e = C(v_1, \dots, v_n)$ alors $constr(e) = C$ et $\#_i(e) = v_i$.

On commence par la compilation d'une ligne de filtrage

$$code(\texttt{match}\ e\ \texttt{with}\ p \to action) = F(p, e, action)$$

4. OCaml n'admet les motifs non linéaires que dans les motifs OU tels que `let x,0 | 0,x = ...`. On ne considère pas les motifs OU ici.

5. l'exception `Match_failure` dans le cas d'OCaml.

où la fonction de compilation F est définie ainsi :

$$
\begin{array}{l}
F(x, e, action) = \\
\quad \texttt{let } x = e \texttt{ in } action \\
F(C, e, action) = \\
\quad \texttt{if } constr(e) = C \texttt{ then } action \texttt{ else } error \\
F(C(p), e, action) = \\
\quad \texttt{if } constr(e) = C \texttt{ then } F(p, \#_1(e), action) \texttt{ else } error \\
F(C(p_1, \ldots, p_n), e, action) = \\
\quad \texttt{if } constr(e) = C \texttt{ then} \\
\quad\quad F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), action) \ldots) \\
\quad \texttt{else } error
\end{array}
$$

Considérons par exemple

```
match x with 1 :: y :: z -> y + length z
```

Sa compilation donne le (pseudo-)code suivant :

```
if constr(x) = :: then
    if constr(#1(x)) = 1 then
      if constr(#2(x)) = :: then
        let y = #1(#2(x)) in
        let z = #2(#2(x)) in
        y + length(z)
      else error
    else error
  else error
```

On note que $\#_2$`(x)` est calculée plusieurs fois. On pourrait introduire des `let` dans la définition de F pour y remédier. On peut montrer la correction de cet algorithme.

Proposition 7. Si $e \xrightarrow{\star} v$ alors

$$
\begin{array}{ll}
F(p, e, action) \xrightarrow{\star} \sigma(action) & \text{s'il existe } \sigma \text{ telle que } v = \sigma(p), \\
F(p, e, action) \xrightarrow{\star} error & \text{sinon.}
\end{array}
$$

Preuve. On procède par récurrence sur p.

- si $p = x$ ou $p = C$, c'est immédiat.
- si $p = C(p_1, \ldots, p_n)$:
  - si $constr(v) \neq C$, il n'existe pas de σ telle que $v = \sigma(p)$ et $F(C(p_1, ..., p_n), e, action) = error$.
  - si $constr(v) = C$, on a $v = C(v_1, \ldots, v_n)$ et σ telle que $v = \sigma(p)$ existe si et seulement s'il existe des σ_i telles que $v_i = \sigma_i(p_i)$. Si l'une des σ_i n'existe pas, alors l'appel $F(p_i, \#_i(e), \ldots)$ se réduit en $error$ et $F(p, e, action)$ également. Si en revanche toutes les σ_i existent, alors par hypothèse de récurrence

$$
\begin{array}{rcl}
F(p, e, action) & = & F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), action) \ldots) \\
& = & F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots \sigma_n(action)) \ldots) \\
& = & \sigma_1(\sigma_2(\ldots \sigma_n(action) \ldots)) \\
& = & \sigma(action)
\end{array}
$$

□

Pour filtrer plusieurs lignes, on remplace *error* par le passage à la ligne suivante, c'est-à-dire

$$\begin{aligned} &code(\texttt{match}\ x\ \texttt{with}\ p_1 \rightarrow e_1 \mid \ \ldots \ \mid p_n \rightarrow e_n) = \\ &\quad F(p_1, x, e_1, F(p_2, x, e_2, \ldots F(p_n, x, e_n, error) \ldots)) \end{aligned}$$

où la fonction de compilation F a maintenant quatre arguments et est définie par

$$\begin{aligned} &F(x, e, succès, échec) = \\ &\quad \texttt{let}\ x = e\ \texttt{in}\ succès \\ &F(C, e, succès, échec) = \\ &\quad \texttt{if}\ constr(e) = C\ \texttt{then}\ succès\ \texttt{else}\ échec \\ &F(C(p_1, \ldots, p_n), e, succès, échec) = \\ &\quad \texttt{if}\ constr(e) = C\ \texttt{then} \\ &\quad\quad F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), succès, échec) \ldots, échec) \\ &\quad \texttt{else}\ échec \end{aligned}$$

La compilation de

```
match x with [] -> 1 | 1 :: y -> 2 | z :: y -> z
```

donne le code suivant

```
    if constr(x) = [] then
       1
     else
       if constr(x) = :: then
         if constr(#1(x)) = 1 then
           let y = #2(x) in 2
         else
           if constr(x) = :: then
             let z = #1(x) in let y = #2(x) in z
           else error
       else
         if constr(x) = :: then
           let z = #1(x) in let y = #2(x) in z
         else error
```

Comme on le constate, cet algorithme est peu efficace car on effectue plusieurs fois les mêmes tests (d'une ligne sur l'autre) et on effectue des tests redondants (si $constr(e) \neq$ `[]` alors nécessairement $constr(e) =$ `::`). On va se tourner vers un algorithme plus efficace.

Un meilleur algorithme. Pour obtenir un meilleur résultat, on va considérer le problème du filtrage simultané de m valeurs par n lignes de motif dans sa globalité. On le représente sous la forme d'une *matrice*

$$\left| \begin{array}{llllll} e_1 & e_2 & \ldots & e_m & & \\ p_{1,1} & p_{1,2} & \ldots & p_{1,m} & \rightarrow & action_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ p_{n,1} & p_{n,2} & \ldots & p_{n,m} & \rightarrow & action_n \end{array} \right|$$

dont la signification est

$$\begin{array}{l}\texttt{match}\ (e_1, e_2, \ldots, e_m)\ \texttt{with} \\ \mid (p_{1,1}, p_{1,2}, \ldots, p_{1,m}) \rightarrow action_1 \\ \mid \ldots \\ \mid (p_{n,1}, p_{n,2}, \ldots, p_{n,m}) \rightarrow action_n\end{array}$$

L'algorithme de compilation, noté F, procède récursivement sur la matrice. Un cas de base correspond à un filtrage sans aucun motif, c'est-à-dire $n = 0$. On a alors

$$F \left| \begin{array}{ccc} e_1 & \ldots & e_m \\ & & \end{array} \right| = error$$

Un autre cas de base correspond à un filtrage sans valeurs à filtrer, c'est-à-dire $m = 0$. On a alors

$$F \left| \begin{array}{cc} \rightarrow & action_1 \\ & \vdots \\ \rightarrow & action_n \end{array} \right| = action_1$$

Lorsque $n > 0$ et $m > 0$, on va se ramener à des matrices plus petites. Si toute la colonne de gauche se compose de *variables* $x_{i,1}$, c'est-à-dire

$$M = \left| \begin{array}{llllll} e_1 & e_2 & \ldots & e_m & & \\ x_{1,1} & p_{1,2} & \ldots & p_{1,m} & \rightarrow & action_1 \\ \vdots & & & & & \\ x_{n,1} & p_{n,2} & \ldots & p_{n,m} & \rightarrow & action_n \end{array} \right|$$

on élimine cette colonne en introduisant des `let`

$$F(M) = F \left| \begin{array}{lllll} e_2 & \ldots & e_m & & \\ p_{1,2} & \ldots & p_{1,m} & \rightarrow & \texttt{let}\ x_{1,1} = e_1\ \texttt{in}\ action_1 \\ \vdots & & & & \\ p_{n,2} & \ldots & p_{n,m} & \rightarrow & \texttt{let}\ x_{n,1} = e_1\ \texttt{in}\ action_n \end{array} \right|$$

Sinon, c'est que la colonne de gauche contient au moins un motif construit. Supposons par exemple qu'il y ait dans cette colonne trois constructeurs différents, C d'arité 1, D d'arité 0 et E d'arité 2.

$$M = \left| \begin{array}{llllll} e_1 & e_2 & \ldots & e_m & & \\ C(q) & p_{1,2} & \ldots & p_{1,m} & \rightarrow & action_1 \\ D & p_{2,2} & & p_{2,m} & \rightarrow & action_2 \\ x & p_{3,2} & & p_{3,m} & \rightarrow & action_3 \\ E(r,s) & p_{4,2} & & p_{4,m} & \rightarrow & action_4 \\ y & p_{5,2} & & p_{5,m} & \rightarrow & action_5 \\ C(t) & p_{6,2} & & p_{6,m} & \rightarrow & action_6 \\ E(u,v) & p_{7,2} & \ldots & p_{7,m} & \rightarrow & action_7 \end{array} \right|$$

Pour chaque constructeur C, D et E, on construit la sous-matrice correspondant au filtrage d'une valeur pour ce constructeur. Pour le constructeur C, on construit la matrice

$$M_C = \left| \begin{array}{lllll} \#_1(e_1) & e_2 & \dots & e_m & \\ q & p_{1,2} & \dots & p_{1,m} & \rightarrow \ action_1 \\ _ & p_{3,2} & & p_{3,m} & \rightarrow \ \texttt{let}\ x = e_1\ \texttt{in}\ action_3 \\ _ & p_{5,2} & & p_{5,m} & \rightarrow \ \texttt{let}\ y = e_1\ \texttt{in}\ action_5 \\ t & p_{6,2} & \dots & p_{6,m} & \rightarrow \ action_6 \end{array} \right|$$

Pour le constructeur D, on construit la matrice

$$M_D = \left| \begin{array}{llll} e_2 & \dots & e_m & \\ p_{2,2} & & p_{2,m} & \rightarrow \ action_2 \\ p_{3,2} & & p_{3,m} & \rightarrow \ \texttt{let}\ x = e_1\ \texttt{in}\ action_3 \\ p_{5,2} & \dots & p_{5,m} & \rightarrow \ \texttt{let}\ y = e_1\ \texttt{in}\ action_5 \end{array} \right|$$

Pour le constructeur e, on construit la matrice

$$M_E = \left| \begin{array}{llllll} \#_1(e_1) & \#_2(e_1) & e_2 & \dots & e_m & \\ _ & _ & p_{3,2} & & p_{3,m} & \rightarrow \ \texttt{let}\ x = e_1\ \texttt{in}\ action_3 \\ r & s & p_{4,2} & & p_{4,m} & \rightarrow \ action_4 \\ _ & _ & p_{5,2} & & p_{5,m} & \rightarrow \ \texttt{let}\ y = e_1\ \texttt{in}\ action_5 \\ u & v & p_{7,2} & \dots & p_{7,m} & \rightarrow \ action_7 \end{array} \right|$$

Enfin, on définit une sous-matrice pour les autres valeurs (de constructeurs différents de C, D et E), c'est-à-dire pour les variables apparaissant dans la première colonne.

$$M_R = \left| \begin{array}{llll} e_2 & \dots & e_m & \\ p_{3,2} & & p_{3,m} & \rightarrow \ \texttt{let}\ x = e_1\ \texttt{in}\ action_3 \\ p_{5,2} & \dots & p_{5,m} & \rightarrow \ \texttt{let}\ y = e_1\ \texttt{in}\ action_5 \end{array} \right|$$

Avec ces quatre matrices, on pose alors

$$\begin{array}{rl} F(M) = & \texttt{case}\ constr(e_1)\ \texttt{in} \\ & \quad C \Rightarrow F(M_C) \\ & \quad D \Rightarrow F(M_D) \\ & \quad E \Rightarrow F(M_E) \\ & \quad \texttt{otherwise} \Rightarrow F(M_R) \end{array}$$

Ici, `case` est une construction élémentaire, utilisée pour comparer $constr(e_1)$ avec C, D et E. Lorsqu'il n'y a que deux constructeurs dans le type de e_1, on peut utiliser un simple `if then else`. Lorsqu'il y a un nombre fini de constructeur, on peut utiliser une table de sauts. Lorsqu'il y a une infinité de constructeurs (par exemple, des chaînes de caractères), on peut utiliser un arbre binaire ou une table de hachage pour réaliser `case`. Enfin, il n'y a parfois qu'un seul constructeur (par exemple, un n-uplet) et alors $F(M) = F(M_C)$ directement.

Il est important de se persuader que cet algorithme termine. C'est bien le cas, car la grandeur

$$\sum_{i,j} taille(p_{i,j})$$

diminue strictement à chaque appel récursif à F, si on définit la taille d'un motif de la façon suivante :

$$\begin{array}{rcl} \mathit{taille}(x) & = & 1 \\ \mathit{taille}(C(p_1, \ldots, p_n)) & = & 1 + \sum_{i=1}^{n} \mathit{taille}(p_i) \end{array}$$

Considérons de nouveau l'exemple

```
match x with [] -> 1 | 1 :: y -> 2 | z :: y -> z
```

c'est-à-dire la matrice

$$M = \left| \begin{array}{lcl} x & & \\ \texttt{[]} & \rightarrow & 1 \\ 1::y & \rightarrow & 2 \\ z::y & \rightarrow & z \end{array} \right|$$

On obtient au final

```
case constr(x) in
  [] -> 1
  :: -> case constr(#1(x)) in
          1 -> let y = #2(x) in 2
          otherwise -> let z = #1(x) in let y = #2(x) in z
```

ce qui est bien meilleur qu'avec l'algorithme précédent. En particulier, il n'y a plus ici de calculs redondants.

Ce nouvel algorithme présente des avantages supplémentaires. Il permet par exemple de détecter les *cas redondants*. Il suffit en effet de remarquer qu'une action n'apparaît pas dans le code produit dans déterminer que le cas de filtrage correspondant est redondant et émettre alors un avertissement. Cet algorithme permet également de détecter des *filtrages non exhaustifs*. Il suffit en effet de vérifier si *error* apparaît dans le code produit.

Notes bibliographiques. La notion de fermeture a été inventée par P. J. Landin en 1964 [17]. La plupart des compilateurs Java n'optimisent malheureusement pas l'appel terminal, pour des raisons techniques liées à la JVM. On doit à Luc Maranget de nombreux articles sur le filtrage [18, 20, 21].

8 Compilation des langages à objets

Les langages à objets sont apparus dans les années 1960, avec Simula I et Simula 67, puis se sont développés avec Smalltalk (1972), puis C++ (1983) ou encore Java (1995). Dans ce chapitre, nous expliquons ce qu'est un langage à objets et comment il peut être compilé, c'est-à-dire notamment comment matérialiser un objet en mémoire représenté et comment réaliser un appel de méthode. Nous utilisons principalement Java à des fins d'illustrations, puis C++ en fin de chapitre pour souligner quelques différences notables entre les deux.

8.1 Brève présentation des concepts objets avec Java

Le premier concept objet est celui de *classe*. La déclaration d'une classe introduit un nouveau type. En toute première approximation, une classe peut être vue comme un enregistrement.

```
class Polar {
  double rho;
  double theta;
}
```

Ici `rho` et `theta` sont les deux *champs* de la classe `Polar`, de type `double`. On crée une *instance* particulière d'une classe, appelée un *objet*, avec la construction `new`. Ainsi,

```
  Polar p = new Polar();
```

déclare une nouvelle variable locale `p`, de type `Polar`, dont la valeur est une nouvelle instance de la classe `Polar`. L'objet est alloué sur le tas. Ses champs recoivent ici des valeurs par défaut (en l'occurrence 0). On peut accéder aux champs de `p`, et les modifier, avec la notation usuelle `p.rho` et `p.theta` :

```
  p.rho = 2;
  p.theta = 3.14159265;
  double x = p.rho * Math.cos(p.theta);
  p.theta = p.theta / 2;
```

On peut introduire un ou plusieurs *constructeurs*, dans le but d'initialiser les champs de l'objet. Un constructeur porte le même nom que la classe.

```
class Polar {
  double rho, theta;
  Polar(double r, double t) {
    if (r < 0) throw new Error("Polar: negative length");
    rho = r;
    theta = t;
  }
}
```

On peut alors écrire [1] par exemple

```
  Polar p = new Polar(2, 3.14159265);
```

Supposons maintenant que l'on veuille maintenir l'*invariant* suivant pour tous les objets de la classe `Polar` :

$$0 \leq \texttt{rho} \quad \wedge \quad 0 \leq \texttt{theta} < 2\pi.$$

Pour cela on déclare les champs `rho` et `theta` *privés*, de sorte qu'ils ne sont plus visibles à l'extérieur de la classe `Polar`.

```
class Polar {
  private double rho, theta;
  Polar(double r, double t) { /* garantit l'invariant */ }
}
```

Si on tente maintenant d'accéder au champ `rho` depuis une autre classe

```
  p.rho = 1;
```

on obtient une erreur de typage :

```
complex.java:19: rho has private access in Polar
```

La valeur du champ `rho` peut néanmoins être fournie par l'intermédiaire d'une *méthode*, c'est-à-dire d'une fonction fournie par la classe `Polar` et applicable à tout objet de cette classe.

```
class Polar {
  private double rho, theta;
  ...
  double norm() { return rho; }
}
```

Pour un objet `p` de type `Polar`, on appelle la méthode `norm` ainsi :

`p.norm()`

On peut le voir naïvement comme l'appel `norm(p)` d'une fonction qui prendrait l'objet en argument, comme ceci :

```
  double norm(Polar x) { return x.rho; }
```

Les objets remplissent donc un premier rôle d'*encapsulation*.

---

1. Une fois un constructeur introduit explicitement, le constructeur par défaut ne prenant pas d'argument disparaît. On ne pourrait plus écrire maintenant `new Polar()` comme nous l'avions fait juste avant.

Il est possible de déclarer un champ comme *statique* et il est alors lié à la classe et non aux instances de cette classe. Dit autrement, il s'apparente à une variable globale.

```
class Polar {
  double rho, theta;
  static double two_pi = 6.283185307179586;
```

De même, une *méthode* peut être *statique* et elle s'apparente alors à une fonction traditionnelle. En voici un exemple :

```
  static double normalize(double x) {
    while (x < 0) x += two_pi;
    while (x >= two_pi) x -= two_pi;
    return x;
  }
}
```

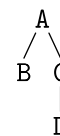
Ce qui n'est pas statique est appelé *dynamique*.

Le second concept objet est celui d'*héritage* : une classe `B` peut être définie comme héritant d'une classe `A`, avec la syntaxe

```
class B extends A { ... }
```

Les objets de la classe `B` héritent alors de tous les champs et méthodes de la classe `A`, auxquels ils peuvent ajouter de nouveaux champs et de nouvelles méthodes. La notion d'héritage s'accompagne d'une notion de *sous-typage* : toute valeur de type `B` peut être vue comme une valeur de type `A`. En Java, chaque classe hérite d'au plus une classe. On appelle cela l'*héritage simple*, par opposition à l'héritage multiple. La relation d'héritage forme donc une arborescence.

```
class A { ... }
class B extends A { ... }
class C extends A { ... }
class D extends C { ... }
```



La classe `A` est appelée la *super classe* de `B` et `C`. De même, `C` est la super classe de `D`. La relation de sous-typage est naturellement transitive. Ainsi, toute valeur de type `D` peut être utilisée comme une valeur de type `A`.

Illustrons l'utilité de l'héritage avec l'exemple classique d'éléments graphiques (des rectangles, des cercles, etc.). On commence par introduire une classe `Graphical` pour représenter n'importe quel élément graphique, avec une position centrale et des dimensions horizontale et verticale.

```
class Graphical {
  int x, y;          /* centre */
  int width, height;

  void move(int dx, int dy) { x += dx; y += dy; }
  void draw() { /* ne fait rien */ }
}
```

Deux méthodes sont déclarées dans la classe `Graphical` : `move` translate l'élément et `draw` le dessine. Comme on ne connaît pas encore la nature de l'élément, la méthode `draw` ne fait rien pour l'instant.

Pour représenter un élément particulier, par exemple un rectangle, on hérite de la classe `Graphical`.

```
class Rectangle extends Graphical {
```

En particulier, `Rectangle` hérite des champs `x`, `y`, `width` et `height` et des méthodes `move` et `draw`. On peut écrire un constructeur qui prend en arguments deux coins du rectangle :

```
  Rectangle(int x1, int y1, int x2, int y2) {
    x = (x1 + x2) / 2;
    y = (y1 + y2) / 2;
    width = Math.abs(x1 - x2);
    height = Math.abs(y1 - y2);
  }
```

On peut utiliser directement toute méthode héritée de `Graphical`, comme par exemple `move` :

```
  Rectangle p = new Rectangle(0, 0, 100, 50);
  p.move(10, 5);
```

Pour le dessin, en revanche, on va *redéfinir* la méthode `draw` dans la classe Rectangle (en anglais, la *redéfinition* se traduit par *overriding*).

```
class Rectangle extends Graphical {
  ...
  void draw() { ... dessine le rectangle ... }
}
```

Le rectangle sera alors effectivement dessiné quand on appelle

```
  p.draw();
```

On procède de même pour introduire une classe `Circle` pour des cercles.

```
class Circle extends Graphical {
  int radius;
  Circle(int cx, int cy, int r) {
    x = cx;
    y = cy;
    radius = r;
    width = height = 2 * radius;
  }
  void draw() { ... dessine le cercle ... }
}
```

Ici, on a ajouté un champ `radius` pour le rayon, afin de le conserver. Un objet de la classe `Circle` a donc cinq champs, dont quatre hérités de `Graphical`.

Type statique et type dynamique. La construction `new C(...)` construit un objet de classe `C`, et la classe de cet objet ne peut être modifiée par la suite. On l'appelle le *type dynamique* de l'objet. En revanche, le *type statique* d'une expression, tel qu'il est calculé par le compilateur pendant le typage, peut être différent du type dynamique, du fait de la relation de sous-typage introduite par l'héritage. Pendant la compilation du programme

```
  Graphical g = new Rectangle(0, 0, 100, 50);
  g.draw(); // dessine le rectangle
```

l'expression `g` a le type `Graphical`, mais le rectangle est effectivement dessiné. C'est bien la méthode `draw` de la classe `Rectangle` qui est exécutée. Bien sûr, la variable `g` est initialisée juste au dessus avec `new Rectangle` et le compilateur a donc connaissance du type dynamique de `g` s'il le souhaite. Mais ce n'est pas toujours possible, comme nous allons le montrer maintenant.

Introduisons un troisième type d'élément graphique, qui est simplement la réunion de plusieurs éléments graphiques. On commence par introduire une classe `GList` pour des listes chaînées de `Graphical`.

```
class GList {
  Graphical g;
  GList next;
  GList(Graphical g, GList next) {
    this.g = g;
    this.next = next;
  }
}
```

Le mot-clé `this` désigne l'objet dont on appelle le constructeur ou la méthode. Il est utilisé ici pour distinguer le paramètre formel `g` du champ `g` de même nom [2]. Un groupe d'éléments graphiques hérite de `Graphical` et contient la liste de ces éléments.

```
class Group extends Graphical {
  GList group;

  Group() { group = null; }

  void add(Graphical g) {
    group = new GList(g, group);
    // + mise à jour de x, y, width, height
  }
```

Il reste à redéfinir les méthodes `draw` et `move` dans la classe `Group` :

```
  void draw() {
    for (GList l = group; l != null; l = l.next)
      l.g.draw();
  }

  void move(int dx, int dy) {
    x += dx; y += dy;
    for (GList l = group; l != null; l = l.next)
      l.g.move(dx, dy);
  }
}
```

2. On s'épargne ainsi la peine d'introduire un nom différent pour le paramètre formel. À noter que l'affectation `g = g` serait acceptée mais ne ferait qu'affecter le paramètre `g` avec sa propre valeur.

Il est clair que pendant le typage de ces deux méthodes, le compilateur ne peut pas connaître le type dynamique de `l.g`. La liste `l` peut en effet contenir des `Rectangle` comme des `Circle`, arbitrairement mélangés. On pourrait même avoir dans `l` des objets de sous-classes de `Graphical` qui n'ont pas encore été définies.

Classe abstraite. Comme il n'y a jamais lieu de créer d'instance de la classe `Graphical`, on peut en faire une *classe abstraite*. On est alors dispensé de donner le code de certaines méthodes, comme par exemple `draw`.

```
abstract class Graphical {
  int x, y;
  int width, height;

  void move(int dx, int dy) { x += dx; y += dy; }
  abstract void draw();
}
```

Il est alors obligatoire de définir `draw` dans toute sous-classe (non abstraite) de `Graphical`.

Surcharge. En Java, plusieurs méthodes d'une même classe peuvent porter le même nom, pourvu qu'elles aient des arguments en nombre et/ou en nature différents. C'est ce que l'on appelle la *surcharge* (en anglais *overloading*). On peut définir ainsi deux méthodes `draw` dans la classe `Rectangle`

```
class Rectangle extends Graphical {
  ...
  void draw() {
    ...
  }
  void draw(String color) {
    ...
  }
}
```

puis écrire ensuite

```
  r.draw() ... r.draw("red") ...
```

La surcharge est résolue au typage. Tout se passe comme si on avait écrit deux méthodes avec des noms différents, par exemple

```
class Rectangle extends Graphical {
  ...
  void draw() {
    ...
  }
  void draw_String(String color) {
    ...
  }
}
```

puis

```
  r.draw() ... r.draw_String("red") ...
```

On peut surcharger également les constructeurs. Ainsi, on peut ajouter à la classe `Rectangle` un second constructeur ne prenant que trois arguments pour construire un carré.

```
class Rectangle extends Graphical {
  Rectangle(int x1, int y1, int x2, int y2) {
    ...
  }
  Rectangle(int x1, int y1, int w) {
    this(x1, y1, x1 + w, y1 + w); /* construit un carré */
  }
  ...
```

Sur la première ligne d'un constructeur, la syntaxe `this(...)` permet d'appeler un autre constructeur de la même classe. On peut utiliser à la place la syntaxe `super(...)` pour appeler plutôt un constructeur de la super classe. En l'absence de l'une de ces deux constructions, un appel `super()` au constructeur sans argument de la super classe est inséré implicitement.

Brève comparaison des paradigmes fonctionnel et objet. Arrivés à ce point-là, nous pouvons légitimement nous demander ce qui différencie les quatre classes `Graphical`, `Rectangle`, `Circle` et `Group` que nous venons de définir d'un programme OCaml tel que

```
  type graphical = Circle of ... | Rectangle of ... | Group of ...
  let move = function Circle _ -> ... | Rectangle _ -> ...
  let draw = function Circle _ -> ... | Rectangle _ -> ...
```

Une première différence, quasi syntaxique, a trait à l'organisation du code. En Java, le code relatif à chaque type d'objet graphique est isolé dans une classe différente, possiblement dans un fichier différent. En conséquence, le code relatif à une opération (dessiner, déplacer, etc.) est éclaté dans les différentes classes. En OCaml, ce sera l'inverse : le code relatif à une opération est isolé dans une unique fonction mais les différents constructeurs du type `graphical` sont réunis dans une même déclaration. Si on s'arrête là, la différence n'est pas si importante que cela. En particulier, l'appel dynamique de méthode de Java et le filtrage d'OCaml jouent des rôles très semblables et sont compilés avec une efficacité comparable.

En revanche, la différence entre Java et OCaml se fera ressentir de façon plus significative si l'on cherche à modifier notre code pour y ajouter une nouvelle sorte d'objet graphique (par exemple des triangles) ou une nouvelle opération (par exemple de changement d'échelle). Dans le cas de Java, il est facile d'ajouter un nouvelle sorte d'objet. On peut le faire dans un nouveau fichier. On peut même faire si le code original est uniquement disponible sous forme compilée. Avec OCaml, c'est moins facile, car il faut modifier le type `graphical` et toutes les fonctions déjà écrites. En particulier, on ne peut pas le faire sans accès au code source. Si on souhaite en revanche ajouter une nouvelle opération, alors c'est le contraire : le faire avec OCaml est aisé (on déclare une nouvelle fonction et on n'a pas besoin du source) mais le faire avec Java est difficile (on modifie les classes existantes et on a donc besoin du source). La figure 8.1 résume cette dualité.

	extension horizontale = ajout d'un cas	extension verticale = ajout d'une fonction
Java	*facile* (un seul fichier)	pénible (plusieurs fichiers)
OCaml	pénible (plusieurs fichiers)	*facile* (un seul fichier)

FIGURE 8.1 – Dualité des paradigmes fonctionnel et objet.

8.2 Compilation de Java

Expliquons maintenant comment sont compilés les concepts objets que nous venons d'introduire. Un objet est un bloc alloué sur le tas contenant d'une part sa classe et d'autre par les valeurs de ses différents champs. Si on reprend l'exemple de nos éléments graphiques, et que l'on construit ces deux objets,

```
new Rectangle(0, 0, 100, 50)
new Circle(50, 50, 10)
```

alors on aura deux blocs alloués en mémoire, respectivement de la forme suivante

	`Rectangle`
`x`	0
`y`	0
`width`	100
`height`	50

	`Circle`
`x`	50
`y`	50
`width`	20
`height`	20
`radius`	10

On ne détaille pas ici sous quelle forme la classe est stockée dans l'objet, mais il est important de comprendre que cette information est présente et immuable. La valeur d'un objet est l'adresse du bloc qui le représente. (Nous l'avions déjà expliqué dans la section 6.2.)

On note que l'héritage simple permet de stocker la valeur d'un champ à un emplacement constant dans le bloc, les champs propres venant après les champs hérités. Ainsi, la valeur de `width` sera toujours stockée dans le troisième champ de l'objet, qu'il s'agisse d'un `Rectangle`, d'un `Circle` ou bien de toute autre sous-classe de `Graphical`. Cette organisation, dite en *préfixe*, permet de compiler l'accès à un champ avec la seule information du type statique, sans connaître le type dynamique. Pour chaque champ, le compilateur connaît la position où ce champ est rangé, c'est-à-dire le décalage à ajouter au pointeur sur l'objet. Si par exemple le champ `width` est rangé à la position `+32` alors l'expression `e.width` est compilée comme

```
  ...                    # on compile e dans %rcx
  movq 32(%rcx), %rax   # champ width
```

Appel de méthode. Toute la subtilité de la compilation des langages à objets est dans l'*appel d'une méthode dynamique*. Pour cela, on construit pour chaque classe un *descripteur de classe* qui contient les adresses des codes de méthodes dynamiques de cette classe. Comme pour les champs, l'héritage simple permet de ranger l'adresse du code

de la méthode m à un emplacement constant dans le descripteur. Les descripteurs de classes peuvent être construits dans le segment de données. Chaque objet contient dans son premier champ un pointeur vers le descripteur de sa classe. Pour les quatre classes suivantes

```
class A           { void f() {...} }
class B extends A { void g() {...} }
class C extends B { void g() {...} }
class D extends C { void f() {...} void h() {...} }
```

on construira quatre descripteurs de classe de la forme suivante [3]

descr. A	descr. B	descr. C	descr. D
A_f	A_f	A_f	D_f
	B_g	C_g	C_g
			D_h

où `A_f`, `B_g`, etc., sont les adresses des codes des différentes méthodes. Comme pour le rangement des champs dans les objets, on a adopté ici une organisation en préfixe pour le rangement de ces adresses dans les descripteurs. Par conséquent, pour compiler un appel de méthode $e.m(e_1, \dots, e_n)$, il suffit de

1. calculer la valeur de e, qui est une adresse a vers un objet ;
2. accéder au descripteur de la classe de cet objet ;
3. trouver l'adresse du code de la méthode m dans ce descripteur, avec un décalage qui ne dépend que de m ;
4. appeler cette fonction de manière traditionnelle, en lui passant la valeur de l'objet en plus des valeurs des arguments $e_1, \dots, e_n$.

Résolution de la surcharge. La surcharge est résolue au typage, c'est-à-dire en utilisant uniquement les types statiques. Une fois cette résolution effectuée, on peut considérer que les constructeurs et méthodes portent des noms distincts. Ainsi, le programme de gauche deviendra le programme de droite et on pourra oublier toute considération de surcharge pendant la compilation proprement dite.

```
class A {
  A() {...}
  A(int x) {...}

  void m() {...}
  void m(A a) {...}
  void m(A a, A b) {...}
```

```
class A {
  A() {...}
  A_int(int x) {...}

  void m() {...}
  void m_A(A a) {...}
  void m_A_A(A a, A b) {...}
```

La surcharge n'en est pas pour le moins délicate. Si on introduit par exemple

3. En pratique, le descripteur de la classe C contient également l'indication de la super classe de C, comme un pointeur vers son descripteur.

```
class A {...}
class B extends A {
  void m(A a) {...}
  void m(B b) {...}
}
```

alors dans l'extrait de programme

```
{ ... B b = new B(); b.m(b); ... }
```

les deux méthodes s'appliquent potentiellement. C'est la méthode `m(B b)` qui est appelée, car *plus précise* du point de vue de l'argument. Dans certains cas, il peut y avoir ambiguïté. En voici un exemple :

```
class A {...}
class B extends A {
  void m(A a, B b) {...}
  void m(B b, A a) {...}
}
{ ... B b = new B(); b.m(b, b); ... }
```

Sur un tel programme, le compilateur Java va échouer au typage, avec le message

```
surcharge1.java:13: reference to m is ambiguous,
  both method m(A,B) in B and method m(B,A) in B match
```

L'algorithme de résolution de la surcharge fonctionne de la manière suivante. À chaque méthode définie

$$\tau\ \mathtt{m}(\tau_1\ x_1, ..., \tau_n\ x_n)$$

dans la classe `C` on associe le profil

$$(\mathtt{C}, \tau_1, \ldots, \tau_n).$$

On *ordonne* les profils en posant $(\tau_0, \tau_1, \ldots, \tau_n) \sqsubseteq (\tau'_0, \tau'_1, \ldots, \tau'_n)$ si et seulement si τ_i est un sous-type de τ'_i pour tout i. Pour un appel de méthode

$$e.m(e_1, \ldots, e_n)$$

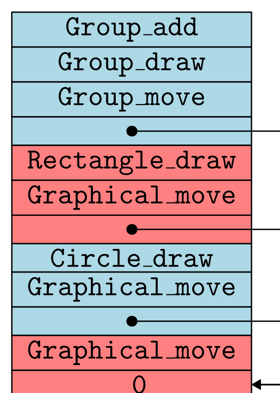
où e a le type statique τ_0 et e_i le type statique τ_i, on considère l'ensemble des éléments *minimaux* dans l'ensemble des profils compatibles. Si cet ensemble est vide, aucune méthode ne s'applique. Si cet ensemble contient plusieurs éléments, il y a ambiguïté. Enfin, si cet ensemble contient un unique élément, c'est la méthode à appeler.

Un exemple complet. La figure 8.2 contient l'intégralité de notre programme Java, que nous allons maintenant compiler manuellement vers l'assembleur x86-64 pour illustrer ce que nous venons d'expliquer. On commence par la construction des trois descripteurs de classes dans le segment de données.

```java
abstract class Graphical {
  int x, y;
  int width, height;
  void move(int dx, int dy) { x += dx; y += dy; }
  abstract void draw();
}
class Circle extends Graphical {
  int radius;
  Circle(int cx, int cy, int r) {
    x = cx; y = cy; radius = r; width = height = 2 * radius; }
  void draw() { ... /* dessin */ ... }
}
class Rectangle extends Graphical {
  Rectangle(int x1, int y1, int x2, int y2) {
    this.x = (x1+x2)/2; this.y = (y1+y2)/2;
    width = Math.abs(x1-x2); height = Math.abs(y1-y2); }
  void draw() { ... /* dessin */ ... }
}
class GList {
  Graphical g;
  GList next;
  GList(Graphical g, GList next) { this.g = g; this.next = next; }
}
class Group extends Graphical {
  private GList group;
  Group() { group = null; }
  void add(Graphical g) {
    group = new GList(g, group);
    ... // mise à jour x,y,width,height
  }
  void draw() {
    for (GList l = group; l != null; l = l.next) l.g.draw(); }
  void move(int dx, int dy) {
    super.move(dx, dy); // pour mettre à jour x,y
    for (GList l = group; l != null; l = l.next) l.g.move(dx, dy); }
}
class Main {
  public static void main(String arg[]) {
    Rectangle g1 = new Rectangle(0, 0, 100, 50);
    g1.move(10, 5); g1.draw();
    Graphical g2 = new Circle(10,10,2);
    Group g3 = new Group(); g3.add(g1); g3.add(g2);
    g3.draw(); g3.move(-5,-7); g3.draw();
  }
}
```

FIGURE 8.2 – Un exemple de programme Java.

```
        .data
descr_Graphical:
        .quad 0
        .quad Graphical_move
descr_Circle:
        .quad descr_Graphical
        .quad Graphical_move
        .quad Circle_draw
descr_Rectangle:
        .quad descr_Graphical
        .quad Graphical_move
        .quad Rectangle_draw
descr_Group:
        .quad descr_Graphical
        .quad Group_move
        .quad Group_draw
        .quad Group_add
```



(les adresses croissent vers le haut)

Ici, `Graphical_move`, `Circle_draw`, etc., correspondent aux adresses des méthodes.

Le code d'un constructeur est une fonction qui suppose l'objet déjà alloué et son adresse dans `%rdi`, le premier champ déjà rempli (avec le descripteur de classe) et les arguments du constructeur dans `%rsi`, `%rdx`, etc., et la pile si besoin. On compile donc ainsi le constructeur de la classe `GList` :

```
class GList {
  Graphical g;
  GList next;
  GList(Graphical g, GList next) {
    this.g = g; this.next = next;
  }
}
```

```
new_GList:
        movq %rsi, 8(%rdi)
        movq %rdx, 16(%rdi)
        ret
```

Les constructeurs des classes `Circle`, `Rectangle` et `Group` sont compilés de façon similaire. Pour les méthodes, on adopte la même convention : l'objet est dans `%rdi` et les arguments de la méthode dans `%rsi`, `%rdx`, etc., et la pile si besoin.

```
abstract class Graphical {
  void move(int dx, int dy) {
    x += dx; y += dy;
  }
}
```

```
Graphical_move:
        addq %rsi, 8(%rdi)
        addq %rdx, 16(%rdi)
        ret
```

La méthode `draw` de la classe `Group` est plus intéressante, car elle contient un appel dynamique.

```
class Group extends Graphical {
  void draw() {
    GList l = group;
    while (l != null) {
      l.g.draw();
      l = l.next;
    }
  }
}
```

```
Group_draw:
   pushq %rbx
   movq  8(%rdi), %rbx
   jmp   2f
1: movq  8(%rbx), %rdi
   movq  (%rdi), %rcx
   call  *16(%rcx)
   movq  16(%rbx), %rbx
2: testq %rbx, %rbx
   jnz   1b
   popq  %rbx
   ret
```

L'appel dynamique est compilé par un saut à une adresse calculée avec `call *`. Nous avions déjà utilisé cette instruction pour compiler l'appel à une fonction de première classe dans le chapitre précédent (voir page 119).

Expliquons enfin comment construire un objet, c'est-à-dire comment compiler la construction `new`, en prenant la première ligne du programme principal.

```
public static void main(String arg[]){
  Rectangle g1 =
    new Rectangle(0, 0, 100, 50);
  ...
```

```
main:
  movq $40, %rdi
  call malloc
  movq %rax, %r12
  movq $descr_Rectangle,(%r12)
  movq %r12, %rdi
  movq $0, %rsi
  movq $0, %rdx
  movq $100, %rcx
  movq $50, %r8
  call new_Rectangle
  ...
```

Ici, on a supposé la variable `g1` alloué dans le registre `%r12`. On commence par allouer un bloc de 40 octets sur le tas avec `malloc`, qui se décomposent en 8 octets pour le pointeur vers le descripteur de classe et 8 octets pour chacun des champs de la classe `Rectangle`[4]. Une fois le résultat de `malloc` obtenu, et copié dans `%r12`, on stocke le descripteur de classe, c'est-à-dire l'adresse représentée par `descr_Rectangle`, dans le premier champ de l'objet. Enfin, on appelle le constructeur `new_Rectangle` en passant l'objet dans `%rdi` et les arguments dans `%rsi`, `%rdx`, `%rcx` et `%r8`.

Exercice 39. Écrire le reste du code assembleur de la fonction `main`. Solution □

4. Le type `int` de Java est spécifié comme représentant des entiers 32 bits signés. On pourrait donc se contenter de 4 octets pour chaque champ. On choisit cependant ici la facilité d'une représentation uniforme où chaque champ occupe exactement 8 octets, qu'il s'agisse d'un pointeur ou d'un entier.

Optimisation de l'appel. Pour plus d'efficacité, on peut chercher à remplacer un appel dynamique (*i.e.*, calculé pendant l'exécution) par un appel statique (*i.e.*, connu à la compilation). Pour un appel $e.m(\dots)$, et e de type statique C, c'est notamment possible lorsque la méthode m n'est redéfinie dans aucune sous-classe de C. Une autre possibilité, plus complexe, consiste à propager les types connus à la compilation (en anglais *type propagation*). Dans un code tel que

```
B b = new B();
A a = b;
a.m();
```

on a connaissance que le type dynamique de `a` est `B` et on en déduit donc à quelle méthode `a.m` fait référence. On peut alors compiler cet appel comme un appel traditionnel.

Transtypage. Comme on l'a vu, le type statique et le type dynamique d'une expression désignant un objet peuvent différer, à cause du sous-typage. Il est parfois nécessaire de « forcer la main » au compilateur, en prétendant qu'un objet e appartient à une certaine classe C, ou plus exactement à l'une des super classes de C. On appelle cela le *transtypage* (en anglais *cast*). La construction de Java pour le transtypage est

$$(C)e$$

Le type statique d'une telle expression est C. Si D le type statique de l'expression e et E le type dynamique de (l'objet désigné par) e, il y a alors trois situations :

- si C est une super classe de D, on parle de transtypage *vers le haut* (en anglais *upcast*) et le code produit pour $(C)e$ est le même que pour e. Cependant, le *cast* a une influence sur le typage puisque le type de $(C)e$ est C.
- si C est une sous-classe de D, on parle de transtypage *vers le bas* (en anglais *downcast*) et le code contient un *test dynamique* pour vérifier que E est bien une sous-classe de C.
- enfin, si C n'est ni une sous-classe ni une super classe de D, le compilateur refuse l'expression.

Le transtypage vers le haut est parfois nécessaire pour faire référence à un champ qui a été masqué par un autre. Considérons par exemple les deux classes suivantes :

```
class A           { int x = 1; }
class B extends A { int x = 2; }
```

Un objet de la classe `B` possède deux champs `x`, un hérité de `A` et un qui lui est propre. Si `b` est un objet de la classe `B`, alors `b.x` fait référence au second. Pour accéder au premier, on peut écrire `((A)b).x`. En effet, l'expression `(A)b` a le type statique `A`, ce qui affecte la compilation de l'accès au champ `x` (le décalage est différent).

Donnons maintenant un exemple de transtypage vers le bas, avec une méthode `m` qui cherche à appeler la méthode `add` d'un `Graphical` :

```
  void m(Graphical g, Graphical x) {
    ((Group)g).add(x);
  }
```

Rien ne garantit que l'objet `g` passé à `m` sera bien un `Group`. En particulier, il pourrait même ne pas posséder de méthode `add`. Le test dynamique est donc nécessaire. L'exception

`ClassCastException` est levée si le test échoue. Pour permettre une programmation un peu plus défensive, il existe une construction booléenne

$$e \texttt{ instanceof } C$$

qui détermine si la classe de e est bien une sous-classe de C. Du coup, on trouve souvent le schéma

```
if (e instanceof C) {
  C c = (C)e;
  ...
}
```

Dans ce cas, le compilateur effectue typiquement une optimisation consistant à ne pas générer de second test pour le transtypage. La compilation de la construction `e instanceof C` ne pose pas de difficulté. Il s'agit d'une simple boucle qui remonte la hiérarchie de classes, en partant de la classe dynamique de `e` jusqu'à atteindre `C` ou `Object`, la classe tout en haut de la hiérarchie.

Le compilateur peut optimiser les constructions $(C)e$ et e `instanceof` C dans certains cas. Si C est l'unique sous-classe de D alors on peut faire un unique test d'égalité plutôt qu'une boucle. Si D est une sous-classe de C alors e `instanceof` C vaut `true` directement.

Exercice 40. Compiler la construction `instanceof` en assembleur. Solution □

8.3 Quelques mots sur C++

On reprend l'exemple de la figure 8.2. On en donne une version en C++ dans la figure 8.3. Au delà de la seule syntaxe, il y a quelques différences notables avec Java. La visibilité des champs et méthodes, par exemple, n'est pas la même par défaut. Pour que les champs et les méthodes de `Graphical` soient visibles, il faut les faire précéder de `public`. De même, `Circle` doit hériter publiquement de `Graphical` pour que les champs et méthodes restent visibles.

D'autre part, il faut explicitement déclarer que la méthode `move` pourra être redéfinie, en ajoutant le qualificatif `virtual` devant sa déclaration. Là encore, le comportement est à l'opposé de Java, où l'on déclare au contraire qu'une méthode ne peut pas être redéfinie (avec `final`).

On note aussi que la méthode `add` de `Group` reçoit son argument `g` sous la forme d'un pointeur. En effet, le mode de passage par défaut de C++ est par valeur, comme nous l'avons expliqué dans la section 6.2, et un objet ne fait pas exception. Dans notre cas, on ne souhaite pas que l'objet soit copié. De manière plus générale, nous avons ici fait apparaître explicitement un certain nombre de pointeurs qui étaient implicites dans le code Java.

Une autre différence notable avec Java est la possibilité d'allouer des objets sur la pile. On le fait par exemple dans la méthode `main` quand on écrit

```
Rectangle g1(0, 0, 100, 50);
```

Le constructeur de `Rectangle` est appelé ici pour un objet alloué sur la pile. On peut également allouer un objet sur le tas, avec une construction `new` analogue à celle de Java. Ainsi, on aurait pu écrire

```
class Graphical {
public:
  int x, y, width, height;
  virtual void move(int dx, int dy) { x += dx; y += dy; }
  virtual void draw() = 0;
};
class Circle : public Graphical {
public:
  int radius;
  Circle(int cx, int cy, int r) {
    x = cx; y = cy; radius = r; width = height = 2 * radius; }
  void draw() { ... /* dessin */ ... }
};
class Rectangle : public Graphical {
public:
  Rectangle(int x1, int y1, int x2, int y2) {
    this->x = (x1+x2)/2; this->y = (y1+y2)/2;
    width = abs(x1-x2); height = abs(y1-y2); }
  void draw() { ... /* dessin */ ... }
};
class GList {
public:
  Graphical* g;
  GList* next;
  GList(Graphical* g, GList* next) { this->g = g; this->next = next; }
};
class Group : public Graphical {
  GList* group;
public:
  Group() { group = NULL; }
  void add(Graphical* g) {
    group = new GList(g, group); ... /* mise à jour x,y,width,height */ }
  void draw() {
    for(GList* l = group; l != NULL; l = l->next) l->g->draw(); }
  void move(int dx, int dy) {
    Graphical::move(dx, dy); // pour mettre à jour x,y
    for(GList* l = group; l != NULL; l = l->next) l->g->move(dx, dy); }
};
int main() {
  Rectangle g1 = Rectangle(0, 0, 100, 50);
  g1.move(10, 5); g1.draw();
  Circle g2 = Circle(10, 10, 2);
  Group g3;
  g3.add(&g1); g3.add(&g2);
  g3.draw(); g3.move(-5,-7); g3.draw();
}
```

FIGURE 8.3 – Le programme de la figure 8.2 en C++.

```
Rectangle* g1 = new Rectangle(0, 0, 100, 50);
```

On note que le résultat de `new` est un pointeur sur un objet. À la différence de Java, ce pointeur est explicite. Qu'un objet soit alloué sur la pile ou sur le tas, sa représentation en mémoire en identique.

Notons enfin une dernière différence importante entre Java et C++. Pour obtenir une seconde variable `v` désignant le même objet que `g1`, on peut écrire en C++

```
Rectangle* v = &g1;
```

Ainsi, `v` est un pointeur vers l'objet représenté par `g1`. Si on avait écrit en revanche

```
Rectangle v = g1;
```

comme on le ferait en Java, on aurait *copié* l'objet `g1` dans un nouvel objet `v`, avec un effet totalement différent. À cet égard, la sémantique de C++ est cohérente avec l'affectation des structures.

Représentation des objets. Sur cet exemple, la représentation d'un objet n'est pas différente de Java.

descr. `Circle`
`x`
`y`
`width`
`height`
`radius`

descr. `Rectangle`
`x`
`y`
`width`
`height`

descr. `Group`
`x`
`y`
`width`
`height`
`group`

Mais en C++, on trouve aussi de l'*héritage multiple*. Par conséquent, on ne peut plus (toujours) utiliser le principe selon lequel la représentation d'un objet d'une super classe de C est un préfixe de la représentation d'un objet de la classe C (et de même pour les descripteurs de classes). Voici un exemple d'héritage multiple, où la classe `FlexibleArray` hérite à la fois de la classe `Array` et de la classe `List`.

```
class Collection {
  int cardinal;
};
class Array : public Collection {
  int nth(int i) ...
};
class List {
  void push(int v) ...
  int pop() ...
};
class FlexibleArray : public Array, public List {
};
```

descr. `FlexibleArray`
`cardinal`
...
descr. `List`
...
...

À droite, on a illustré la représentation d'un objet de la classe `FlexibleArray`. Les représentations d'objets des deux classes `Array` et `List` y sont juxtaposées, suivies des champs

propres à la classe `FlexibleArray`[5]. En particulier, un transtypage comme

```
FlexibleArray fa;
List l = (List)fa;
```

n'est pas traduit par l'identité mais par une arithmétique de pointeur, en l'occurrence une affectation de la forme

```
l = fa + 24;
```

Supposons maintenant que `List` hérite également de `Collection`. On aura maintenant *deux* champs `cardinal` dans un objet de la classe `FlexibleArray`, l'un hérité de `Array` et l'autre de `List`.

```
...
class List : public Collection {
  void push(int v) ...
  int pop() ...
};
class FlexibleArray : public Array, public List {
};
```

descr. `FlexibleArray`
`cardinal`
...
descr. `List`
`cardinal`
...
...

En particulier, il y maintenant une ambiguïté si on fait référence au champ `cardinal` d'un objet de la classe `FlexibleArray`.

```
int main() {
  FlexibleArray fa;
  cout << fa.cardinal << endl;
};
```

```
test.cc: In function 'int main()':
test.cc:42:14: error: request for member 'cardinal' is ambiguous
```

Il faut préciser de quel champ `cardinal` il s'agit, avec cette notation :

```
cout << fa.Array::cardinal << endl;
```

Pour n'avoir qu'une seule instance de `Collection` à l'intérieur de `FlexibleArray`, il faut modifier la façon dont `Array` et `List` héritent de `Collection`, en utilisant l'*héritage virtuel*.

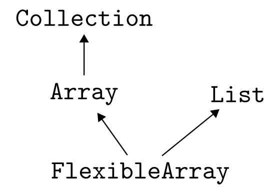
```
class Collection                                { ... };
class Array : public virtual Collection         { ... };
class List : public virtual Collection          { ... };
class FlexibleArray : public Array, public List { ... };
```

Il n'y a alors plus qu'une seule instance de la classe `Collection` dans la classe `FlexibleArray` et donc en particulier un seul champ `cardinal`.

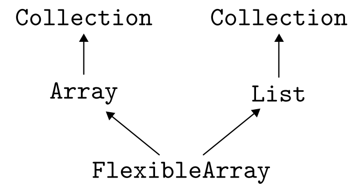
On peut résumer les trois situations ci-dessus en illustrant à chaque fois le diagramme de classes correspondant.

5. L'option `-fdump-class-hierarchy` du compilateur GNU C++ donne un accès complet à la représentation des objets et des descripteurs de classe.

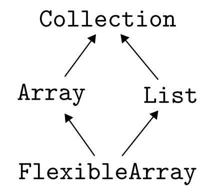
```
class Collection
class Array : Collection
class List
class FlexibleArray : Array, List
```




```
class Collection
class Array : Collection
class List : Collection
class FlexibleArray : Array, List
```




```
class Collection
class Array : virtual Collection
class List : virtual Collection
class FlexibleArray : Array, List
```



Cette dernière situation est appelé le *diamant* ou encore le *losange*. Elle apparaît parfois comme un problème, lorsqu'elle est accidentelle, qui jouit d'une réputation sulfureuse. Les anglo-saxons en parlent sous le terme de *deadly diamond of death*, ce qui veut tout dire.

9
Compilateur optimisant

Dans ce chapitre, nous allons écrire un compilateur pour un fragment simple du langage C vers l'assembleur x86-64, en cherchant à produire du code raisonnablement efficace. En particulier, on va chercher à bien utiliser les seize registres et les nombreuses instructions de l'architecture x86-64.

Le fragment du langage C que nous allons compiler contient des entiers (type `int` uniquement), des pointeurs vers des structures (allouées sur le tas uniquement), des fonctions et les structures de contrôle `if`, `while` et `return`. La syntaxe abstraite de ce fragment est donnée figure 9.1. Ce fragment peut paraître modeste, mais en utilisant quelques fonctions de la bibliothèque C on peut allouer des structures sur le tas (avec `sbrk` ou `malloc`) ou encore faire des entrées-sorties (par exemple avec `putchar`). Pour simplifier un peu notre compilateur, on fait ici le choix d'entiers 64 bits signés pour le type `int`, ce que le standard C nous autorise à faire, même si ce n'est pas l'usage [1]. De cette manière, entiers et pointeurs occuperont tous 64 bits.

Il est illusoire de chercher à produire du code efficace en une seule passe. La production de code va donc être décomposée en plusieurs phases. Le nombre et la nature de ces phases dépend des compilateurs. Nous choisissons ici l'architecture du compilateur CompCert [2] de Xavier Leroy. Cette architecture n'est pas liée au langage C. On pourrait tout autant l'utiliser pour compiler un langage fonctionnel ou orienté objets.

Notre point de départ est l'arbre de syntaxe abstraite issu du typage. En particulier, on suppose avoir distingué tous les identificateurs, avoir distingué également les variables locales des variables globales, en enfin que le type de chaque sous-expression est connu.

9.1 Sélection d'instructions

La première phase de notre compilateur s'appelle la *sélection d'instructions*. Elle consiste à remplacer les opérations arithmétiques du C par celles de x86-64 et à remplacer les accès aux champs de structures par des opérations explicites d'accès à la mémoire.

On peut le faire naïvement, en traduisant chaque opération arithmétique de C par

1. Sur une architecture 64 bits, un compilateur C choisit typiquement de représenter le type `int` sur 32 bits, réservant le type `long int` aux entiers 64 bits. Pour rendre notre compilateur plus standard, il suffirait donc de remplacer `int` par `long int`.

2. voir http://compcert.inria.fr/

```
E  ::= n                              expression
     | L
     | L = E
     | E op E | - E | ! E
     | x(E,...,E)
     | sizeof(struct x)

L  ::= x                              valeur gauche
     | E->x

op ::= == | != | < | <= | > | >=      opérateurs binaires
     | && | || | + | - | * | /

S  ::= ;                              instruction
     | E;
     | if (E) S else S
     | while (E) S
     | return E;
     | B

B  ::= { V...V S...S }                bloc

V  ::= int x,...,x;                   variables ou champs
     | struct x *x,...,*x;

T  ::= int | struct x *               type

D  ::= V                              déclaration de variables globales
     | T x(T x,...,T x) B             déclaration de fonction
     | struct x {V...V};              déclaration de structure

P  ::= D...D                          programme
```

FIGURE 9.1 – Syntaxe abstraite de Mini-C.

l'instruction correspondante de x86-64. Cependant, x86-64 fournit des instructions permettant une plus grande efficacité, comme par exemple l'addition d'un registre et d'une constante ou encore le décalage des bits d'en entier vers la gauche ou la droite, qu'on peut interpréter comme une multiplication ou à une division par une puissance de deux.

D'autre part, il est possible, et souhaitable, d'évaluer autant d'expressions que possible pendant la compilation. On appelle cela l'*évaluation partielle*. Ainsi, on peut simplifier l'expression $(1 + e_1) + (2 + e_2)$ en $e_1 + e_2 + 3$, en s'épargnant ainsi une addition. De la même façon, on peut simplifier l'expression $!(e_1 < e_2)$ et $e_1 \geq e_2$, en remplaçant ainsi deux opérations par une seule. Bien entendu, de telles simplifications soit préserver la sémantique de programmes. Si par exemple un ordre d'évaluation gauche/droite était spécifié, on ne pourrait simplifier $(0 - e_1) + e_2$ en $e_2 - e_1$ que si les expressions e_1 et e_2 n'interfèrent pas. C'est le cas par exemple si e_1 et e_2 sont pures *i.e.* sans aucun effet. En C, l'ordre d'évaluation n'est pas spécifié et on peut donc toujours remplacer $(0-e_1)+e_2$ par e_2-e_1 si on le souhaite, même si les expressions e_1 et e_2 interfèrent. C'est au programmeur de ne pas écrire des expressions telles que `(0-x++) + ++x`.

Un autre exemple est la simplification de $e + 1 < 10$ en $e < 9$. Cela peut paraître astucieux, mais la sémantique n'est pas toujours préservée. En effet, si e est le plus grand entier, $e + 1$ va provoquer un débordement arithmétique et se retrouver inférieur à 9. En arithmétique *non signée* en C, le comportement du débordement arithmétique est spécifié, comme un calcul modulo 2^n avec n la taille des entiers, et une telle simplification ne pourrait donc pas être effectuée. En arithmétique *signée*, en revanche, le débordement est un comportement non spécifié en C. On peut donc faire cette simplification pour le type `int`. Avec `gcc -O2`, elle est effectivement faite et lorsque e vaut `INT_MAX` on obtient un résultat différent de celui obtenu avec `gcc -O1` qui n'optimise pas $e+1 < 10$ en $e < 9$.

Un dernier exemple est la simplification de $0\times e$ par 0. Elle n'est possible que si l'expression e est sans effet. Comme nos expressions incluent des appels de fonction, déterminer si e est sans effet n'est pas décidable. Mais on peut se contenter d'une approximation correcte, comme par exemple

$$\begin{array}{rcl} pure(n) & = & true \\ pure(x) & = & true \\ pure(e_1 \texttt{ + } e_2) & = & pure(e_1) \wedge pure(e_2) \\ & \vdots & \\ pure(e_1 \texttt{ = } e_2) & = & false \\ pure(f(e_1,\dots,e_n)) & = & false \quad \text{(on ne sait pas)} \end{array}$$

Lorsque e n'est pas pure, il reste possible de simplifier $0 \times e$ en 0 après avoir évalué e pour ses effets. On s'épargne ainsi une multiplication inutile. Dans le cas d'une division $0/e$ il faut être encore plus soigneux : évaluer e pour ses effets, tester si $e = 0$ et signaler une division par zéro le cas échéant et sinon donner la valeur 0 à l'expression sans faire de division. On peut se demander pourquoi le programmeur aurait écrit en premier lieu des expressions telles que $0 \times e$ ou $0/e$, mais il faut garder à l'esprit l'utilisation de macros (même si elle est déconseillée) ou encore la compilation de code C produit automatiquement.

La sélection d'instructions va transformer les arbres de syntaxe abstraite en de nouveaux arbres dans une syntaxe abstraite légèrement différente, où les opérations sont maintenant celles de x86-64. Cette nouvelle syntaxe abstraite est donnée figure 9.2. La

principale différence se situe dans les expressions. Le reste consiste uniquement à oublier les types et à regrouper les variables locales en début de fonction et les variables globales en début de programme.

Pour réaliser la sélection d'instructions tout en incorporant de l'évaluation partielle, on peut utiliser des constructeurs intelligents (en anglais *smart constructors*). Il s'agit de fonctions se comportant comme des constructeurs de la syntaxe abstraite, tout en effectuant des simplifications à la volée. Par exemple, on peut se donner le constructeur suivant pour l'addition :

$$
\begin{array}{rcl}
\texttt{mkAdd}(n_1,\ n_2) & = & n_1 + n_2 \\
\texttt{mkAdd}(0,\ e) & = & e \\
\texttt{mkAdd}(e,\ 0) & = & e \\
\texttt{mkAdd}((\texttt{add}\ n_1)\ e,\ n_2) & = & \texttt{mkAdd}(n_1 + n_2,\ e) \\
\texttt{mkAdd}(n,\ e) & = & (\texttt{add}\ n)\ e \\
\texttt{mkAdd}(e,\ n) & = & (\texttt{add}\ n)\ e \\
\texttt{mkAdd}(e_1,\ e_2) & = & \texttt{add}\ e_1\ e_2 \quad \text{sinon}
\end{array}
$$

(Attention à ne pas confondre l'opérateur unaire `add` n, qui ajoute l'opérande immédiate n à son argument, à l'opérateur binaire d'addition `add`.) On pourrait faire encore plus de simplifications, par exemple en utilisant intelligemment l'instruction `lea`. Quoique l'on fasse, il faut garantir que la fonction de simplification termine. Ici, la taille des arguments de `mkAdd` diminue strictement lors de l'appel récursif, ce qui garantie la terminaison.

Une fois de tels constructeurs intelligents définis, la traduction peut se faire mot à mot. On note $IS(e)$ la traduction d'une expression e. Sa définition est donné figure 9.3. Les accès à la mémoire au travers de la construction `->` sont traduits en des accès indirects avec décalage. Ce décalage peut être calculé facilement, car chaque champ de structure occupe la même taille, à savoir 8 octets. Si x est le troisième champ d'une structure, par exemple, son décalage sera $d = 16$. De la même façon, on connaît la taille totale occupée par une structure et on peut donc traduire `sizeof(struct` x`)` directement par un entier. Pour le reste de la syntaxe abstraite, c'est-à-dire les appels de fonctions, les instructions et les déclarations, la sélection d'instructions est un morphisme. Voici un exemple de sélection d'instructions :

```
struct list {
  int val;
  struct list *next; };

int print(struct list *l) {
  struct list *p;
  p = l;
  while (p) {
    int c;
    c = p->val;
    putchar(c);
    p = p->next;
  }
  return 0;
}
```

```
// plus besoin du type list



print(l) {
  locals p, c;
  p = l;
  while (p) {

    c = load 0(p);
    putchar(c);
    p = load 8(p);
  }
  return 0;
}
```

$$
\begin{array}{rcll}
E & ::= & n & \text{expression} \\
& | & x & \\
& | & x \texttt{ = } E & \\
& | & \texttt{load } n(E) & \\
& | & \texttt{store } n(E)\ E & \\
& | & \mathit{binop}\ E\ E & \\
& | & \mathit{unop}\ E & \\
& | & x\texttt{(}E,\dots,E\texttt{)} & \\
\\
\mathit{binop} & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{imul} \mid \texttt{div} & \text{opérateurs binaires x86-64} \\
& | & \texttt{mov} \mid \texttt{setl} \mid \texttt{setle} \mid \dots & \\
\\
\mathit{unop} & ::= & \texttt{add } n \mid \texttt{sete } n \mid \dots & \text{opérateurs unaires x86-64} \\
\\
S & ::= & \texttt{;} & \text{instruction} \\
& | & E\texttt{;} & \\
& | & \texttt{if } (E)\ S \texttt{ else } S & \\
& | & \texttt{while } (E)\ S & \\
& | & \texttt{return } E\texttt{;} & \\
& | & \texttt{\{ } S \dots S \texttt{ \}} & \\
\\
F & ::= & x(x,\dots,x)\ \texttt{\{ } x\dots x\ S\dots S \texttt{ \}} & \text{déclaration de fonction} \\
\\
P & ::= & x\dots x\ F\dots F & \text{programme}
\end{array}
$$

FIGURE 9.2 – Syntaxe abstraite pour la sélection d'instructions.

$$
\begin{array}{rcll}
\mathit{IS}(e_1\texttt{+}e_2) & = & \texttt{mkAdd}(\mathit{IS}(e_1),\ \mathit{IS}(e_2)) & \\
\mathit{IS}(e_1\texttt{-}e_2) & = & \texttt{mkSub}(\mathit{IS}(e_1),\ \mathit{IS}(e_2)) & \\
& \vdots & & \\
\mathit{IS}(\texttt{!}e_1) & = & \texttt{mkNot}(\mathit{IS}(e_1)) & \\
\mathit{IS}(\texttt{-}e_1) & = & \texttt{mkNeg}(\mathit{IS}(e_1)) & \\
\mathit{IS}(e_1\texttt{->}x) & = & \texttt{load } d(\mathit{IS}(e_1)) & (d \text{ calculé en fonction de } x) \\
\mathit{IS}(e_1\texttt{->}x \texttt{ = } e_2) & = & \texttt{store } d(\mathit{IS}(e_1))\ \mathit{IS}(e_2) & (d \text{ calculé en fonction de } x) \\
\mathit{IS}(x\texttt{(}e_1,\dots,e_n\texttt{)}) & = & x\texttt{(}\mathit{IS}(e_1),\dots,\mathit{IS}(e_n)\texttt{)} & \\
\mathit{IS}(\texttt{sizeof(struct } x\texttt{)}) & = & n & (\text{taille de la structure } x)
\end{array}
$$

FIGURE 9.3 – Sélection d'instructions pour Mini-C.

Et voici un autre exemple avec la fonction factorielle :

```
int fact(int x) {

  if (x <= 1) return 1;
  return x * fact(x-1);
}
```

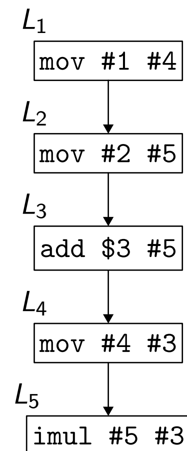
```
fact(x) {
  locals:
  if (setle x 1) return 1;
  return imul x fact((add -1) x);
}
```

Cet exemple de la fonction factorielle nous servira de fil conducteur tout au long des phases suivantes.

9.2 Production de code RTL

La deuxième phase consiste en une transformation vers un langage appelé RTL pour *Register Transfer Language*, avec plusieurs objectifs. D'une part, nous allons détruire la structure arborescente des expressions et des instructions au profit d'un *graphe de flot de contrôle* (en anglais *control flow graph* ou CFG), pour faciliter les phases ultérieures. En particulier, on va ainsi supprimer la distinction entre expressions et instructions. D'autre part, nous allons introduire une notion de *pseudo-registres*. À la différence des registres de la machine, ces pseudo-registres sont en nombre illimité. Ce n'est que plus tard, dans les phases suivantes, que ces pseudo-registres deviendront des registres x86-64, autant que possible, ou des emplacements de pile sinon. Enfin, les instructions RTL vont se rapprocher des instructions assembleur x86-64.

Illustrons l'idée du langage RTL sur un exemple. Considérons l'expression `b * (3 + d)` où `b` et `d` sont deux variables locales. On alloue des pseudo-registres pour ces deux variables, par exemple `#1` pour `b` et `#2` pour `d`. (On note `#`n le n-ième pseudo-registre.) On se donne un pseudo-registre pour recevoir le résultat de l'expression, par exemple `#3`. Le graphe de flot de contrôle RTL qui affecte à `#3` la valeur de l'expression est donné ci-contre. Il se décompose en cinq instructions RTL séquentielles. Chaque instruction est repérée par une étiquette, notée L_1, L_2, etc. La première instruction, à l'étiquette L_1, copie la valeur de `b` dans un nouveau pseudo-registre `#4`. Les deux instructions suivantes calculent la valeur de `3+d` dans un autre pseudo-registre `#5`. La quatrième instruction copie `#4` dans `#3` et enfin la dernière instruction multiplie `#3` par `#5`. Les instructions RTL prennent leur opérande de destination en dernier argument, dans la convention de l'assembleur AT&T. Bien sûr, il y avait d'autres façons de procéder ici. On aurait pu par exemple s'épargner l'usage du pseudo-registre `#4` et copier directement `#1` dans `#3` à l'avant-dernière instruction.



Le graphe de flot de contrôle peut être représenté par un dictionnaire associant une instruction RTL à chaque étiquette. Inversement, chaque instruction RTL indique quelle est l'étiquette suivante, ou les étiquettes suivantes pour une instruction de branchement. Ainsi, l'instruction RTL

$$\texttt{mov}\ n\ r\ \ \rightarrow L$$

signifie « charger la constante n dans le pseudo-registre r et transférer le contrôle à l'étiquette L ». L'ensemble des instructions RTL est donné figure 9.4.

Traduction d'une expression. On traduit une expression en se donnant une fonction $RTL(e, r_d, L_d)$ où e est l'expression à traduire, r_d un pseudo-registre devant recevoir la valeur de l'expression et L_d une étiquette L_d où transférer ensuite le calcul. La fonction RTL renvoie l'étiquette correspondant au point d'entrée dans le graphe pour le calcul de e. Ainsi, on traduit une expression « en partant de la fin ». Pour traduire une addition e_1+e_2, par exemple, on procède ainsi :

$$\begin{array}{rcll} RTL(e_1\texttt{+}e_2, r_d, L_d) & = & \text{ajouter } L_3 : \texttt{add}\ r_2\ r_d\ \to L_d & (r_2, L_3 \text{ frais}) \\ & & L_2 \leftarrow RTL(e_2, r_2, L_3) & \\ & & L_1 \leftarrow RTL(e_1, r_d, L_2) & \\ & & \text{renvoyer } L_1 & \end{array}$$

Il faut lire ce code à l'envers : on commence par évaluer e_1 dans r_d, puis e_2 dans un nouveau pseudo-registre r_2, puis enfin on effectuer l'addition dans r_d. Un cas de base, réduit à une seule instruction, est par exemple celui du chargement d'une constante n dans r_d :

$$\begin{array}{rcll} RTL(n, r_d, L_d) & = & \text{ajouter } L_1 : \texttt{mov}\ n\ r_d\ \to L_d & (L_1 \text{ frais}) \\ & & \text{renvoyer } L_1 & \end{array}$$

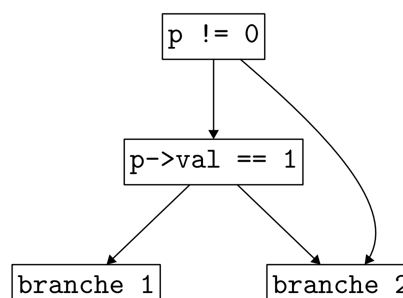
On procède de même avec les variables globales et les accès à la mémoire. Pour un appel de fonction, le principe est le même que pour une addition, avec seulement plus d'arguments à évaluer dans des pseudo-registres.

$$\begin{array}{rcll} RTL(x(e_1, \dots, e_n), r_d, L_d) & = & \text{ajouter } L_n : r_d \leftarrow \texttt{call}\ x(r_1, \dots, r_n)\ \to L_d & \\ & & L_{n-1} \leftarrow RTL(e_n, r_n, L_n) & \\ & & \dots & \\ & & L_1 \leftarrow RTL(e_1, r_1, L_2) & \\ & & \text{renvoyer } L_1 \qquad (r_1, \dots, r_n, L_1, \dots, L_n \text{ frais}) & \end{array}$$

Pour les variables locales, on se donne une table où chaque variable est associée à un pseudo-registre. La lecture ou l'écriture d'une variable locale est alors traduite avec l'instruction `mov` (opérateur binaire).

Branchements. Pour traduire les opérateurs `&&` et `||` et les instructions `if` et `while`, on va utiliser les instructions RTL de *branchement*. L'idée est de traduire un programme comme

```
if (p != 0 && p->val == 1)
  ...branche 1...
else
  ...branche 2...
```



par un graphe de flot de contrôle tel que celui donné ci-contre (où les blocs sont schématiques). En particulier, il y a deux façons de se retrouver dans la seconde branche, soit parce que `p != 0` est faux, soit parce que `p != 0` est vrai mais `p->val == 1` est faux, et on souhaite ne construire qu'une seule fois le morceau de graphe correspondant à cette seconde branche.

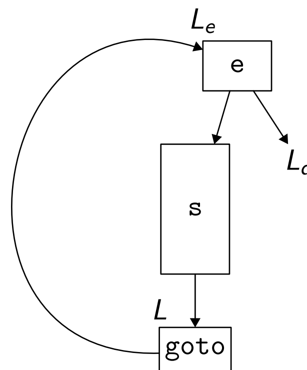
Pour y parvenir, on introduit une nouvelle fonction de traduction, $RTL_c(e, L_t, L_f)$ où e est une expression à traduire, interprétée comme une condition, et L_t et L_f deux étiquettes. L'idée est d'évaluer l'expression e puis de transférer le contrôle à l'étiquette L_t si la valeur est vraie et à l'étiquette L_f sinon. Voici une définition possible de la fonction RTL_c.

$$
\begin{array}{rcl}
RTL_c(e_1\ \texttt{\&\&}\ e_2, L_t, L_f) & = & RTL_c(e_1,\ RTL_c(e_2, L_t, L_f),\ L_f) \\[1em]
RTL_c(e_1\ \texttt{||,}\ e_2, L_t, L_f) & = & RTL_c(e_1,\ L_t,\ RTL_c(e_2, L_t, L_f)) \\[1em]
RTL_c(e_1\ \texttt{<=}\ e_2, L_t, L_f) & = & \text{ajouter } L_3 : \texttt{jle}\ r_2\ r_1\ \ \to L_t, L_f \\
& & L_2 \leftarrow RTL(e_2, r_2, L_3) \\
& & L_1 \leftarrow RTL(e_1, r_1, L_2) \\
& & \text{renvoyer } L_1 \\[1em]
RTL_c(e, L_t, L_f) & = & \text{ajouter } L_2 : \texttt{jz}\ r\ \ \to L_f, L_t \\
& & L_1 \leftarrow RTL(e, r, L_2) \\
& & \text{renvoyer } L_1
\end{array}
$$

Le dernier cas correspond à une expression e quelconque, la sémantique de C étant qu'une valeur est vraie dès lors qu'elle est non nulle, quel que soit son type. On peut bien entendu traiter plus de cas particuliers avant de recourir à ce cas général.

Traduction d'une instruction. La traduction des instructions pose le problème de l'instruction `return`, qui doit transférer le contrôle à la sortie de la fonction. On se donne un pseudo-registre r_{ret} pour recevoir le résultat de la fonction et une étiquette L_{ret} correspondant à la sortie de la fonction. La traduction d'une instruction s prend alors la forme $RTL(s, L_d)$ où L_d est l'étiquette où transférer ensuite le contrôle, si l'instruction s n'a pas conduit à `return`.

$$
\begin{array}{rcl}
RTL(\texttt{;},\ L_d) & = & \text{renvoyer } L_d \\[1em]
RTL(e\texttt{;},\ L_d) & = & \text{renvoyer } RTL(e, r_1,\ L_d) \\[1em]
RTL(\texttt{return}\ e\texttt{;},\ L_d) & = & RTL(e, r_{ret},\ L_{ret}) \\[1em]
RTL(\texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2,\ L_d) & = & RTL_c(e,\ RTL(s_1,\ L_d),\ RTL(s_2,\ L_d)) \\[1em]
RTL(\texttt{\{}s_1 \dots s_n\texttt{\}},\ L_d) & = & L_n \leftarrow RTL(s_n,\ L_d) \\
& & \dots \\
& & L_1 \leftarrow RTL(s_1,\ L_2) \\
& & \text{renvoyer } L_1
\end{array}
$$



Pour traduire une `while(e)s`, il faut construire une boucle dans le graphe de flot de contrôle. Vu que l'on construit le code de bas en haut, cela pose une petite difficulté. On s'en sort en utilisant l'instruction RTL `goto` pour fermer la boucle *a posteriori*, comme ceci :

$$
\begin{array}{rcl}
RTL(\texttt{while(}e\texttt{)}s,\ L_d) & = & L_e \leftarrow RTL_c(e, RTL(s,\ L),\ L_d) \\
& & \text{ajouter } L : \texttt{goto}\ L_e \\
& & \text{renvoyer } L_e
\end{array}
$$

Traduction d'une fonction. Pour traduire une fonction, on commence par allouer des pseudo-registres frais pour ses arguments, son résultat et ses variables locales. On crée une étiquette fraîche L_d pour la *sortie* de la fonction. Enfin, on traduit le corps s de la fonction avec $RTL(s, L_d)$ et on note le résultat comme étant l'étiquette d'*entrée* de la fonction. Pour la fonction `fact`, on obtient ceci au final :

```
#2 fact(#1)
  entry : L10
  exit  : L1
  L10: mov #1 #6  -> L9
  L9 : jle $1 #6  -> L8, L7
  L8 : mov $1 #2  -> L1
```

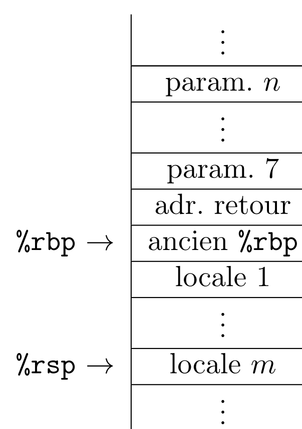
```
L7: mov #1 #5            -> L6
L6: add $-1 #5           -> L5
L5: #3 <- call fact(#5) -> L4
L4: mov #1 #4            -> L3
L3: mov #3 #2            -> L2
L2: imul #4 #2           -> L1
```

Chaque fonction est traduite indépendamment, avec son propre graphe de flot de contrôle. En ce sens, notre traduction est *intraprocédurale i.e.* sans connaissance des autres fonctions et en particulier sans connaissances des points d'appel de cette fonction. Par opposition, une analyse *interprocédurale* pourrait prendre en compte le contexte d'appel des fonctions.

9.3 Production de code ERTL

La troisième phase consiste en une traduction vers un nouveau langage, appelé ERTL pour *Explicit Register Transfer Language*, avec l'objectif d'expliciter les *conventions d'appel*. En particulier, nous allons expliciter le fait que, dans un appel, six premiers paramètres sont passés dans `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` et les suivants sur la pile, et que le résultat est renvoyé dans `%rax`. Nous allons appliquer ces conventions d'appel autant pour nos propres fonctions que pour d'éventuelles fonctions de bibliothèque utilisées dans nos programmes, telles que `putchar` ou `sbrk`, sans distinction. Par ailleurs, nous allons expliciter le fait que l'instruction de division `idivq` impose l'usage des registres `%rax` et `%rdx`. Enfin, nous allons assurer que les registres *callee-saved* (`%rbx`, `%r12`, `%r13`, `%r14`, `%r15`, `%rbp`) sont bien préservés par l'appelé.

Le tableau d'activation s'organise ainsi comme illustré ci-contre. Au delà du sixième, les arguments sont passés sur la pile. Vient ensuite l'adresse de retour placée par `call`, puis la sauvegarde du registre `%rbp`. Enfin, on trouve la place allouée par les données locales, c'est-à-dire tout ce qu'on n'aura pas pu allouer dans des registres. C'est donc l'allocation de registres qui déterminera m. On se sert ici du registre `%rbp` pour retrouver facilement les paramètres d'une part et les variables locales d'autre part. En effet, la pile est susceptible d'être utilisée pour un appel à une fonction ayant plus de six arguments et la valeur de `%rsp` varie alors. Il serait alors plus complexe de retrouver les différents éléments du tableau d'activation à l'aide de `%rsp` uniquement.



Les instructions ERTL sont données figure 9.5. Les dix premières sont identiques aux instructions RTL, à ceci près que r peut maintenant désigner également un registre physique. L'instruction `call` est simplifiée, car il ne reste plus que le nom de la fonction à appeler. En effet, de nouvelles instructions vont être insérées pour charger les paramètres

dans des registres et sur la pile, et pour récupérer le résultat dans `%rax`. On conserve néanmoins l'information du nombre de paramètres passés dans des registres (qui sera utilisée par la phase 4). Ainsi, on écrit `call` *fact*(1) pour signifier un appel à *fact* avec un seul argument passé par registre.

Viennent enfin de nouvelles instructions pour manipuler la pile. Les deux instructions `alloc_frame` et `delete_frame` alloue et désalloue respectivement la partie locale du tableau d'activation. Ces instructions n'ont pas d'argument, car la taille n'est pas encore connue. Elle ne le sera qu'après l'allocation de registres, réalisée dans la phase suivante. Les instructions `push_param` et `get_param` permettent respectivement à l'appelant de placer un paramètre sur la pile et à l'appelé de le récupérer. L'argument n de `get_param` est une position relative par rapport à `%rbp`. Par exemple, le dernier argument mis sur la pile, le cas échéant, se trouve à l'adresse `%rbp` + 16.

Pour traduire le code RTL en code ERTL, on ne change pas la structure du graphe de flot de contrôle ; on se contente d'insérer de *nouvelles instructions*. On le fait principalement à trois endroits :

— au début de chaque fonction, pour allouer le tableau d'activation, sauvegarder les registres *callee-saved* et copier les paramètres dans les pseudo-registres correspondants ;

— à la fin de chaque fonction, pour copier le pseudo-registre contenant le résultat dans `%rax`, restaurer les registres *callee-saved* et désallouer le tableau d'activation ;

— à chaque appel, pour copier les pseudo-registres contenant les paramètres dans `%rdi`, ..., `%r9` et sur la pile avant l'appel et copier `%rax` dans le pseudo-registre contenant le résultat après l'appel.

Traduction des instructions. Les instructions qui ne sont pas modifiées sont le chargement d'une constante, la lecture et l'écriture d'une variable globale, la lecture et l'écriture en mémoire, une opération unaire, une opération binaire autre que la division et les opérations de branchement. Ces instructions restent placées aux mêmes étiquettes et transfèrent le contrôle vers les mêmes étiquettes qu'auparavant.

Montrons maintenant les changements apportés dans ERTL. On commence par le cas de la division. En RTL, on a pour l'instant une instruction

$$L_1 :\ \texttt{div}\ r_1\ r_2 \rightarrow L$$

où r_1 et r_2 sont deux pseudo-registres. Attention au sens : on divise ici r_2 par r_1. Dans ERTL, cela devient trois instructions[3]

$$\begin{aligned} L_1 &:\ \texttt{mov}\ r_2\ \texttt{\%rax} \rightarrow L_2 \\ L_2 &:\ \texttt{div}\ r_1\ \texttt{\%rax} \rightarrow L_3 \\ L_3 &:\ \texttt{mov}\ \texttt{\%rax}\ r_2 \rightarrow L \end{aligned}$$

avec L_2 et L_3 des étiquettes fraîches. Même si on a ajouté de nouvelles instructions, le point d'entrée est toujours L_1 et le point de sortie est toujours L.

Considérons maintenant un appel de fonction, c'est-à-dire une instruction RTL

$$L_1 :\ \texttt{call}\ r \leftarrow f(r_1, \ldots, r_n) \rightarrow L$$

3. L'instruction x86-64 `idiv` r divise en réalité l'entier représenté par la concaténation de `%rdx` et de `%rax` par r, puis met le quotient dans `%rax` et le reste dans `%rdx`. Nous expliciterons cela un peu plus tard, dans la phase suivante.

instr	::=	`mov` n $r \to L$	chargement d'une constante
		`mov` x $r \to L$	lecture d'une variable globale
		`mov` r $x \to L$	écriture d'une variable globale
		`load` $n(r)$ $r \to L$	lecture en mémoire
		`store` r $n(r) \to L$	écriture en mémoire
		unop $r \to L$	opération unaire
		binop r $r \to L$	opération binaire
		ubranch $r \to L, L$	branchement unaire
		bbranch r $r \to L, L$	branchement binaire
		`call` $r \leftarrow x(r, \ldots, r) \to L$	appel de fonction
		`goto` $\to L$	branchement inconditionnel
ubranch	::=	`jz` | `jnz` | `jl` n | `jle` n | ...	branchement unaire
bbranch	::=	`jl` | `jle` | `jg` | `jge` | ...	branchement binaire

FIGURE 9.4 – Langage RTL.

instr	::=	`mov` n $r \to L$	chargement d'une constante
		`mov` x $r \to L$	lecture d'une variable globale
		`mov` r $x \to L$	écriture d'une variable globale
		`load` $n(r)$ $r \to L$	lecture en mémoire
		`store` r $n(r) \to L$	écriture en mémoire
		unop $r \to L$	opération unaire
		binop r $r \to L$	opération binaire
		ubranch $r \to L, L$	branchement unaire
		bbranch r $r \to L, L$	branchement binaire
		`goto` $\to L$	branchement inconditionnel
		`call` $x(n) \to L$	appel de fonction
		`return`	instruction explicite de retour
		`alloc_frame` $\to L$	allouer le tableau d'activation
		`delete_frame` $\to L$	désallouer le tableau d'activation
		`push_param` $r \to L$	empiler la valeur de r
		`get_param` n $r \to L$	accéder à un paramètre sur la pile

FIGURE 9.5 – Langage ERTL.

où r, r_1, $r_2, \ldots, r_n$ sont des pseudo-registres. Elle est traduite en une séquence d'instructions ERTL pour effectuer, dans cet ordre, les tâches suivantes :

1. copier $\min(n, 6)$ arguments $r_1, r_2, \ldots$ dans `%rdi`, `%rsi`, …, `%r9` ;
2. si $n > 6$, passer les autres sur la pile avec `push_param` ;
3. exécuter `call` $f(\min(n, 6))$;
4. copier `%rax` dans r ;
5. si $n > 6$, dépiler $8 \times (n - 6)$ octets.

Cette séquence d'instructions commence à la même étiquette L_1 et transfert le contrôle au final à la même étiquette L qu'auparavant. Par exemple, le code RTL

```
 L5: #3 <- call fact(#5)  -> L4
```

est traduit en ERTL par les trois instructions suivantes :

```
 L5: mov #5 %rdi   -> L12
L12: call fact(1)  -> L11
L11: mov %rax #3   -> L4
```

Traduction d'une fonction. Pour traduire une fonction RTL en une fonction ERTL, on traduit son graphe de flot de contrôle en traduisant chaque instruction RTL comme expliqué ci-dessus. Par ailleurs, on ajoute de nouvelles instructions ERTL en entrée et en sortie de fonction. À l'entrée de la fonction, on commence par allouer le tableau d'activation, en ajoutant l'instruction `alloc_frame`. Puis, on sauvegarde les registres *callee-saved*. Pour cela, on se donne autant de pseudo-registres frais qu'il y a de registres *callee-saved* et on copie la valeur de ces derniers dans ces pseudo-registres. Enfin, on copie les paramètres dans leurs pseudo-registres. S'il s'agit de paramètres passés dans des registres, on les copie avec `mov`. Sinon, on va les chercher sur la pile avec `get_param`. Si on prend l'exemple de la fonction `fact` et si on suppose pour simplifier que les seuls registres *callee-saved* sont `%rbx` et `%r12`, on obtient ceci :

RTL

```
#2 fact(#1)
  entry : L10




  L10: mov #1 #6    -> L9
  ...
```

ERTL

```
fact(1)
  entry : L17
  L17: alloc_frame  -> L16
  L16: mov %rbx #7  -> L15
  L15: mov %r12 #8  -> L14
  L14: mov %rdi #1  -> L10
  L10: mov #1 #6    -> L9
  ...
```

Les registres `%rbx` et `%r12` sont sauvegardés respectivement dans `#7` et `#8`. On note que le point d'entrée du graphe de flot de contrôle a changé ; c'est maintenant `L17` au lieu de `L10`. On a rajouté quatre instructions, qui se poursuivent ensuite par ce qui était auparavant le code RTL, à partir de `L10`.

À la sortie de la fonction, on ajoute une instruction `mov` pour copier le pseudo-registre contenant le résultat dans `%rax`. Puis on restaure les registres `callee-saved`, en copiant les valeurs depuis les pseudo-registres utilisés pour leur sauvegarde. Enfin, on désalloue le

tableau d'activation avec `delete_frame` avant de faire `return`. Pour la fonction `fact`, on obtient ceci :

RTL

```
#2 fact(#1)
  entry : L10
  exit  : L1
  ...
  L8 : mov $1 #2    -> L1
  ...
  L2 : imul #4 #2   -> L1
```

ERTL

```
fact(1)
  entry : L17

  ...
  L8 : mov $1 #2    -> L1
  ...
  L2 : imul #4 #2   -> L1
  L1 : mov #2 %rax  -> L21
  L21: mov #7 %rbx  -> L20
  L20: mov #8 %r12  -> L19
  L19: delete_frame -> L18
  L18: return
```

Dans le code RTL, on avait deux sauts vers l'étiquette de sortie `L1`, correspondant aux deux instructions `return` dans le code C. Dans le code ERTL, on a toujours ces deux sauts vers `L1`, mais des instructions ERTL ont maintenant été ajoutées à cet endroit-là. En particulier, le résultat, contenu dans `#2`, est copié dans `%rax` et les registres *callee-saved* sont restaurés en reprenant les valeurs dans `#7` et `#8`. On note qu'il n'y a plus de notion d'étiquette de sortie dans le code ERTL, car on a maintenant une instruction `return` explicite.

L'intégralité du code ERTL de la fonction `fact` est donné figure 9.6 page 167. C'est encore loin de ce que l'on imagine être un bon code x86-64 pour cette fonction. À ce point, il faut comprendre plusieurs choses. D'une part, l'allocation de registres (phase 4) tâchera d'associer des registres physiques aux pseudo-registres de manière à limiter l'usage de la pile mais aussi de supprimer certaines instructions. Si par exemple on réalise `#8` par `%r12`, on supprime tout simplement les deux instructions `L15` et `L20`. D'autre part, le code n'est pas encore organisé linéairement (le graphe est seulement affiché de manière arbitraire). Ce sera le travail de la phase 5, qui tâchera notamment de minimiser les sauts.

Optimisation de l'appel terminal. C'est au niveau de la traduction RTL $\rightarrow$ ERTL qu'il faut réaliser l'optimisation des *appels terminaux* (voir section 7.2). En effet, les instructions à produire ne sont pas les mêmes et ce changement aura une influence dans la phase suivante d'allocation des registres.

Il y a cependant une difficulté si la fonction appelée par un appel terminal n'a pas le même nombre d'arguments passés sur la pile ou de variables locales. En effet, il faut alors modifier la structure du tableau d'activation. Il y a au moins deux solutions. On peut par exemple limiter l'optimisation de l'appel terminal aux cas où la structure du tableau d'activation ne change pas. C'est le cas notamment s'il s'agit d'un appel terminal d'une fonction récursive à elle-même. Une autre solution consiste, pour l'appelant à modifier le tableau d'activation et à transférer le contrôle à l'appelé *après* l'instruction de création de son tableau d'activation.

9.4 Production de code LTL

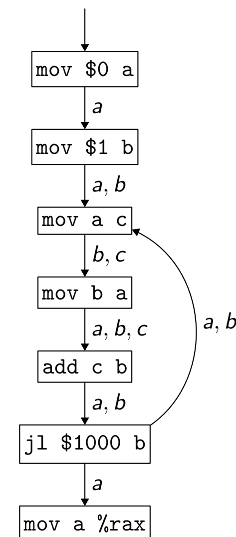
La quatrième phase est la traduction vers un langage appelé *LTL* pour *Location Transfer Language*. Il s'agit de faire disparaître les pseudo-registres au profit de registres physiques, de préférence, et d'emplacements de pile, sinon. C'est ce que l'on appelle l'*allocation de registres*. Il s'agit d'une phase complexe du compilateur, que l'on va elle-même décomposer en plusieurs étapes. On va commencer par réaliser une analyse de *durée de vie*. Il s'agit de déterminer à quels moments précis la valeur d'un pseudo-registre est nécessaire pour la suite du calcul. Avec cette information, on va construire un *graphe d'interférence*. Il s'agit d'un graphe indiquant quels sont les pseudo-registres qui ne peuvent pas être réalisés par le même emplacement. Enfin, on va faire l'allocation de registres proprement dite en *coloriant ce graphe*.

9.4.1 Analyse de durée de vie

Dans ce qui suit, on appelle *variable* un pseudo-registre ou un registre physique. L'analyse de durée de vie concerne toutes les variables, avec le sens donné par la définition suivante.

Définition 27 (variable vivante)**.** En un point de programme, une variable est dite *vivante* si la valeur qu'elle contient est susceptible d'être utilisée dans la suite de l'exécution. □

On emploie les mots « est susceptible d'être utilisée » car la propriété « est utilisée » n'est pas décidable. On va donc se contenter d'une approximation. Cette approximation devra être correcte, au sens où une réponse négative, c'est-à-dire la variable v n'est pas vivante, signifie qu'il est garantie que la valeur de v n'est plus utilisée dans la suite du calcul.



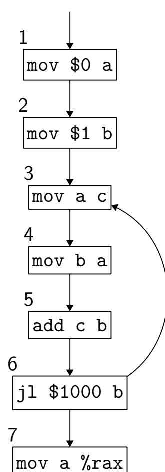
Prenons l'exemple du code ERTL dont le graphe de flot de contrôle est représenté ci-contre. Les variables sont ici `a`, `b`, `c` et `%rax`. Sur chaque arête du graphe, on a indiqué les variables vivantes. Par exemple, `a` est vivante sur la deuxième arête en partant du haut, car sa valeur est utilisée dans la troisième instruction (`mov a c`) et `a` n'a pas été affectée entre-temps. La variable `b`, en revanche, n'est pas vivante sur cette arête, car sa valeur est définie par la seconde instruction (`mov $1 b`). Mais elle est vivante sur la troisième arête, car sa valeur est utilisée par l'instruction `mov b a`, qui la copie dans `a`. La variable `%rax` n'est jamais vivante, car sa valeur n'est jamais utilisée, mais seulement définie par la toute dernière instruction. On prendra le temps de bien comprendre cet exemple.

La notion de variable vivante se déduit des *définitions* et des *utilisations* des variables effectuées par chaque instruction. Ainsi, l'instruction `add` r_1 r_2 utilise les variables r_1 et r_2, et définit la variable r_1. Pour une instruction située à l'étiquette l du graphe, on note *def*(l) l'ensemble des variables définies par cette instruction et *use*(l) l'ensemble des variables utilisées par cette instruction.

Pour calculer les variables vivantes, il est commode de les associer non pas aux arêtes mais plutôt aux *nœuds* du graphe de flot de contrôle, c'est-à-dire à chaque instruction. Mais il faut alors distinguer les variables *vivantes à l'entrée* d'une instruction et les variables *vivantes à la sortie*. Pour une instruction située à l'étiquette l du graphe, on note $in(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes arrivant sur l et $out(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes sortant de l. Les équations qui définissent $in(l)$ et $out(l)$ sont alors les suivantes :

$$\begin{cases} in(l) & = & use(l) \cup (out(l) \backslash def(l)) \\ out(l) & = & \bigcup_{s \in succ(l)} in(s) \end{cases}$$

Il s'agit d'équations récursives dont la plus petite solution est celle qui nous intéresse. Nous sommes dans le cas d'une fonction monotone sur un domaine fini (les parties de l'ensemble fini des variables) et nous pouvons donc appliquer le théorème de Knaster–Tarski (voir page 64). Cela signifie que l'on part d'ensembles $in(l)$ et $out(l)$ vides pour toutes les instructions, puis on applique les équations ci-dessus jusqu'à obtenir un point fixe. Si on reprend l'exemple ci-dessus, en numérotant les instructions de 1 à 7 du haut vers le bas, on converge après sept itérations [4] :



	use	*def*	itération 1 *in*	itération 1 *out*	itération 2 *in*	itération 2 *out*		itération 7 *in*	itération 7 *out*
1		a					…		a
2		b				a	…	a	a, b
3	a	c	a		a	b	…	a, b	b, c
4	b	a	b		b	b, c	…	b, c	a, b, c
5	b, c	b	b, c		b, c	b	…	a, b, c	a, b
6	b		b		b	a	…	a, b	a, b
7	a		a		a		…	a	

En supposant un graphe de flot de contrôle contenant N sommets et N variables, un calcul brutal a une complexité $O(N^4)$ dans le pire des cas, ce qui n'est pas acceptable en pratique. Notre graphe de flot de contrôle pour la fonction `fact` contient déjà 22 sommets et plus de 20 variables, alors que le code C ne fait que deux lignes. Fort heureusement, on peut améliorer l'efficacité du calcul ci-dessus de plusieurs façons. En premier lieu, on peut faire les calculs dans l'« ordre inverse » du graphe de flot de contrôle et en calculant *out* avant *in*. Sur l'exemple précédent, on converge alors en trois itérations au lieu de sept. Ensuite, on peut fusionner les sommets qui n'ont qu'un unique prédécesseur et qu'un unique successeur [5]. Enfin, on peut utiliser un algorithme plus subtil qui ne recalcule que les valeurs de *in* et *out* qui peuvent avoir changé ; c'est l'algorithme de Kildall. L'idée est la suivante : si $in(l)$ change, alors il faut refaire le calcul pour les prédécesseurs de l uniquement. On en déduit l'algorithme suivant :

4. Il n'y a pas de rapport, *a priori*, entre le nombre d'instructions et le nombre d'itérations.
5. On appelle cela un *bloc de base* (en anglais *basic block*).

```
soit WS un ensemble contenant tous les sommets
tant que WS n'est pas vide
    extraire un sommet l de WS
    old_in <- in(l)
    out(l) <- ...
    in(l)  <- ...
    si in(l) est différent de old_in(l) alors
        ajouter tous les prédécesseurs de l dans WS
```

Calcul de *def* et *use*. Le calcul des ensembles *def*(l) (définitions) et *use*(l) (utilisations) est immédiat pour la plupart des instructions ERTL. Voici tous les cas simples :

	def	*use*
`mov` n r	$\{r\}$	$\emptyset$
`mov` x r	$\{r\}$	$\emptyset$
`mov` r x	$\emptyset$	$\{r\}$
`load` $n(r_1)$ r_2	$\{r_2\}$	$\{r_1\}$
`store` r_1 $n(r_2)$	$\emptyset$	$\{r_1, r_2\}$
unop r	$\{r\}$	$\{r\}$
binop r_1 r_2	$\{r_2\}$	$\{r_1, r_2\}$

	def	*use*
ubranch r	$\emptyset$	$\{r\}$
bbranch r_1 r_2	$\emptyset$	$\{r_1, r_2\}$
`goto`	$\emptyset$	$\emptyset$
`alloc_frame`	$\emptyset$	$\emptyset$
`delete_frame`	$\emptyset$	$\emptyset$
`push_param` r	$\emptyset$	$\{r\}$
`get_param` n r	$\{r\}$	$\emptyset$

On fait cependant un cas particulier pour la division, qui prend le dividende dans l'ensemble `%rdx` : `%rax` et place le quotient et le reste respectivement dans `%rax` et `%rdx`.

	def	*use*
`div` r `%rax`	{`%rax`, `%rdx`}	{`%rax`, `%rdx`, r}

Reste enfin les instructions `call` et `return`. Pour une instruction `call` $f(k)$, l'entier k nous indique le nombre de paramètres passés dans des registres, ce qui définit les registres utilisés par l'appel. On exprime par ailleurs que tous les registres *caller-saved* peuvent être écrasés par l'appel.

	def	*use*
`call` $f(k)$	*caller-saved*	les k premiers de `%rdi`,`%rsi`,...,`%r9`

Enfin, pour l'instruction `return`, on exprime le fait que `%rax` et tous les registres *callee-saved* sont susceptible d'être utilisés.

	def	*use*
`return`	$\emptyset$	{`%rax`} $\cup$ *callee-saved*

La figure 9.7 donne le résultat de l'analyse de durée de vie pour la fonction `fact`.

9.4.2 Graphe d'interférence

On va maintenant exploiter le résultat de l'analyse de durée de vie pour construire un *graphe d'interférence* qui exprime les contraintes sur les emplacements possibles pour les pseudo-registres.

```
fact(1)
  entry : L17
  L17: alloc_frame  -> L16
  L16: mov %rbx #7  -> L15
  L15: mov %r12 #8  -> L14
  L14: mov %rdi #1  -> L10
  L10: mov #1 #6    -> L9
  L9 : jle $1 #6 -> L8, L7
  L8 : mov $1 #2    -> L1
  L1 : goto         -> L22
  L22: mov #2 %rax  -> L21
  L21: mov #7 %rbx  -> L20
  L20: mov #8 %r12  -> L19
  L19: delete_frame -> L18
  L18: return
  L7 : mov #1 #5    -> L6
  L6 : add $-1 #5   -> L5
  L5 : goto         -> L13
  L13: mov #5 %rdi  -> L12
  L12: call fact(1) -> L11
  L11: mov %rax #3  -> L4
  L4 : mov #1 #4    -> L3
  L3 : mov #3 #2    -> L2
  L2 : imul #4 #2   -> L1
```

FIGURE 9.6 – Code ERTL pour la fonction `fact`.

	in	*out*
`L17: alloc_frame --> L16`	%r12,%rbx,%rdi	%r12,%rbx,%rdi
`L16: mov %rbx #7 --> L15`	%r12,%rbx,%rdi	#7,%r12,%rdi
`L15: mov %r12 #8 --> L14`	#7,%r12,%rdi	#7,#8,%rdi
`L14: mov %rdi #1 --> L10`	#7,#8,%rdi	#1,#7,#8
`L10: mov #1 #6   --> L9`	#1,#7,#8	#1,#6,#7,#8
`L9 : jle $1 #6 -> L8, L7`	#1,#6,#7,#8	#1,#7,#8
`L8 : mov $1 #2   --> L1`	#7,#8	#2,#7,#8
`L1 : goto        --> L22`	#2,#7,#8	#2,#7,#8
`L22: mov #2 %rax --> L21`	#2,#7,#8	#7,#8,%rax
`L21: mov #7 %rbx --> L20`	#7,#8,%rax	#8,%rax,%rbx
`L20: mov #8 %r12 --> L19`	#8,%rax,%rbx	%r12,%rax,%rbx
`L19: delete_frame--> L18`	%r12,%rax,%rbx	%r12,%rax,%rbx
`L18: return`	%r12,%rax,%rbx	
`L7 : mov #1 #5   --> L6`	#1,#7,#8	#1,#5,#7,#8
`L6 : add $-1 #5  --> L5`	#1,#5,#7,#8	#1,#5,#7,#8
`L5 : goto        --> L13`	#1,#5,#7,#8	#1,#5,#7,#8
`L13: mov #5 %rdi --> L12`	#1,#5,#7,#8	#1,#7,#8,%rdi
`L12: call fact(1)--> L11`	#1,#7,#8,%rdi	#1,#7,#8,%rax
`L11: mov %rax #3 --> L4`	#1,#7,#8,%rax	#1,#3,#7,#8
`L4 : mov #1 #4   --> L3`	#1,#3,#7,#8	#3,#4,#7,#8
`L3 : mov #3 #2   --> L2`	#3,#4,#7,#8	#2,#4,#7,#8
`L2 : imul #4 #2  --> L1`	#2,#4,#7,#8	#2,#7,#8

FIGURE 9.7 – Analyse de durée de vie pour la fonction `fact`.

Définition 28 (interférence)**.** On dit que deux variables v_1 et v_2 *interfèrent* si elles ne peuvent pas être réalisées par le même emplacement (registre physique ou emplacement mémoire). □

Comme l'interférence n'est pas décidable, on va se contenter de conditions suffisantes. Soit une instruction qui définit une variable v : toute autre variable w vivante à la sortie de cette instruction peut interférer avec v. Cependant, dans le cas particulier d'un instruction

`mov` w v

on ne souhaite pas déclarer que v et w interfèrent car il peut être précisément intéressant de réaliser v et w par le même emplacement et d'éliminer ainsi une ou plusieurs instructions. On adopte donc la définition suivante.

Définition 29 (graphe d'interférence)**.** Le *graphe d'interférence* d'une fonction est un graphe non orienté dont les sommets sont les variables de cette fonction et dont les arêtes sont de deux types : interférence ou préférence. Pour chaque instruction qui définit une variable v et dont les variables vivantes en sortie, autres que v, sont $w_1, \dots, w_n$, on procède ainsi :

- si l'instruction n'est pas une instruction `mov` w v, on ajoute les n arêtes d'interférence $v - w_i$;
- s'il s'agit d'une instruction `mov` w v, on ajoute les arêtes d'interférence $v - w_i$ pour tous les w_i différents de w et on ajoute l'arête de préférence $v - w$.

Si une arête $v - w$ est à la fois de préférence et d'interférence, on conserve uniquement l'arête d'interférence. □

La figure 9.8 montre le graphe obtenu pour la fonction `fact`, en considérant ses 8 pseudo-registres et uniquement 10 registres physiques (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, `%rax`, `%r10`, `%rbx` et `%r12`), car on a omis certains registres pour simplifier. Les arêtes de préférence sont indiquées en pointillés. On voit par exemple qu'on a une arête de préférence entre `#2` et `%rax`, qui provient de l'instruction `mov #2 %rax`. On observe que certains sommets sont de fort degré. Pour `#7` et `#8`, cela s'explique par le fait qu'on y a sauvegardés les registres *callee-saved* et donc que leur durée de vie s'étend sur toute la fonction. Le sommet `#1` est également de fort degré, car il contient l'argument de la fonction `fact`, qui est utilisé sur l'ensemble de la fonction et en particulier après l'appel récursif.

9.4.3 Coloriage du graphe d'interférence

Une fois le graphe d'interférence construit, on peut voir le problème de l'allocation de registres comme un problème de *coloriage de graphe*, où les couleurs sont les registres physiques. Deux sommets liés par une arête d'interférence ne peuvent pas recevoir la même couleur. Deux sommets liés par une arête de préférence doivent recevoir la même couleur autant que possible. Il y a dans le graphe des sommets qui sont des registres physiques, c'est-à-dire des sommets déjà coloriés.

Si on observe attentivement le graphe de la fonction `fact`, on s'aperçoit que le coloriage est impossible. En effet, on a seulement deux couleurs pour colorier `#1`, `#7` et `#8`, à savoir `%rbx` et `%r12`, et ces trois pseudo-registres interfèrent deux à deux. Fort heureusement, nous ne sommes pas dans une situation habituelle de coloriage de graphe : si un sommet

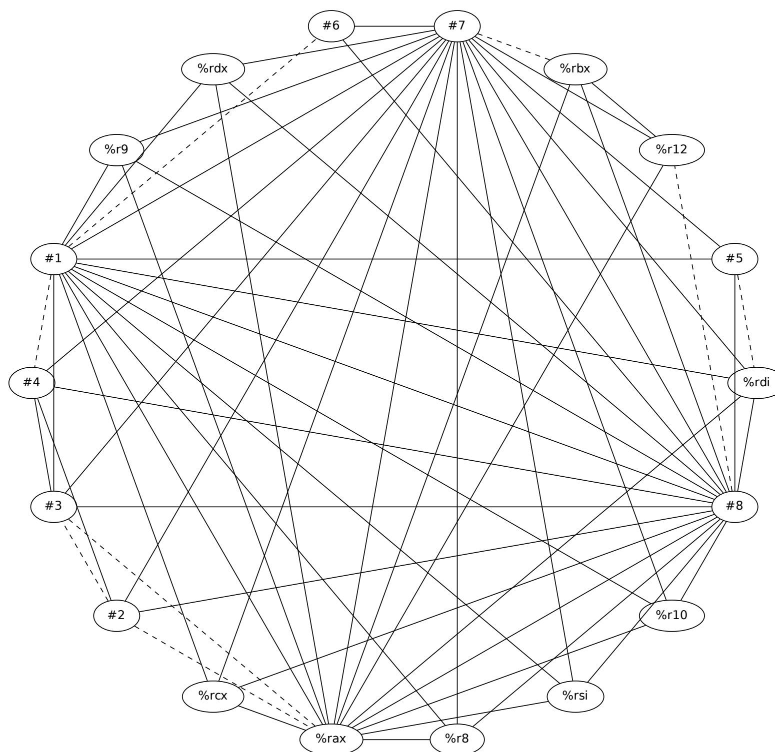


FIGURE 9.8 – Graphe d'interférence pour la fonction `fact`.

ne peut être colorié, on peut toujours l'allouer sur la pile. On dit alors qu'il est *vidé en mémoire* (en anglais on parle de *spilled register*).

Quand bien même le graphe serait effectivement coloriable, le déterminer serait trop coûteux car il s'agit là d'un problème NP-complet. On va donc colorier en utilisant des *heuristiques*, avec pour objectifs une complexité linéaire ou quasi-linéaire et une bonne exploitation des arêtes de préférence. Une technologie répandue aujourd'hui s'appelle *Iterated Register Coalescing*. Elle exploite les idées suivantes.

Notons K le nombre de couleurs, c'est-à-dire le nombre de registres physiques (10 dans notre exemple). Une première idée, due à Kempe, date de 1879 : si un sommet a un degré strictement inférieur à K, alors on peut le retirer du graphe, colorier le reste, et on sera ensuite assuré de pouvoir lui donner une couleur. Cette étape est appelée *simplification*. Les sommets retirés sur donc mis sur une pile. Comme retirer un sommet diminue le degré d'autres sommets, cela peut donc produire de nouveaux candidats à la simplification.

Lorsqu'il ne reste que des sommets de degré supérieur ou égal à K, on en choisit un comme candidat à être vidé en mémoire (en anglais *potential spill*). Il est alors retiré du graphe, mis sur la pile et le processus de simplification peut reprendre. On choisit de préférence un sommet qui est peu utilisé, car les accès à la mémoire coûtent cher, et qui a un fort degré, pour favoriser de futures simplifications.

Lorsque le graphe est vide, on commence le processus de coloration proprement dit, appelé *sélection*. On dépile les sommets un à un et pour chacun on lui affecte une couleur de la manière suivante. S'il s'agit d'un sommet de faible degré, on est assuré de lui trouver une couleur. S'il s'agit au contraire d'un sommet de fort degré, c'est-à-dire d'un candidat à être vidé en mémoire, alors de deux choses l'une : soit il peut être tout de même colorié

car ses voisins utilisent moins de K couleurs au total (on parle de *coloriage optimiste*) ; soit il ne peut pas être colorié et doit être effectivement être vidé en mémoire (en anglais *actual spill*).

Enfin, il convient d'utiliser au mieux les arêtes de préférence. Pour cela, on utilise une technique appelée *coalescence* (en anglais *coalescing*) qui consiste à fusionner deux sommets du graphe. Comme cela peut augmenter le degré du sommet résultant, on ajoute un critère suffisant pour ne pas détériorer la K-colorabilité. Voici un exemple de tel critère :

Définition 30 (critère de George)**.** Un sommet pseudo-registre v_2 peut être fusionné avec un sommet v_1 si tout voisin de v_1 qui est un registre physique ou de degré $\geq K$ est également voisin de v_2. De même, un sommet physique v_2 peut être fusionné avec un sommet v_1 si tout voisin de v_1 qui est un pseudo-registre ou de degré $\geq K$ est également voisin de v_2. □

Un pseudo-code pour l'allocation de registres est donné figure 9.9. Il s'écrit naturellement sous la forme de cinq fonctions mutuellement récursives, qui prennent en argument le graphe g à colorier et qui renvoient le coloriage. La fonction `fusionner(g, v1, v2)` construit un nouveau graphe en fusionnant les sommets `v1` et `v2` en un seul sommet `v2`. Pour la notion de coût utilisée dans la fonction `spill`, on peut utiliser par exemple

$$\textit{coût}(v) = \frac{\textit{nombre d'utilisations de } v}{\textit{degré de } v}$$

Si on applique cet algorithme sur le graphe d'interférence de la fonction `fact` (figure 9.8 page 169), on va effectuer successivement les actions suivantes :

1. `simplify` appelle `coalesce` qui fusionne `#2- - -#3`
2. `simplify` appelle `coalesce` qui fusionne `#4- - -#1`
3. `simplify` appelle `coalesce` qui fusionne `#6- - -#1`
4. `simplify` appelle `coalesce` qui fusionne `#3- - -%rax`
5. `simplify` appelle `coalesce` qui fusionne `#5- - -%rdi`
6. `simplify` appelle `coalesce`. Cette fois il n'y a plus d'arête de préférence satisfaisant le critère de George. `coalesce` appelle alors `freeze` qui appelle `spill` qui appelle `select` avec `#7`.
7. `simplify` appelle `select` avec `#1`
8. `simplify` appelle `coalesce` qui fusionne `#8- - -%r12`

À ce point, le graphe ne contient plus de pseudo-registre. Du coup, `simplify` appelle `coalesce` qui appelle `freeze` qui appelle `spill` qui renvoie un coloriage vide. On dépile donc maintenant un par un les sommets retirés du graphe, en leur attribuant une couleur à chaque fois, soit dans `coalesce`, soit dans `select`.

1. `coalesce` attribue à `#8` la couleur `%r12`
2. `select` attribue à `#1` la couleur `%rbx`
3. `select` attribue à `#7` la couleur `spill`
4. `coalesce` attribue à `#5` la couleur `%rdi`
5. `coalesce` attribue à `#3` la couleur `%rax`

```
simplify(g) =
  s'il existe un sommet v sans arête de préférence
     de degré minimal et < K
  alors
    select(g, v)
  sinon
    coalesce(g)

coalesce(g) =
  s'il existe une arête de préférence v1-v2
     satisfaisant le critère de George
  alors
    g <- fusionner(g, v1, v2)
    c <- simplify(g)
    c[v1] <- c[v2]
    renvoyer c
  sinon
    freeze(g)

freeze(g) =
  s'il existe un sommet v de degré minimal < K
  alors
    g <- oublier les arêtes de préférence de v
    simplify(g)
  sinon
    spill(g)

spill(g) =
  si g est vide
  alors
    renvoyer le coloriage vide
  sinon
    choisir un sommet v de coût minimal
    select(g, v)

select(g, v) =
  c <- simplify(g privé de v)
  s'il existe une couleur r possible pour v
  alors
    c[v] <- r
  sinon
    c[v] <- spill
  renvoyer c
```

FIGURE 9.9 – Pseudo-code pour l'allocation de registres.

6. `coalesce` attribue à `#6` la couleur de `#1`, c'est-à-dire `%rbx`
7. `coalesce` attribue à `#4` la couleur de `#1`, c'est-à-dire `%rbx`
8. `coalesce` attribue à `#2` la couleur de `#3`, c'est-à-dire `%rax`

Il reste à allouer sur la pile les registres vidés en mémoire. On peut chercher à minimiser cet espace. En effet, plusieurs pseudo-registres peuvent occuper le même emplacement de pile s'ils n'interfèrent pas. Par ailleurs, on a toujours envie de tenir compte des arêtes de préférence, pour éliminer des opérations `mov` entre registres vidés en mémoire qui coûteraient très cher. C'est de nouveau un problème de coloriage de graphe, mais cette fois avec une infinité de couleurs possibles, chaque couleur correspondant à un emplacement de pile différent. Sur notre exemple de la fonction `fact`, il n'y a qu'un seul registre vidé en mémoire, à savoir `#7`. On réserve donc un unique emplacement sur la pile et `#7` est alloué à l'emplacement `-8(%rbp)`.

9.4.4 Traduction vers LTL

On peut maintenant traduire notre programme ERTL vers le langage LTL. Les instructions LTL sont données figure 9.10. Dans ces instructions, r désigne nécessairement un registre physique et o désigne une opérande qui peut être un registre physique ou un emplacement de pile. Les instructions `alloc_frame`, `delete_frame` et `get_param` ont disparu, au profit de manipulation explicite de `%rsp` et `%rbp`. L'instruction `call` est maintenant limitée au nom de la fonction, car il n'est plus nécessairement maintenant de se souvenir du nombre d'arguments passés dans des registres. L'instruction `push_param` est maintenant une instruction plus générale `push` et une nouvelle instruction `pop` fait son apparition.

On traduit chaque instruction ERTL en une ou plusieurs instructions LTL, en se servant du coloriage du graphe et de la structure du tableau d'activation, qui est maintenant connue. Une variable `r` apparaissant dans le code ERTL peut être déjà un registre physique, un pseudo-registre réalisé par un registre physique ou un pseudo-registre réalisé par un emplacement de pile. Dans certains cas, la traduction est facile car l'instruction assembleur permet toutes les combinaisons. Par exemple, l'instruction ERTL

$$L_1 : \ \texttt{mov}\ n\ r \to L$$

devient l'instruction LTL

$$L_1 : \ \texttt{mov}\ n\ \mathit{color}(r) \to L$$

que $\mathit{color}(r)$ soit un registre physique, comme dans `mov $42, %rax`, ou un emplacement de pile, comme dans `mov $42, -8(%rbp)`. Dans d'autres cas, en revanche, c'est plus compliqué car toutes les opérandes ne sont pas autorisées par le jeu d'instructions x86-64. Le cas d'un accès à une variable globale, par exemple,

$$L_1 : \ \texttt{mov}\ x\ r \to L$$

pose un problème quand `r` est alloué sur la pile car on ne peut pas écrire[6]

```
movq x, n(%rbp)
```

6. L'assembleur émet l'erreur `too many memory references for 'movq'`.

Il faut donc utiliser un registre intermédiaire. Le problème est qu'on vient justement de réaliser l'allocation de registres. Une solution simple consiste à réserver deux registres particuliers, qui seront utilisés comme registres temporaires pour ces transferts avec la mémoire et ne seront pas utilisés par ailleurs. En pratique, on n'a pas nécessairement le loisir de gâcher ainsi deux registres. On doit alors modifier le graphe d'interférence et relancer une allocation de registres pour déterminer un registre libre pour le transfert. Heureusement, cela converge très rapidement en pratique, en deux ou trois étapes seulement.

Si on opte pour la première solution, par exemple en utilisant `%r10` et `%r11`, on peut alors facilement traduire chaque instruction ERTL. Ainsi, pour traduire l'instruction ERTL

$$L_1 : \ \texttt{mov}\ x\ r \rightarrow L$$

on considère deux cas de figure. Si $color(r)$ est un registre physique hw, on a une instruction LTL

$$L_1 : \ \texttt{mov}\ x\ hw \rightarrow L$$

si en revanche $color(r)$ est un emplacement de pile $n($`%rbp`$)$, alors on a deux instructions LTL

$$L_1 : \ \texttt{mov}\ x\ \texttt{\%r10} \rightarrow L_2$$
$$L_2 : \ \texttt{mov}\ \texttt{\%r10}\ n(\texttt{\%rbp}) \rightarrow L$$

où L_2 est une étiquette fraîche. Il en va de même pour lire le contenu d'une variable. On a parfois besoin des deux temporaires, par exemple dans le cas d'une instruction ERTL

$$L_1 : \ \texttt{store}\ r_1\ n(r_2) \rightarrow L$$

où r_1 et r_2 sont tous les deux alloués sur la pile. Pendant la traduction vers LTL, on applique un traitement spécial dans certains cas. D'une part, l'instruction `mov` $r_1\ r_2 \rightarrow L$ est traduite par `goto` $\rightarrow L$ lorsque r_1 et r_2 ont la même couleur. C'est là que l'on récolte les fruits d'une bonne allocation de registres. D'autre part, l'instruction x86-64 `imul` exige que sa seconde opérande soit un registre. Il faut utiliser un temporaire si ce n'est pas le cas. Enfin, une opération binaire ne peut avoir ses deux opérandes en mémoire. Il faut utiliser un temporaire si ce n'est pas le cas.

On connaît maintenant la taille du tableau d'activation. Si n est le nombre d'arguments de la fonction et m le nombre d'emplacements sur la pile utilisés par l'allocation de registres, le tableau d'activation a la structure ci-contre, où chaque case occupe ici 8 octets. On traduit `alloc_frame` par

	⋮
	param. 7
	⋮
	param. n
	adr. retour
`%rbp` →	ancien `%rbp`
	locale 1
	⋮
`%rsp` →	locale m
	⋮

```
push %rbp
mov %rsp %rbp
add $ − 8m %rsp
```

et `delete_frame` par

```
mov %rbp %rsp
pop %rbp
```

Dans le cas où $m = 0$, on peut simplifier respectivement en un simple `push` et un simple `pop`. Enfin, on peut traduire l'instruction ERTL `get_param` $k\ r$ en terme d'accès par rapport à `%rbp`, c'est-à-dire[7] `mov` $k($`%rbp`$)$ $color(r)$.

7. Si $color(r)$ est un emplacement de pile, il faut utiliser un temporaire.

instr	::=	`mov` $n\ o \rightarrow L$	chargement d'une constante
		`mov` $x\ r \rightarrow L$	lecture d'une variable globale
		`mov` $r\ x \rightarrow L$	écriture d'une variable globale
		`load` $n(r)\ r \rightarrow L$	lecture en mémoire
		`store` $r\ n(r) \rightarrow L$	écriture en mémoire
		$\mathit{unop}\ o \rightarrow L$	opération unaire
		$\mathit{binop}\ o\ o \rightarrow L$	opération binaire
		$\mathit{ubranch}\ r \rightarrow L, L$	branchement unaire
		$\mathit{bbranch}\ r\ r \rightarrow L, L$	branchement binaire
		`goto` $\rightarrow L$	branchement inconditionnel
		`call` $x \rightarrow L$	appel de fonction
		`return`	instruction explicite de retour
		`push` $o \rightarrow L$	empiler une valeur
		`pop` $r \rightarrow L$	dépiler une valeur

FIGURE 9.10 – Langage LTL.

```
fact()
  entry : L17
  L17: push %rbp           -> L24
  L24: mov %rsp %rbp       -> L23
  L23: add $-8 %rsp        -> L16
  L16: mov %rbx -8(%rbp)   -> L15
  L15: goto                -> L14
  L14: mov %rdi %rbx       -> L10
  L10: goto                -> L9
  L9 : jle $1 %rbx     -> L8, L7
  L8 : mov $1 %rax         -> L1
  L1 : goto                -> L22
  L22: goto                -> L21
  L21: mov -8(%rbp) %rbx   -> L20
```

```
  L20: goto                -> L19
  L19: mov %rbp %rsp       -> L25
  L25: pop %rbp            -> L18
  L18: return
  L7 : mov %rbx %rdi       -> L6
  L6 : add $-1 %rdi        -> L5
  L5 : goto                -> L13
  L13: goto                -> L12
  L12: call fact           -> L11
  L11: goto                -> L4
  L4 : goto                -> L3
  L3 : goto                -> L2
  L2 : imul %rbx %rax      -> L1
```

FIGURE 9.11 – Code LTL pour la fonction `fact`.

Pour la fonction `fact`, on obtient au final le code LTL donné figure 9.11. Ce code contient de nombreuses instructions `goto` qui vont disparaître pendant la phase suivante.

9.5 Production de code assembleur x86-64

Il nous reste une dernière étape. Notre code est toujours sous la forme d'un *graphe de flot de contrôle* et l'objectif est de produire du *code assembleur linéaire*. Plus précisément, les instructions LTL de branchement conditionnel contiennent deux étiquettes, une étiquette en cas de test positif et une autre en cas de test négatif, alors que les instructions de branchement conditionnel de l'assembleur contiennent une unique étiquette pour le cas positif et poursuivent l'exécution sur l'instruction suivante en cas de test négatif. Cette dernière transformation que nous devons effectuer s'appelle la *linéarisation*.

Pour réaliser cette transformation, on parcourt le graphe de flot de contrôle et on produit le code x86-64 tout en notant dans une table les étiquettes déjà visitées. Lors d'un branchement, on s'efforce autant que possible de produire le code assembleur naturel si la partie du code correspondant à un test négatif n'a pas encore été visitée. Dans le pire des cas, on utilise un branchement inconditionnel (`jmp`). On utilise aussi une seconde table pour noter les étiquettes qui devront apparaître au final dans le code assembleur, comme destination d'instructions de saut. En effet, on ne souhaite pas produire du code assembleur où chaque instruction est précédée d'une étiquette. Ce serait correct mais illisible[8].

La linéarisation peut être réalisée par deux fonctions mutuellement récursives. Une première fonction,`lin`, produit le code à partir d'une étiquette donnée, s'il n'a pas déjà été produit, et une instruction de saut vers cette étiquette sinon. La second fonction, `instr`, traduit l'instruction correspondant à une étiquette donnée, sans condition, et rappelle `lin` pour la suite de la linéarisation.

```
lin(L) =
  si L n'a pas été visitée
  alors
    marquer L comme visitée
    instr(L)
  sinon
    marque l'étiquette L comme nécessaire
    produire l'instruction "jmp L"
```

La fonction `instr` traduit chaque instruction LTL en une ou plusieurs instructions x86-64. Dans certains cas, c'est du mot à mot, comme par exemple pour le chargement d'une constante ou la lecture d'une variable globale :

$$\begin{array}{rl} \texttt{instr}(L_1 : \texttt{mov}\ n\ d \to L) = & \text{produire } L_1 : \texttt{mov}\ n\ d \\ & \text{appeller } \texttt{lin}(L) \\ \texttt{instr}(L_1 : \texttt{mov}\ x\ r \to L) = & \text{produire } L_1 : \texttt{mov}\ x\ r \\ & \text{appeller } \texttt{lin}(L) \end{array}$$

Le cas intéressant est celui d'un branchement. On considère d'abord le cas favorable où le code correspondant à un test négatif n'a pas encore été produit. Il peut donc être placé

8. Plutôt que de noter les étiquettes destinations de sauts dans une table au fur et à mesure de la linéarisation, on pourrait aussi calculer cet ensemble d'étiquettes au préalable.

immédiatement après le branchement conditionnel[9].

$$\texttt{instr}(L_1 : \texttt{branch}\ cc \to L_2, L_3) = \begin{array}{l} \text{produire } L_1 : \texttt{j}cc\ L_2 \\ \text{appeller } \texttt{lin}(L_3) \\ \text{appeller } \texttt{lin}(L_2) \end{array}$$

Sinon, il est possible que le code correspondant au test positif (L_2) n'ait pas encore été produit et on peut alors avantageusement *inverser la condition* de branchement.

$$\texttt{instr}(L_1 : \texttt{branch}\ cc \to L_2, L_3) = \begin{array}{l} \text{produire } L_1 : \texttt{j}\overline{cc}\ L_3 \\ \text{appeller } \texttt{lin}(L_2) \\ \text{appeller } \texttt{lin}(L_3) \end{array}$$

où la condition $\overline{cc}$ est l'inverse de la condition cc. Enfin, dans le cas où le code correspondant aux deux branches a déjà été produit, on n'a pas d'autre choix que de produire un branchement inconditionnel.

$$\texttt{instr}(L_1 : \texttt{branch}\ cc \to L_2, L_3) = \begin{array}{l} \text{produire } L_1 : \texttt{j}cc\ L_2 \\ \text{produire } \texttt{jmp}\ L_3 \end{array}$$

On peut essayer d'estimer la condition qui sera vraie le plus souvent pour que le branchement soit effectif le moins souvent. S'il s'agit d'un test de boucle, par exemple, on peut considérer qu'il est plus souvent vrai, car on n'écrit rarement des boucles qu'au plus une itération.

Comme on l'a vu, le code LTL contient de nombreux `goto`, d'origines diverses — boucles `while` dans la phase RTL, insertion de code dans la phase ERTL, suppression d'instructions `mov` dans la phase LTL. On s'efforce de les éliminer lorsque c'est possible.

$$\begin{array}{rll} \texttt{instr}(L_1 : \texttt{goto} \to L_2) = & \text{produire } \texttt{jmp}\ L_2 & \text{si } L_2 \text{ a déjà été visitée} \\ = & \text{produire l'étiquette } L_1 & \\ & \text{appeller } \texttt{lin}(L_2) & \text{sinon} \end{array}$$

Au final, on obtient pour la fonction `fact` le code x86-64 donné figure 9.12. Ce code n'est pas optimal, mais néanmoins de l'ordre de ce que donne `gcc -O1`. Il est bien meilleur que celui donné par `gcc -O0` ou `clang -O0`. Mais il est moins bon que celui donné par `gcc -O2` ou `clang -O1`, où la fonction `fact` est transformée en une boucle. Sans parler d'une transformation aussi radicale, on peut noter plusieurs points sur lesquels notre code assembleur n'est pas parfait. En premier lieu, le tableau d'activation est créé et `%rbx` y est sauvegardé avant de tester si `x <= 1`. Lorsque c'est le cas, on aurait pu s'épargner ce travail. Ceci est dû à notre traduction vers ERTL, qui sauvegarde systématiquement les registres *callee-saved* en entrée de fonction. Par ailleurs, lorsque `x <= 1`, on saute en `L8`, pour mettre 1 dans `%rax`, puis on fait un saut inconditionnel en `L1` pour terminer la fonction. On pourrait s'épargner ce saut supplémentaire, soit en dupliquant la fin de fonction, soit en mettant 1 dans `%rax` dès le départ, pour sauter alors directement en `L1`. On note également que l'instruction `movq %rbx, %rdi` est inutile, car ces deux registres contiennent déjà la même valeur. En effet, on a copié `%rdi` dans `%rbx` trois instructions

9. Il n'est pas nécessaire d'appeler $\texttt{lin}(L_2)$ ensuite si ce code a déjà été produit, mais ce n'est pas incorrect. Cela va juste produire une instruction `jmp` inutile et inatteignable.

plus haut. Ici, une analyse de flot de données assez simple pourrait déterminer que les deux registres contiennent la même valeur et supprimer cette instruction. Enfin, on aurait pu utiliser l'instruction `decq %rdi` plutôt que `addq $-1, %rdi`, par une meilleure sélection d'instruction, de même qu'on aurait pu utiliser `push` et `pop` pour sauvegarder et restaurer `%rbx`.

Mais il est toujours plus facile d'optimiser *un* programme à la main.

Notes bibliographiques. L'allocation de registres de type *Iterated Register Coalescing* est due à George et Appel [10], sur la base de travaux antérieurs de Chaitin.

```
fact:   pushq %rbp
        movq  %rsp, %rbp
        addq  $-8, %rsp
        movq  %rbx, -8(%rbp)
        movq  %rdi, %rbx
        cmpq  $1, %rbx
        jle   L8
        movq  %rbx, %rdi
        addq  $-1, %rdi
        call  fact
        imulq %rbx, %rax
L1:
        movq  -8(%rbp), %rbx
        movq %rbp, %rsp
        popq %rbp
        ret
L8:
        movq  $1, %rax
        jmp   L1
```

FIGURE 9.12 – Code x86-64 pour la fonction `fact`.

Annexes

Solutions des exercices

Exercice 1, page 9

Elle met le registre `%rax` à zéro, car quelle que soit la valeur d'un bit b, le ou exclusif de b et b vaut toujours 0. C'est légèrement plus économe que `movq $0, %rax`, car l'instruction est représentée en machine par trois octets au lieu de sept.

Exercice 2, page 9

L'entier -16 s'écrit $111\dots1110000_2$ en complément à deux (des chiffres 1 terminés par quatre chiffres 0). Du coup, l'instruction `andq $-16, %rsp` a pour effet de mettre à 0 les quatre chiffres de poids faible de `%rsp`. On peut l'interpréter de façon arithmétique comme le plus grand multiple de 16 inférieur ou égal à `%rsp`.

C'est notamment une façon d'aligner la pile avant une instruction `call`, lorsqu'on ne sait pas si la pile est ou non alignée. Dans ce cas, il faut également ajouter des instructions pour restaurer la pile dans son état initial. La section 1.3 explique comment utiliser le registre `%rbp` pour cela.

Exercice 3, page 13

L'entier `a` représente les colonnes de l'échiquier restant à remplir. Les entiers `b` et `c` représentent les colonnes qui sont en prise avec des reines déjà placées sur les lignes précédentes. L'entier `a & ~b & ~c` représente donc les colonnes qu'il faut considérer pour la ligne courante. On parcourt les bits à 1 de cet entier avec la boucle `while`, en extrayant à chaque fois le bit à 1 le plus faible avec `e & -e`. Pour chaque colonne examinée, on fait un appel récursif avec `a`, `b` et `c` mis à jour en conséquence. On a trouvé une solution lorsqu'on parvient à `a` $= 0$.

Note : On trouvera une justification de l'astuce `e & -e` dans l'excellent livre de Henry Warren *Hacker's Delight* [27]. Ce livre contient par ailleurs beaucoup d'information utile à celui qui écrit un compilateur, comme par exemple une méthode systématique pour remplacer une division par une constante par des opérations moins coûteuses que la division.

Exercice 4, page 15

La valeur de n est dans `%rdi` et n'est pas modifiée. On choisit de placer c dans `%rax` car ce sera la valeur de retour. On choisit de placer s dans un `%rsi`, un registre *caller-saved*. Voici un code possible :

```
isqrt:
        xorq  %rax, %rax
        movq  $1, %rsi
        jmp   2f
1:      incq  %rax
        leaq  1(%rsi, %rax, 2), %rsi
2:      cmpq  %rdi, %rsi
        jle   1b
        ret
```

On utilise ici le fait qu'une étiquette peut être un entier, auquel on peut faire référence en avant ou en arrière. Ainsi, l'étiquette `1b` signifie l'étiquette `1` plus haut dans le code et l'étiquette `2f` signifie l'étiquette `2` plus loin dans le code. On a placé le test de la boucle (étiquette 2) *après* le corps de la boucle (étiquette 1). Initialement, on saute au test avec `jmp 2f`. Avec cette façon de procéder, on n'effectue qu'une seule opération de branchement par tour de boucle.

Pour calculer `isqrt`(17), il suffit d'écrire

```
main:
        movq  $17, %rdi
        call  isqrt
```

Le résultat se trouve alors dans `%rax`. Si on veut l'afficher, on peut utiliser par exemple la fonction de bibliothèque `printf`, comme ceci :

```
        movq  $Sprint, %rdi
        movq  %rax, %rsi
        xorq  %rax, %rax     # pas d'arguments en virgule flottante
        call  printf
Sprint:
        .string "isqrt(17) = %d\n"
```

(La fonction `printf` étant une fonction variadique, on doit indiquer son nombre d'arguments en virgule flottante dans `%rax` ; ici, il n'y en a pas.)

Exercice 5, page 15

On commence par la version réalisée avec une boucle.

```
fact:   movq    $1, %rax        # r <- 1
1:      imulq   %rdi, %rax      # r <- x * r
        decq    %rdi            # x <- x-1
        jg      1b              # on continue si x > 0
        ret
```

On a utilisé ici le fait que l'instruction `decq` a positionné les drapeaux. On peut donc faire un branchement conditionnel immédiatement après.

Pour la version récursive, c'est plus complexe, car il faut sauvegarder l'argument sur la pile.

```
factrec:cmpq    $1, %rdi        # x <= 1 ?
        jle     1f
        pushq   %rdi            # sauve x sur la pile
        decq    %rdi            # x <- x-1
        call    factrec         # appel fact(x-1)
        popq    %rcx            # restaure x
        imulq   %rcx, %rax      # x * fact(x-1)
        ret
1:      movq    $1, %rax
        ret
```

Exercice 6, page 19

```
let succ e =
  Add (e, Cte 1)
```

Exercice 7, page 19

```
let rec eval env = function
  | Cte n         -> n
  | Var x         -> env x
  | Add (e1, e2) -> eval env e1 + eval env e2
  | Mul (e1, e2) -> eval env e1 * eval env e2
```

Exercice 8, page 20

La primitive $+$ doit être appliquée à une paire, tandis que la fonction `fun` $x \to$ `fun` $y \to + (x, y)$ est appliquée successivement à deux arguments. Plus précisément, son application à un argument nous renvoie une fonction, qu'on peut ensuite appliquer à un second argument. En particulier, on peut donc l'appliquer partiellement à un unique argument, à la différence de $+$. On dit d'une fonction qui prend ses arguments successivement, en renvoyant une fonction à chaque fois, qu'elle est *curryfiée*. Le terme vient du nom du mathématicien Haskell Curry.

Exercice 9, page 26

Pour simplifier, on note F le terme `fun` $\mathit{fact} \to$ `fun` $n \to e$ où e est le terme

$$\texttt{if} =(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ \mathit{fact}\ (+(n,\ -1)))$$

La dérivation a alors la forme suivante :

$$\cfrac{\cfrac{\mathit{opfix} \twoheadrightarrow \mathit{opfix} \quad F \twoheadrightarrow (\texttt{fun}\ \mathit{fact} \to \ldots) \quad \ldots}{(\mathit{opfix}\ F) \twoheadrightarrow (\texttt{fun}\ n \to e')}\,(\text{FIX}) \quad 2 \twoheadrightarrow 2 \quad \cfrac{\vdots}{e'[n \leftarrow 2] \twoheadrightarrow 2}}{(\mathit{opfix}\ F)\ 2}\,(\text{APP})$$

où e' est le terme $e[fact \leftarrow opfix\ F]$. On laisse le lecteur (courageux) compléter cette dérivation.

Exercice 10, page 26

Notons e l'expression (2.3), c'est-à-dire $\Delta\ \Delta$. Supposons qu'il y ait une valeur v pour cette expression. Alors la dérivation est nécessairement de la forme

$$\dfrac{\dfrac{}{\Delta \twoheadrightarrow \Delta} \quad \dfrac{}{\Delta \twoheadrightarrow \Delta} \quad \dfrac{\vdots}{\Delta[x \leftarrow \Delta] \twoheadrightarrow v}}{\Delta\ \Delta \twoheadrightarrow v}(\text{APP})$$

car la règle (APP) est la seule qui s'applique. Or $\Delta[x \leftarrow \Delta] = (x\ x)[x \leftarrow \Delta] = \Delta\ \Delta$. Donc la troisième prémisse est une dérivation de même conclusion. En considérant initialement une dérivation de hauteur minimale pour $e \twoheadrightarrow v$, on obtient une contradiction.

Exercice 11, page 29

Pour simplifier, on note f le terme

$$opfix\ (\texttt{fun}\ fact \rightarrow \texttt{fun}\ n \rightarrow \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ fact\ (+(n,\ -1)))).$$

On a alors la séquence de réductions suivante :

$$\begin{array}{ll}
 & f\ 2 \\
\rightarrow & (\texttt{fun}\ n \rightarrow \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ 2 \\
\rightarrow & \texttt{if}\ {=}(2, 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))) \\
\rightarrow & \texttt{if}\ \mathit{false}\ \texttt{then}\ 1\ \texttt{else}\ \times(2,\ f\ (+(2,\ -1))) \\
\rightarrow & \times(2,\ f\ (+(2,\ -1))) \\
\rightarrow & \times(2,\ (\texttt{fun}\ n \rightarrow \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ (+(2,\ -1))) \\
\rightarrow & \times(2,\ (\texttt{fun}\ n \rightarrow \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ 1)) \\
\rightarrow & \times(2,\ \texttt{if}\ {=}(1, 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(1,\ f\ (+(1,\ -1)))) \\
\rightarrow & \times(2,\ \texttt{if}\ \mathit{false}\ \texttt{then}\ 1\ \texttt{else}\ \times(1,\ f\ (+(1,\ -1)))) \\
\rightarrow & \times(2,\ \times(1,\ f\ (+(1,\ -1)))) \\
\rightarrow & \times(2,\ \times(1,\ (\texttt{fun}\ n \rightarrow \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ (+(1,\ -1)))) \\
\rightarrow & \times(2,\ \times(1,\ (\texttt{fun}\ n \rightarrow \texttt{if}\ {=}(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ f\ (+(n,\ -1))))\ 0)) \\
\rightarrow & \times(2,\ \times(1,\ \texttt{if}\ {=}(0, 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(0,\ f\ (+(0,\ -1))))) \\
\rightarrow & \times(2,\ \times(1,\ \texttt{if}\ \mathit{true}\ \texttt{then}\ 1\ \texttt{else}\ \times(0,\ f\ (+(0,\ -1))))) \\
\rightarrow & \times(2,\ \times(1,\ 1)) \\
\rightarrow & \times(2,\ 1) \\
\rightarrow & 2
\end{array}$$

Exercice 12, page 35

Pour *opif*, il faut soit ajouter des constantes booléennes dans notre syntaxe abstraite, soit décider par exemple que *opif* teste l'égalité à zéro. Dans ce second cas, on peut écrire

```
| Op "opif" ->
    let (Paire (Const n1, Paire (Fun (_,bt), Fun (_,be)))) = eval e2 in
    eval (if n1 = 0 then bt else be)
```

Pour *opfix*, on suit la règle de sémantique :

```
| Op "opfix" ->
    let Fun (f, e) as b = eval e2 in
    eval (subst e f (App (Op "opfix", b)))
```

Noter l'usage de `as` pour éviter de reconstruire `Fun (f, e)`.

Exercice 13, page 37

En premier lieu, il faudrait distinguer les noms de toutes les variables, afin qu'un programme comme

```
let x = 1 in
let f = fun y -> x+y in
let x = 2 in
f 0
```

n'écrase pas la valeur `x` = 1 sauvegardée dans la fermeture de `f` par la valeur `x` = 2. Il suffirait ici de renommer la seconde variable. Mais cela ne suffit pas. En effet, dans un programme comme

```
let f = fun x -> fun y -> x+y in
let f1 = f 1 in
let f2 = f 2 in
f1 0 + f2 0
```

on a deux fermetures représentant les valeurs de `f1` et `f2`, la première sauvegardant la valeur `x` = 1 et la seconde la valeur `x` = 2. Si l'environnement stocké dans la fermeture était une structure mutable, la construction de la seconde fermeture aurait un effet de bord sur la première, conduisant à une valeur incorrecte.

Bien sûr, il suffirait de *copier* l'environnement au moment de la construction de la fermeture. Mais ce serait justement le symptôme d'un mauvais choix de structure pour l'environnement. Utiliser directement une structure purement applicative est plus simple. L'ajout et l'accès coûtent un peu plus cher ($O(\log n)$ au lieu de $O(1)$) mais la copie est évitée.

Exercice 14, page 38

Les règles pour la sémantique à grands pas sont les suivantes :

$$\frac{}{E, \texttt{skip} \twoheadrightarrow E} \qquad \frac{E, s_1 \twoheadrightarrow E_1 \quad E_1, s_2 \twoheadrightarrow E_2}{E, s_1; s_2 \twoheadrightarrow E_2}$$

$$\frac{E, e \twoheadrightarrow v}{E, x \leftarrow e \twoheadrightarrow E\{x \mapsto v\}}$$

$$\frac{E, e \twoheadrightarrow \texttt{true} \quad E, s_1 \twoheadrightarrow E_1}{E, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_1} \qquad \frac{E, e \twoheadrightarrow \texttt{false} \quad E, s_2 \twoheadrightarrow E_2}{E, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{true} \quad E, s \twoheadrightarrow E_1 \quad E_1, \texttt{while}\ e\ \texttt{do}\ s \twoheadrightarrow E_2}{E, \texttt{while}\ e\ \texttt{do}\ s \twoheadrightarrow E_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{while}\ e\ \texttt{do}\ s \twoheadrightarrow E}$$

Exercice 15, page 47

Une solution simple consiste à réunir trois expressions régulières, pour les langages des mots contenant respectivement zéro, un et deux caractères b :

$$a\star \mid a\star b\, a\star \mid a\star b\, a\star b\, a\star$$

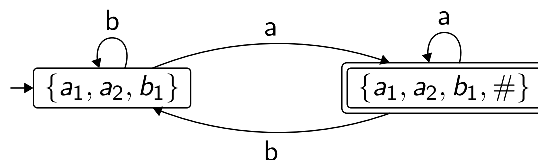
Exercice 16, page 49

La seule différence avec *first* se situe dans la concaténation.

$$\begin{array}{rcl}
\mathit{last}(\emptyset) & = & \emptyset \\
\mathit{last}(\epsilon) & = & \emptyset \\
\mathit{last}(a) & = & \{a\} \\
\mathit{last}(r_1\, r_2) & = & \mathit{last}(r_1) \cup \mathit{last}(r_2) \quad \text{si } \mathit{null}(r_2) \\
 & = & \mathit{last}(r_2) \quad \text{sinon} \\
\mathit{last}(r_1 \mid r_2) & = & \mathit{last}(r_1) \cup \mathit{last}(r_2) \\
\mathit{last}(r\star) & = & \mathit{last}(r)
\end{array}$$

Exercice 17, page 50

On distingue les caractères de la façon suivante : $(a_1|b_1) \star a_2$. On obtient alors l'automate :

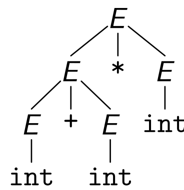


À la différence de l'automate page 47, il est déterministe, c'est-à-dire que pour chaque état et chaque caractère, il n'y a qu'une seule transition possible.

Exercice 18, page 63

On montre la double inclusion des langages reconnus par les grammaires (4.1) et (4.2). Un sens est facile : tout mot reconnu par la grammaire (4.2) est reconnu par la grammaire (4.1). En effet, étant donné un arbre de dérivation, il suffit d'y remplacer $E \to E{+}T$ par $E \to E{+}E$ et $T \to T{+}F$ par $E \to E{*}E$ et d'y supprimer les nœuds $E \to T$ et $T \to F$. On peut le faire rigoureusement par récurrence sur la taille de l'arbre de dérivation.

L'autre sens est plus délicat. En effet, un arbre de dérivation tel que



doit être transformé de façon à faire apparaître la règle $E \to E$+T à sa racine, ce qui est un changement plus subtile. Notons $L(E)$, $L(T)$ et $L(F)$ les langages reconnus par les trois non terminaux de la grammaire (4.2). On a clairement $L(F) \subseteq L(T) \subseteq L(E)$.

On commence par montrer que si $w_1, w_2 \in L(T)$ alors w_1*$w_2 \in L(T)$, par récurrence sur la taille de la dérivation de w_2. Si la dérivation commence par $T \to F$, c'est-à-dire $w_2 \in L(F)$, c'est évident. Sinon, la dérivation commence par $T \to T$*F, c'est-à-dire $w_2 \to^\star w_2'$*f, et on peut appliquer l'hypothèse de récurrence w_1*w_2' puis conclure avec $T \to T$*F.

De même, on montre que si $w_1 \in L(E)$ et $w_2 \in L(T)$ alors w_1*$w_2 \in L(E)$, par récurrence sur la taille de la dérivation de w_1. Puis on montre que si $w_1, w_2 \in L(E)$ alors w_1*$w_2 \in L(E)$, par récurrence sur la taille de la dérivation de w_2. Enfin, on montre que si $w_1, w_2 \in L(E)$ alors w_1+$w_2 \in L(E)$, par récurrence sur la taille de la dérivation de w_2.

On peut maintenant montrer par récurrence sur la taille de l'arbre de dérivation qu'un mot reconnu par la grammaire (4.1) est reconnu par la grammaire (4.2). Si le mot est de la forme `int`, c'est évident. Si le mot est de la forme `(`E`)`, alors on applique l'hypothèse de récurrence à la dérivation de E et on ajoute $E \to T \to F \to$ `(`E`)`. Si la dérivation est de la forme $E \to E$*E, alors on applique l'hypothèse de récurrence aux deux sous-dérivations puis on utilise le résultat précédent. De même si la dérivation est de la forme $E \to E$+E.

Exercice 19, page 63

Cette grammaire est ambiguë car le mot `lam var var` admet deux arbres de dérivation, correspondant respectivement à (`lam var`) `var` et `lam` (`var var`). On peut reconnaître le même langage avec cette grammaire non ambiguë :

$$\begin{array}{rcl} T & \to & A \\ & | & T\ A \\ A & \to & \texttt{nat} \\ & | & \texttt{lam}\ A \end{array}$$

On peut en faire la preuve par double inclusion, comme dans l'exercice précédent.

Exercice 20, page 65

D'une part, on montre par récurrence sur le nombre d'étapes du calcul du point fixe, on montre que si on obtient $\text{NULL}(X) = \texttt{true}$ alors effectivement $X \to^\star \epsilon$.

D'autre part, on montre par récurrence sur le nombre d'étapes de la dérivation $X \to^\star \epsilon$ qu'on obtient bien $\text{NULL}(X) = \texttt{true}$ par le calcul du point fixe.

Exercice 21, page 66

	S	T
NULL(X)	`false`	`false`

	S	T
FIRST(X)	{`(`, `nat`, `lam`}	{`(`, `nat`, `lam`}

	S	T
FOLLOW(X)	$\{\#\}$	$\{$(,), nat, lam, $\#\}$

Exercice 22, page 66

	S	E	L
NULL(X)	false	false	true

	S	E	L
FIRST(X)	$\{$(, sym$\}$	$\{$(, sym$\}$	$\{$(, sym$\}$

	S	E	L
FOLLOW(X)	$\{\#\}$	$\{$(,), sym, $\#\}$	$\{$)$\}$

Exercice 24, page 68

On a calculé NULL, FIRST et FOLLOW pour cette grammaire dans l'exercice 21. On en déduit la table d'expansion suivante :

	nat	lam	(	)	#
S	$T\#$	$T\#$	$T\#$		
T	nat	lam T	(TT)		

Cette grammaire est donc LL(1).

Exercice 25, page 68

On a calculé NULL, FIRST et FOLLOW pour cette grammaire dans l'exercice 22. On en déduit la table d'expansion suivante :

	sym	(	)	#
S	$E\#$	$E\#$		
E	sym	(L)		
L	EL	EL	ϵ	

La grammaire de LISP est donc LL(1).

Exercice 26, page 73

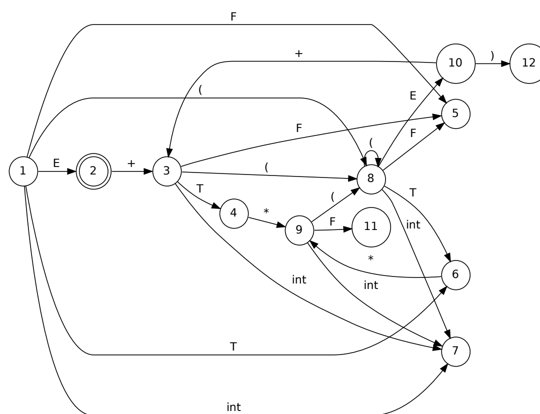
On obtient un automate avec 9 états et les tables suivantes :

	action					*action*
état	`nat`	`lam`	`(`	`)`	#	T
0	`shift` 2	`shift` 4	`shift` 3			1
1					succès	
2	`reduce` $T \to$ `nat`					
3	`shift` 2	`shift` 4	`shift` 3			6
4	`shift` 2	`shift` 4	`shift` 3			5
5	`reduce` $T \to$ `lam` T					
6	`shift` 2	`shift` 4	`shift` 3			7
7				`shift` 8		
8	`reduce` $T \to$ `(` $T\,T$ `)`					

Cette grammaire est donc LR(0).

Exercice 27, page 73

L'automate est le suivant :



Exercice 28, page 73

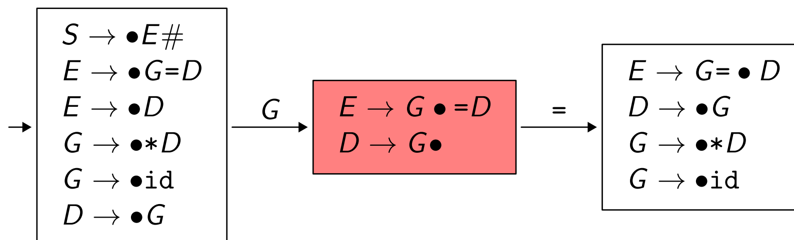
On obtient un automate avec 8 états et les tables suivantes :

	action				*action*	
état	`sym`	`(`	`)`	#	E	L
0	`shift` 2	`shift` 3			1	
1				succès		
2	`reduce` $E \to$ `sym`					
3	`shift` 2	`shift` 3	`reduce` $L \to \epsilon$		6	4
4			`shift` 5			
5	`reduce` $E \to$ `(` L `)`					
6	`shift` 2	`shift` 3	`reduce` $L \to \epsilon$		6	7
7			`reduce` $L \to EL$			

Cette grammaire est donc SLR(1).

Exercice 29, page 74

L'automate contient en particulier les trois états suivants :



Dans le second (en rouge), il y a un conflit lecture/réduction à la lecture du caractère =.

	=	
1	...	...
2	shift 3 reduce $D \to G$	...
3	⋮	⋱

Le caractère = fait bien partie des suivants de G et donc la grammaire n'est pas SLR(1).

Exercice 30, page 74

La réduction $D \to G$ n'est maintenant effectuée que pour le caractère # et le conflit disparaît :

	#	=	
1	...	...	...
2	reduce $D \to G$	shift 3	...
3	⋮	⋮	⋱

Exercice 31, page 77

Le conflit correspond à une entrée de la forme

`IF CONST THEN IF CONST THEN CONST ELSE CONST`

au moment de la lecture du `ElSE`. En effet, on peut alors réduire `IF CONST THEN CONST`, ce qui correspondrait à

`IF CONST THEN (IF CONST THEN CONST) ELSE CONST`

au bien au contraire lire `ELSE`, ce qui correspondrait à

`IF CONST THEN (IF CONST THEN CONST ELSE CONST)`

Ce conflit présent dans la grammaire de nombreux langages de programmation porte le nom en anglais de *dangling else*, qu'on pourrait traduire par « le `else` qui pendouille ». On le résout en général en faveur de la lecture, pour associer le `ELSE` au `THEN` le plus proche. Comme la priorité de la règle

$$E \to \texttt{IF}\ E\ \texttt{THEN}\ E$$

est celle de `THEN`, il suffit de donner à `ELSE` une priorité plus forte que celle de `THEN`, c'est-à-dire

```
%nonassoc THEN
%nonassoc ELSE
```

Exercice 32, page 81

Il suffit de donner à la première occurrence de `fun` $x \to x$ le type

$$(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$$

et à la seconde le type

$$(\texttt{int} \to \texttt{int}).$$

Exercice 33, page 82

$$\frac{\dfrac{\dfrac{\vdots}{\cdots\vdash =(n,0):\texttt{bool}} \quad \dfrac{}{\cdots\vdash 1:\texttt{int}} \quad \dfrac{\dfrac{}{\cdots\vdash \times:\texttt{int}\times\texttt{int}\to\texttt{int}} \quad \dfrac{}{\cdots\vdash n:\texttt{int}} \quad \dfrac{\vdots}{\cdots\vdash \mathit{fact}\ \ (+(n,-1)):\texttt{int}}}{\cdots\vdash \times(n,\mathit{fact}\ \ (+(n,-1))):\texttt{int}}}{\mathit{fact}:\texttt{int}\to\texttt{int}, n:\texttt{int}\vdash \texttt{if}\ =(n,0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\mathit{fact}\ \ (+(n,-1))):\texttt{int}} \qquad \dfrac{\dfrac{}{\cdots\vdash \mathit{fact}:\texttt{int}\to\texttt{int}} \quad \dfrac{}{\cdots\vdash 2:\texttt{int}}}{\mathit{fact}:\texttt{int}\to\texttt{int}\vdash \mathit{fact}\ \ 2:\texttt{int}}}{\vdash \texttt{let rec}\ \mathit{fact}\ n\ =\ \texttt{if}\ =(n,\ 0)\ \texttt{then}\ 1\ \texttt{else}\ \times(n,\ \mathit{fact}\ (+(n,\ -1)))\ \texttt{in}\ \mathit{fact}\ 2 : \texttt{int}}$$

Exercice 34, page 82

On se donne les types suivants pour les types et les expressions de Mini-ML.

```
type typ =
  | Tint
  | Tarrow   of typ * typ
  | Tproduct of typ * typ
type expression =
  | Var   of string
  | Const of int
  | Op    of string
  | Fun   of string * typ * expression
  | App   of expression * expression
  | Pair  of expression * expression
  | Let   of string * expression * expression
```

On peut facilement réaliser l'environnement Γ avec le module `Map` d'OCaml

```
module Smap = Map.Make(String)
type env = typ Smap.t
```

Il s'agit là d'une structure persistante, ce qui nous sera utile plus loin. Par ailleurs, il s'agit d'arbres équilibrés et on a donc une insertion et une recherche en $O(\log n)$ dans un environnement contenant n entrées, ce qui est tout à fait acceptable. Écrivons maintenant la fonction qui calcule le type d'une expression. Certains cas sont immédiats :

```
let rec type_expr env = function
  | Const _ -> Tint
  | Var x -> Smap.find x env
  | Op "+" -> Tarrow (Tproduct (Tint, Tint), Tint)
  | Pair (e1, e2) -> Tproduct (type_expr env e1, type_expr env e2)
```

Pour la fonction, le type de la variable est donné :

```
  | Fun (x, ty, e) ->
      Tarrow (ty, type_expr (Smap.add x ty env) e)
```

Pour la variable locale, il est calculé :

```
  | Let (x, e1, e2) ->
      type_expr (Smap.add x (type_expr env e1) env) e2
```

On note ici l'intérêt de la persistance de `env` : on ajoute dans `env`, en obtenant une nouvelle structure, et il n'est jamais nécessaire de retirer quelque chose de `env`. Enfin, les seules vérifications se trouvent dans l'application :

```
  | App (e1, e2) -> begin match type_expr env e1 with
      | Tarrow (ty2, ty) ->
          if type_expr env e2 = ty2 then ty
          else failwith "erreur : argument de mauvais type"
      | _ ->
          failwith "erreur : fonction attendue"
    end
```

En pratique, on fait un effort pour offrir de meilleurs messages d'erreurs, plus précis (par exemple en affichant les types trouvés et attendus) et localisés (quand les arbres de syntaxe abstraite le sont).

Exercice 35, page 87

On peut donner à $\texttt{fun}\ x \to x\ x$ le type $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$, comme ceci :

$$\dfrac{\dfrac{\dfrac{}{x : \forall\alpha.\alpha \to \alpha \vdash x : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)} \qquad \dfrac{}{x : \forall\alpha.\alpha \to \alpha \vdash x : \forall\alpha.\alpha \to \alpha}}{x : \forall\alpha.\alpha \to \alpha \vdash x\ x : \forall\alpha.\alpha \to \alpha}}{\emptyset \vdash \texttt{fun}\ x \to x\ x : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)}$$

Exercice 36, page 94

Le type de `map_id` n'est pas généralisé, car il s'agit d'une application.

```
val map_id : '_a list -> '_a list = <fun>
```

Du coup, la première application de `map_id` résout `'_a` comme étant `int` et la seconde application échoue. Une solution consiste à écrire `map_id` plutôt sous la forme

```
let map_id l = List.map (fun x -> x) l
```

(On parle d'η-expansion quand on écrit `fun x -> f x` plutôt que `f`.) Du coup, le type de `map_id` est maintenant généralisé car il s'agit d'une valeur, en l'occurrence une abstraction.

Exercice 37, page 114

La présence de procédures mutuellement récursives pose une petite difficulté au typage, car il faut connaître le profil d'une procédure pour typer ses appels au sein d'une autre procédure. Le langage Pascal (le vrai) fournit une syntaxe pour cela, avec le mot-clé `forward`. On peut ainsi écrire

```
procedure p(x: integer); forward;    (* déclare p *)
procedure q(y: integer, z: integer); (* déclare et définit q *)
  begin ... appelle p ou q ... end;
procedure p(x: integer);            (* définit p *)
  begin ... appelle p ou q ... end;
```

Pour ce qui est de la compilation, en revanche, il n'y a rien à changer.

Exercice 38, page 123

Il suffit de construire des fermetures soi-même, à la main. On peut même le faire avec un type ad hoc

```
enum kind { Kid, Kleft, Kright };

struct Kont {
  enum kind kind;
  union { struct Node *r; int hl; };
  struct Kont *kont;
};
```

et une fonction pour l'appliquer

```
int apply(struct Kont *k, int v) { ... }
```

Cela s'appelle la *défonctionnalisation* (Reynolds, 1972).

Exercice 39, page 143

L'appel de méthode `g1.move(10, 5)` est compilé ainsi :

```
        movq %r12, %rdi       # g1.move(10, 5)
        movq $10, %rsi
        movq $5, %rdx
        movq (%rdi), %rcx
        call *8(%rcx)
```

De même pour `g1.draw()`.

```
        movq %r12, %rdi       # g1.draw()
        movq (%rdi), %rcx
        call *16(%rcx)
```

On compile `new Circle` comme on l'a fait pour `new Rectangle`, en supposant ici `g2` allouée dans `%r13`.

```
        movq $48, %rdi # %r13 = g2 = new Circle(10,10,2)
        call malloc
        movq %rax, %r13
        movq $descr_Circle, (%r13)
        movq %r13, %rdi
        movq $10, %rsi
        movq $10, %rdx
        movq $2, %rcx
        call new_Circle
```

Le reste ne pose pas de difficulté supplémentaire.

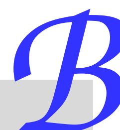
```
        movq $16, %rdi       # %r14 = g3 = new Group()
        call malloc
        movq %rax, %r14
        movq $descr_Group, (%r14)
        movq %r14, %rdi
        call new_Group

        movq %r14, %rdi       # g3.add(g1)
        movq %r12, %rsi
        movq (%rdi), %rcx
        call *24(%rcx)

        ...
```

Exercice 40, page 145

Compilons la construction e `instanceof` C en supposant la valeur de e dans `%rdi` et le descripteur de C dans `%rsi`. On écrit une boucle qui remonte la hiérarchie de classes. On sait qu'on a atteint `Object` lorsque le descripteur de la super classe est nul.

```
instanceof:
        movq    (%rdi), %rdi
L:      cmpq    %rdi, %rsi      # même descripteur ?
        je      Ltrue
        movq    (%rdi), %rdi    # on passe à la super classe
        testq   %rdi, %rdi
        jnz     L               # on a atteint Object ?
Lfalse: movq    $0, %rax
        ret
Ltrue:  movq    $1, %rax
        ret
```

On a pris soin de tester l'égalité avant de tester si la super classe existe, afin que *e* `instanceof Object` renvoie bien `true`.



Petit lexique français-anglais de la compilation

français	anglais	voir pages
affectation	assignment	
appel	call	98
par valeur	by value	
par référence	by reference	
par nom	by name	
par nécessité	by need	
appel terminal	tail call	120
appelant	caller	11
appelé	callee	11
assembleur (langage)	assembly language	5
assembleur (outil)	assembler	5
compilateur	compiler	
décalage	shift	
drapeau	flag	9
filtrage	pattern-matching	123
grammaire	grammar	
interprète	interpreter	33
langage source	source language	
langage cible	target language	
mémoire	memory	
octet	byte	3
pile d'appels	call stack	10
redéfinition	overriding	134
registre	register	
règles d'inférence	inference rules	5
sémantique	semantics	20
dénotationnelle	denotational semantics	
opérationnelle	operational semantics	
à petits pas	small steps semantics	

français	anglais	voir pages
à grands pas	big steps semantics	
sucre syntaxique	syntactic sugar	19
syntaxe abstraite	abstract syntax	18
tas	heap	
valeur gauche	left value	97
variable libre	free variable	21

Bibliographie

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilateurs : principes, techniques et outils*. Pearson, 2007.

[2] A.V. Aho, M.S. Lam, J.D. Ullman, and R. Sethi. *Compilers : Principles, Techniques, and Tools*. Pearson Education, 2011.

[3] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48 :117 – 126, 1986.

[4] Randal E. Bryant and David R. O'Hallaron. *Computer Systems : A Programmer's Perspective*. Pearson, 3rd edition, 2015. http://csapp.cs.cmu.edu/.

[5] Olivier Carton. *Langages formels, Calculabilité et Complexité*. Éditions Vuibert, 2014.

[6] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82 : Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[7] Association Française de Normalisation (AFNOR). *Vocabulaire international de l'informatique*. AFNOR, 1975.

[8] Jean-Christophe Filliâtre. Mesurer la hauteur d'un arbre. In Zaynah Dargaye and Yann Régis-Gianas, editors, *Trente-et-unièmes Journées Francophones des Langages Applicatifs*, Gruissan, France, January 2020. https://www.lri.fr/~filliatr/hauteur/.

[9] Jacques Garrigue. *Relaxing the Value Restriction*, pages 196–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[10] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3) :300–324, May 1996.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580 and 583, October 1969.

[13] G. Kahn, D. Clement, J. Despeyroux, T. Despeyroux, and L. Hascoet. Natural semantics on the computer. Technical Report R.G. 4-85, Greco de Programmation, Université de Bordeaux, June 1985.

[14] Brian Kernighan and Dennis Ritchie. *The C Programming Language (2nd Ed.)*. Prentice-Hall, 1988.

[15] D.-M. Kernighan and B.-W. Ritchie. *Le langage C ANSI (2ème édition)*. Masson, 1990.

[16] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6) :607 – 639, 1965.

[17] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4) :308, 1964.

[18] Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

[19] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4) :363–446, 2009.

[20] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17, May 2007.

[21] Luc Maranget. Compiling pattern matching to good decision trees. In *ML '08 : Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, New York, NY, USA, 2008. ACM.

[22] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[23] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[24] François Pottier. Menhir, a LR(1) parser generator for OCaml. http://gallium.inria.fr/~fpottier/menhir/.

[25] Joseph E. Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[26] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1) :1–34, 1990.

[27] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, 2003.

[28] J.B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1) :111 – 156, 1999.

[29] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115 :38–94, 1992.

Index

Δ, 26
λ-abstraction, 20, 120
λ-calcul, 20, 63, 66, 68, 73
42, 42
91 (fonction), 121

abstraction, 20
`add` (instruction x86-64), 8
adresse, 4
algorithme W, 90
alignement, 12
allocation de registres, 164
alphabet, 46
ambiguë (grammaire), 62
analyse
    ascendante, 69
    descendante, 66
    lexicale, 45
    syntaxique, 55
    sémantique, 79
appel
    par nom, 99
    par nécessité, 99
    par référence, 99
    par valeur, 24, 99
    terminal, 120, 163
appelant, 11
appelé, 11
applicatif (langage), 98
application, 20
arbre
    de dérivation, 61
    de syntaxe abstraite, 18
arithmétique
    des ordinateurs, 3
ARM, 3
arrière
    partie arrière du compilateur, 97
ASCII, 46
assembleur, 5
AT&T, 5, 6, 156
automate
    à pile, 55
    fini, 47
avant
    partie avant du compilateur, 45
axiome, 59

big-endian, 7
bit, 3
    de signe, 4
blanc, 45
boutisme, 7
byte, 3

C, iii, 19, 97, 103, 120, 151
C++, iii, 97, 106, 115, 120, 131, 145
call by name, 99
call by need, 99
call by reference, 99
call by value, 99
callee, 11
callee-saved, 11, 159
caller, 11

caller-saved, 11
capture
  de variable, 21
cast, 144
CISC, 3
classe, 131
coloriage (de graphe), 168
commentaire, 45
  imbriqué, 54
conflit, 71
constructeur, 131
continuation, 122
continuation-passing style, 122
conventions
  d'appel, 11
CPS, 122
`cqto` (instruction x86-64), 8
Curry, Haskell, 183
curryfication, 183

dangling reference, 105
de Bruijn (indices de), 63
débordement de pile, 122
décidable (problème), 63
désassembleur, 7
diamant, 149
downcast, 144
drapeau, 9

encapsulation, 132
ERTL (langage), 159
étoile, 46
Explicit Register Transfer Language, 159
expression régulière, 46
extension
  de signe, 4
  horizontale, 138
  verticale, 138

F (système), 86
fermeture, 35, 116
Fibonacci, 109
filtrage, 123
FIRST, 63
FOLLOW, 63
fonction
  curryfiée, 183
`Format`, 55

GC, 117
Girard, Jean-Yves, 94
GNU, 5, 6
grammaire, 18, 59
  ambiguë, 62

Haskell, 45
héritage, 133
Hindley-Milner, 87
Hoare, logique de, 40

`idiv` (instruction x86-64), 8
`imul` (instruction x86-64), 8
indice de de Bruijn, 63
inférence (règles d'), 23
interférence (graphe d'), 166
interprète, 33
item, 70

Java, iii, 17, 97, 99, 115, 120, 131
`jnz` (instruction x86-64), 9
jugement
  de typage, 80
`jz` (instruction x86-64), 9

Kahn, Gilles, 55
Kempe, Alfred, 169
Kildall, Gary, 165
Knaster–Tarski (théorème de), 64
Knuth, Donald E., 77

langage, 46
  fonctionnel, 115
  à objets, 131
`lea` (instruction x86-64), 9, 154
left value, 97
Leroy, Xavier, 40
`lexbuf`, 52
lexème, 45
Linux, 5, 6
LISP, 66, 68, 73, 188
little-endian, 7
LL(1), 68
Location Transfer Language, 164
logique
  de Hoare, 40
LR(1), 74
LR(0), 71
LTL (langage), 164

McCarthy, John, 121
`menhir`, 75
méthode, 132
Milner, Robin, 79
Mini-ML, 20, 80
MMU, 10
mot, 46
motif, 123
`mov` (instruction x86-64), 8
`movabsq` (instruction x86-64), 8
`movs` (instruction x86-64), 8
`movz` (instruction x86-64), 8

non terminal (symbole), 59
NULL, 63

objet, 131
objets (langage à), 131
OCaml, iii, 19, 53, 55, 79, 80, 89, 97, 102, 115, 137
`ocamllex`, 51
octet, 3
opfix, 21, 27
opif, 20, 27
overloading, 136
overriding, 134

paramètre
    passage, 97
`parse`, 51
partie
    arrière du compilateur, 97
    avant du compilateur, 45
Pascal, 108
pattern, 123
pile d'appels, 10
    débordement, 122
point fixe, 24, 64
pointeur, 4
polymorphisme, 85
pretty-printer, 56
production, 59
pseudo-registre, 156
Python, 45

redéfinition, 134
reduce/reduce, 71
Register Transfer Language, 156
registre, 4
    allocation de registres, 151
Reynolds, John C., 94
RISC, 3
RTL (langage), 156

`sal` (instruction x86-64), 9
`sar` (instruction x86-64), 9
sélection d'instructions, 151
sémantique, 17
    axiomatique, 40
    dénotationnelle, 40
    opérationnelle, 20
    à grands pas, 22
    à petits pas, 28
shift/reduce, 71
`shortest`, 51
`shr` (instruction x86-64), 9
SLR(1), 73
spill, 169
stack frame, 11
stack overflow, 122
statique, 133
Strachey, Christopher S., 41
`sub` (instruction x86-64), 8
substitution, 22
sucre syntaxique, 19
surcharge, 136, 139
symbole
    non terminal, 59
    terminal, 59
syntax-directed, 82
syntaxe
    abstraite, 17
    dirigé par la, 82
Syracuse
    suite de, 109

tableau d'activation, 11
tas, 10
terminaison, 98
terminal (appel), 120
terminal (symbole), 59
top-down parsing, 66
transtypage, 144
typage
    dynamique, 35
    statique, 79

type
  polymorphe, 85
  principal, 93
  simple, 80

unification, 90
upcast, 144
UTF-8, 46

valeur gauche, 97
variable
  libre, liée, 21
vidage en mémoire, 169

W (algorithme), 90

x86-64, 3, 151
  `add`, 8
  `cqto`, 8
  `idiv`, 8
  `imul`, 8
  `jnz`, 9
  `jz`, 9
  `lea`, 9, 154
  `mov`, 8
  `movabsq`, 8
  `movs`, 8
  `movz`, 8
  `sal`, 9
  `sar`, 9
  `shr`, 9
  `sub`, 8

`yacc`, 75