MÉCANISMES DE LA PROGRAMMATION ORIENTÉE-OBJET

EN JAVA

BENJAMIN WERNER

ÉCOLE POLYTECHNIQUE

INF 371 - 2024

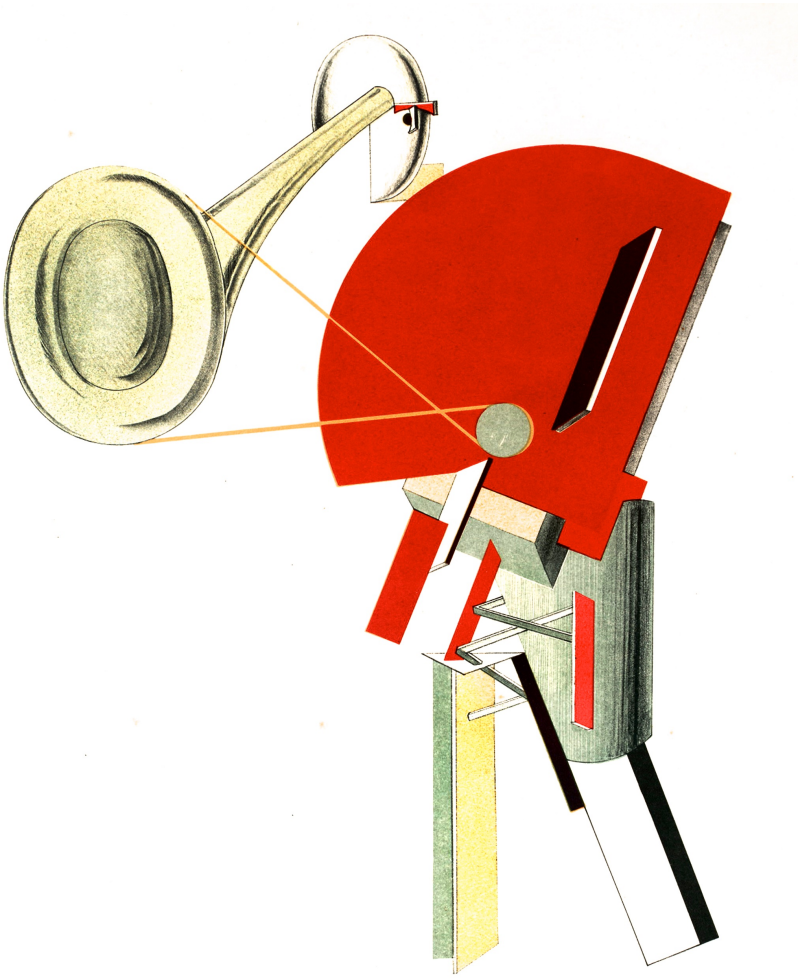


Illustration de couverture : *Annonceur* de El Lissitzky, 1923.

Table des matières

Introduction

Il faut se garder de pousser trop loin l'analogie entre les langages de programmation et les langues humaines dites "naturelles" comme le français, l'anglais, le chinois... Il y a toutefois quelques points communs :

- Dans les deux cas, le langage permet la *communication*. Pour les langues naturelles, c'est entre êtres intelligents ; pour les langages informatiques c'est entre le programmeur et l'ordinateur.
- De plus, avant même que cette communication n'ait lieu, le langage permet à un locuteur humain de *structurer* et d'articuler ce qui peut ensuite être communiqué. C'est la connaissance du langage qui permet à l'humain des pensées et des reflexions complexes. De même, le langage de programmation fournit les outils conceptuels qui permettent de concevoir puis de construire des programmes évolués.

On peut aussi mentionner des différences notables. En particulier, contrairement aux langues naturelles, les langages informatiques sont entièrement artificiels. Ce sont également des langages formels, c'est-à-dire qu'ils obéissent à des règles de grammaire plus précises et plus restrictives que leurs contreparties naturelles. Ils correspondent aussi à des besoins plus spécialisés. L'écriture de logiciels est déjà un champ bien délimité, mais à l'intérieur de celui-ci, en fonction du type de programme voulu, de l'architecture matérielle sur laquelle il est amené à être exécuté, ou de contraintes particulières comme la rapidité d'exécution ou l'importance de la correction, on pourra être amené à préférer un langage ou un autre.

C'est pour ces raisons que de nouveaux langages de programmation sont inventés tous les jours. C'est aussi parce que tout langage de programmation est un compromis que, à l'image des langues naturelles (Eco, 1994), il n'existe pas de langage de programmation parfait.

Le rôle du langage de programmation

Du point de vue matériel, l'ordinateur est une machine capable d'exécuter une suite *d'instructions*. Le jeu d'instructions, c'est-à-dire l'ensemble des instructions d'un ordinateur donné, compose ce qu'on appelle son *langage machine*. Nous donnerons dans ce cours une idée de ce qu'est un langage machine, à travers un ordinateur idéal. S'il est possible de programmer directement en langage machine, cela se révèle vite extrêmement peu pratique.

Tout d'abord parce que, justement, le code machine reflète mal la vision conceptuelle que peut avoir le programmeur. De fait, les premiers langages de programmation sont apparus très rapidement après les premiers ordinateurs, au début des années 1950. Et ce, à une époque où la taille et la complexité des programmes étaient pourtant infiniment plus réduites qu'aujourd'hui.

Le rôle du langage de programmation est donc de servir d'intermédiaire entre l'esprit du programmeur et le langage machine. Plus exactement :

– le programmeur écrit le programme dans le langage de programmation. Cela reste, encore aujourd'hui, presque toujours une suite de caractères dans un ou plusieurs fichiers texte. Ce texte constitue ce qu'on appelle le *code source*.

– Le code source est traduit par un (autre) programme dédié vers le langage machine. Lorsque cette traduction est effectuée une fois pour toute sur l'ensemble du programme, on parle de *compilation*. Le programme qui effectue la traduction est appelé *compilateur*. Le résultat de la traduction est le code compilé ou code *exécutable*. On peut alors exécuter le programme sans plus avoir accès au code source.

Dans certains cas, comme pour le langage Python, le code source est traduit petit à petit au cours de l'exécution. Le code source est alors gardé en mémoire. On parle alors de langage *interprété* et, par exemple, de *l'interpréteur* Python. Dans ce cours, on s'intéresse à un langage compilé, Java, et nous mettons clairement l'accent sur les mécanismes de compilation.

Quelle machine ?

Une fois qu'on s'est placé dans le cadre d'un langage compilé, il y a à nouveau deux approches possibles :

– Le compilateur peut produire un code exécutable écrit dans le langage machine de l'ordinateur, c'est-à-dire pour le processeur matériel sur lequel le programme sera exécuté. On parle alors de compilation native. Cette approche permet d'exploiter au mieux les caractéristiques du processeur. En revanche, ce code exécutable ne sera pas ou peu portable, c'est-à-dire qu'il ne pourra pas être exécuté sur un ordinateur différent. Il faudra donc une version du compilateur pour chaque type d'ordinateur.

– Alternativement, le compilateur peut produire un code exécutable sur une *machine virtuelle*, c'est-à-dire un jeu d'instructions défini préalablement. On parle en général de *bytecode* pour désigner les programmes écrits dans le langage d'une telle machine virtuelle. Dans ce cas, le bytecode devra être lui-même interprété pour faire tourner le programme. Le désavantage principal est que la vitesse d'exécution est inférieure que dans le cas de la compilation native. L'avantage principal est une meilleure portabilité :

  – Si on change d'ordinateur, il n'est plus nécessaire de réécrire le compilateur. Il suffit de porter l'interpréteur de bytecode.

  – De plus, le code compilé est lui aussi portable : il peut être exécuté sur tout ordinateur pour lequel on dispose d'un interpréteur de bytecode.

C'est cette dernière approche qui est utilisée pour Java : les programmes sont compilés pour la *Java Virtual Machine* ou JVM.

Il ne faut toutefois pas exagérer l'importance de ce choix entre machine virtuelle et compilation native. D'une part, les jeux d'instructions de la plupart des machines abstraites ne sont pas si éloignés de ceux de processeurs matériels. D'autre part, les interpréteurs de bytecode sont aujourd'hui très efficaces et réduisent d'autant la différence de vitesse d'exécution [1].

Qu'est-ce-qu'un bon langage de programmation ?

On comprend que tel ou tel langage de programmation permettra de plus ou moins bien exploiter les caractéristiques de l'ordinateur sur lequel les programmes seront exécutés ; en

1. On utilise entre autre la compilation à la volée du bytecode vers le code natif, où le code de la machine abstraite est traduit en code natif peu avant d'être exécuté.

particulier d'obtenir une vitesse de calcul plus ou moins grande. Mais le langage doit aussi être adapté au programmeur.

Plus précisément, c'est parce qu'un programme est un objet extrêmement complexe et que l'écriture d'un programme est une tâche intellectuelle particulièrement ardue, que l'on demande à un langage de programmation un certain nombre de propriétés qui peuvent sembler relever du confort, mais qui permettront, en pratique, d'effectivement réaliser des logiciels de grande taille :

- De permettre au code source de refléter les concepts des algorithmes qu'ils implémentent,
- la possibilité de découper le code en *modules* aussi indépendants que possible, pour permettre à la fois la répartition du travail entre programmeurs, mais aussi de partager des parties de code entre des programmes et donc de créer des bibliothèques logicielles *réutilisables*,
- plus généralement, la modularité permet aussi de renforcer la structure, la clarté, la facilité de maintenance et d'évolution du code,
- on attend enfin d'un langage de programmation d'être lisible et de faciliter la découverte d'erreurs (*bugs*) et leur correction.

Pour saisir combien ces points ont pu être essentiels, on peut considérer la croissance des mémoires des ordinateurs, et par là, de la taille des logiciels. En 1980 un micro-ordinateur disposait au maximum de 64 kilo-octets de mémoire, et quelques méga-octets pour un VAX qui coûtait quelques dizaines de milliers de dollars de l'époque. Aujourd'hui un smartphone à cent euros est équipé de plusieurs giga-octets de mémoire vive. Le développement des langages de programmation a accompagné de manière essentielle celui des capacités des ordinateurs, en terme de vitesse d'exécution et de taille mémoire.

L'idée des langages à objets, ou *orientés-objet*, dont le développement fut particulièrement intense entre 1985 et 1995 correspond à ce double souci de rendre les langages à la fois plus modulaires et plus intuitifs. De fait, ces langages ont contribué à permettre la conception de logiciels de plus en plus complexes. Tout particulièrement, l'orienté-objet est très largement lié au développement des environnements et interfaces graphiques, qui aujourd'hui sont omniprésents dans presque tous les produits informatiques.

À propos de ce cours

Nous l'avons dit, il est essentiel de faire des compromis lors de la conception d'un langage de programmation. Il faut également en faire lorsqu'on choisit le sujet et le périmètre d'un cours. Nous présentons ici le langage Java, comment il fonctionne et comment il est utilisé. On choisit Java pour un certain nombre de caractéristiques :

- C'est un langage compilé,
- il est typé statiquement, c'est-à-dire que tous les objets ont un type, et que ce type est vérifié au moment de la compilation - et pas lors de l'exécution,
- c'est un langage effectivement utilisé,
- il possède la plupart des mécanismes de programmation orientée-objet.

Il s'agit là des critères principaux. La compilation et le typage statique le distinguent par exemple de son lointain cousin JavaScript. Une alternative sérieuse serait, par exemple, C++ qui partage avec Java les caractéristiques ci-dessus. On choisit Java essentiellement car il est moins piégeux grace à une gestion plus automatisée de la mémoire (voir par exemple 5.7).

Mais surtout, ces caractéristiques du langage permettent d'expliquer effectivement son fonctionnement, en montrant comment ses différentes constructions sont traduites en langage machine. Expliciter cette traduction - la compilation - est le choix le plus original de ce cours. Certes, cela oblige le lecteur et l'étudiant à se confronter à un deuxième langage de programmation, heureusement très simple. Mais les avantages sont multiples et très concrets :

- Cela permet de mieux appréhender le fonctionnement réel de l'ordinateur et ouvre la voie à l'étude de l'architecture des ordinateurs.
- Cela rend beaucoup plus lumineux le comportement de nombreuses constructions de Java, en particulier celles relevant de l'orienté-objet ; comportement qui parait souvent artificiel dans des présentations plus superficielles.
- Enfin cela donne une base claire pour programmer effectivement. En particulier le comportement des programmes, leurs complexités en temps ou en espace mémoire peuvent être comprises et justifiées.

En d'autres termes, cela rend le cours peut-être un peu plus abstrait, mais ne le rend pas moins *pratique* pour autant.

Structure de ce document

Ce document de cours est organisé en deux parties :

- La première est la plus structurée, et présente dans l'ordre les caractéristiques principales de Java, puis comment elle peuvent être compilées vers la machine abstraite. Cette partie forme l'épine dorsale du cours.
- La seconde partie présente un certain nombre de structures algorithmiques et comment elles peuvent être implémentées en Java. Ces chapitres servent d'abord de support à certains exercices de programmation en TD, mais aussi à rendre plus concrètes certains principes de programmation orientée-objet. Enfin ils peuvent compléter votre culture algorithmique.

La quasi-totalité de ce document est originale et a été rédigée pour ce cours. Il y a quelques paragraphes qui ont été repris et adapté à partir du poly de l'ancien cours *Algorithmique et Programmation*[2] (Pottier and Werner, 2014) que j'ai donné avec François Pottier entre 2010 et 2014, surtout dans le chapitre sur les graphes.

Avertissement

Il y a déjà eu plusieurs version de ce poly et il a continué à évoluer. Ce document contient certainement encore des erreurs. Je serai reconnaissant pour tout rapport m'en signalant.

Je suis également preneur de toute suggestion d'ordre pédagogique. N'hésitez pas à me signaler des points obscurs, d'autres qui mériteraient plus d'explications, etc. De la promotion 2016, je remercie en particulier Germain Poullot pour une lecture attentive, constructive et acérée ainsi qu'Antoine Carossio. Pour la promotion 2017, merci à Baptiste Moreau, Youssef Salib, Célia Constantini, Louis Hennecart, Grégoire Grzeczkowicz et au moins deux autres élèves dont j'ai oublié de noter les noms. Pour les 2018, merci à Gabriel Flath, et pour les 2019 à (au moins) Élie Aharonian, Florent Allard, Vincent Auffray, Mehdi Chouta, Louis Detzen,

2. Ce cours était numéroté INF431 avant la réforme de l'année 2. Une partie importante de ce poly a depuis été traduite en anglais par Benjamin Doerr pour le (nouveau) cours INF421.

Louis Cousturian, Elliott Donvez, Simon Dreyer, Ali Haidar, Michael Fotso Fotso, Aurélien Legoupil, Meng Yutong, Didier Ngatcha Bakoue, Philippe Nugnès, Théo Pinettes, Thomas Pochart, Josselin Sommerville, Mathéo Vergnolle. Pour les 2020, Alexandre Benoist, Fabien Roger et Aymen Echarghaoui. Pour les 2021, Martin Plazanet et Franck Bambou chez les 2022. Parmi les enseignants, merci à Jean-Marie Madiot, Marc Glisse, Sebastian Will et un grand merci à Pierre-Yves Strub pour les discussions décisives sur la XVM, pour l'émulateur qu'il a écrit pour ce cours, ainsi que pour une grande partie des idées du chapitre 8. Les erreurs restantes sont évidemment de ma responsabilité.

Première partie

Le langage Java

Chapitre 1

Le noyau impératif

Ce chapitre nous permettra d'écrire nos tout premiers programmes Java. En dehors de points de syntaxe, nous n'allons toutefois pas encore étudier des traits propres à Java, mais communs à tout les langages impératifs.

Avant d'être un langage orienté-objet, Java est en effet un langage impératif - alors que OCaml par exemple est un langage essentiellement fonctionnel. Cela signifie que les structures de base des programmes sont des *instructions* qui sont exécutées les unes après les autres, donc de manière *séquentielle*.

Le fragment impératif est souvent considéré comme un langage de programmation "minimal". Il y a plusieurs raisons pour commencer par lui :

- Il fait partie de la très grande majorité des langages de programmation ; même si c'est sous des syntaxes diverses, c'est-à-dire qu'on écrira les choses différemment en Java, OCaml, Python ou C.
- Il est particulièrement concis et simple à définir.
- Cette simplicité fait qu'il est également assez facile de définir sa sémantique ; c'est-à-dire définir précisément et mathématiquement la fonction qui à un programme associe son résultat.
- Il est relativement proche du langage machine, en ce sens qu'on comprend assez facilement le lien entre un programme écrit dans ce fragment, et la manière dont il sera exécuté physiquement par l'ordinateur.
- Il est également suffisamment expressif pour permettre de coder des calculs compliqués. En fait, il permet de coder tous les calculs imaginables (on dit qu'il est *Turing-complet*.)

1.1 Les constructions

Le noyau impératif est donc un *fragment commun* à Java, C, C++... Nous le présentons ici dans la syntaxe Java.

1.1.1 Les expressions

L'ensemble des expressions du noyau impératif est défini par les clauses suivantes :

- Une constante numérique comme 2, 4, 1812 ou 1793 est une expression,
- les constantes booléennes `true` et `false` sont des expressions,
- une variable (`x`, `y`, `z`...) est une expression,
- si `e1` et `e2` sont des expressions, alors on peut les composer par un opérateur numérique (au choix, $+, -, *, /$). C'est-à-dire que `e1 + e2`, `e1 - e2`, `e1 * e2` et `e1 / e2` sont également des expressions.
- De même, on peut composer deux expressions par un opérateur booléen : `e1 && e2` et `e1 || e2` sont également des expressions.
- Enfin on dispose aussi d'opérateurs unaires : `- e1` et `\~ e2` sont des expressions (la seconde désigne la négation booléenne).

1.1.2 L'affectation

Si `x` est une variable, et `e` une expression (comme défini ci-dessus) alors l'affectation de la valeur de `e` à `x` est une instruction. En java c'est écrit : `x = e;`.

Voici, en exemple, une séquence de telles affectations :

```
x = 4 ;
y = 2*(x+2);
x = x + y;
```

1.1.3 La séquence

Un programme ne se compose évidemment pas d'une seule instruction ; il faut pouvoir les enchaîner, comme dans l'exemple précédent. On dispose pour cela de la construction de séquence qui permet de former une instruction à partir de deux instructions.

En Java, la séquence formée des instructions p_1 et p_2 est notée $\{p_1\ p_2\}$. Il s'agit bien sûr du programme qui va exécuter d'abord p_1 puis p_2. S'il n'y a pas d'ambiguïtés on omet les accolades. On remarque qu'on peut considérer l'opérateur de séquence comme associatif : on n'a pas besoin de différencier $\{p_1\ \{p_2\ p_3\}\}$ et $\{\{p_1\ p_2\}\ p_3\}$ qui désignent la même séquence.

1.1.4 Le test

Le test, ou *choix conditionnel*, permet de choisir entre l'exécution de deux instructions. En Java, il est noté `if (e) p1 else p2`.

- Cette construction n'est bien typée que si `e` est une expression de type `boolean`.
- Ce programme va d'abord calculer la valeur de `e`. Si cette valeur s'avère être `true`, le programme va ensuite exécuter `p1` et si la valeur est `false`, le programme va exécuter `p2`.

Lorsque la séquence `p2` est vide, on peut simplement écrire `if (e) p1`.

Solution page 183.

Exercice 1.1.1 Le programme suivant affiche-t-il quelque chose ? Si oui, quoi ?

```
if (false)
  System.out.println("Hello ");
  System.out.println("world");
```

1.1.5 La boucle

La structure de boucle est importante, ne serait-ce parce qu'elle permet la construction de programmes qui ne terminent pas forcément. La syntaxe Java est `while (e) p`.

Comme dans beaucoup de langages de programmation, cette structure spécifie le comportement suivant :

1. on évalue l'expression `e` (qui doit être de type `boolean`),
2. si la valeur obtenue pour `e` est `false` on ne fait rien,
3. si la valeur obtenue pour `e` est `true` on exécute `p` puis on revient à l'étape 1.

Autrement dit, ce programme est équivalent à un "programme" formé par un emboîtement infini de tests :

```
if (e) then {
   p;
   if (e) then {
        p;
        if (e) then {
          p;
          if (e) then {
            p;
            if (e) then { ... }
} } } }
```

Parce que cette instruction correspond virtuellement à une construction infinie, elle permet de construire des programmes qui ne terminent pas. Plus généralement, la boucle est la construction qui donne sa complexité au langage de programmation :

– C'est essentiellement grace à elle que le langage de programmation est ce qu'on appelle Turing-complet, c'est-à-dire qu'il permet d'exprimer tous les algorithmes connus.

– C'est à cause d'elle que le langage acquiert sa complexité logique. Non seulement on peut écrire des programmes qui ne terminent pas, mais le problème de savoir si un programme donné termine ou pas est en général *indécidable*. C'est-à-dire qu'il n'existe pas d'algorithme capable, pour tout programme, de déterminer si ce programme termine [1].

1.1.6 La déclaration de variable

Avant d'utiliser des variables dans des expressions ou des affectations, il est nécessaire de les déclarer. Il y a plusieurs possibilités pour faire cela en Java. Pour l'instant, nous distinguons simplement entre variables globales et locales.

Il est possible de déclarer une variable qui est utilisée localement, dans une partie du programme. Il faut, lors de la déclaration, préciser le type de la variable (on verra qu'en Java tout nouveau nom de variable doit être accompagné de son type). Pour une variable de type nombre entier, on écrira `{ int x; p }`. Dans ce cas, `x` est locale au programme `p` ; on ne peut y faire référence ailleurs.

On peut combiner la déclaration de la variable et une première affectation ; la syntaxe suivante :

```
{  int x = e;
   p
}
```

1. Ces questions sont développées, entre autre, dans le cours INF412 d'Olivier Bournez (Bournez, 2017). On trouve également de nombreux travaux de vulgarisation sur ces sujets comme (Hofstadter, 1979) très en vogue à son époque ou, plus récemment la très bonne bande dessinée *Logicomix* (Doxiadis et al., 2009) — même si elle présente les logiciens sous un angle pas toujours flatteur !

est juste une abréviation de :

```
{  int x;
   x = e;
   p
}
```

On peut également déclarer des variables globales, c'est-à-dire utilisables dans l'ensemble du programme ; on le voit dans la partie suivante.

1.1.7 Types de base de Java

Nous avons déjà mentionné ci-dessus deux types de base disponibles en Java :

- le type `int` des entiers ; plus précisément ce type désigne les entiers de 32 bits, c'est-à-dire compris entre -2^{31} et $2^{31} - 1$;
- le type `boolean` qui ne comporte que deux valeurs, `true` et `false`.

Il existe d'autres types de base, en particulier :

- le type `byte` des entiers 8 bits, compris entre -128 et 127 ;
- le type `short` des entiers 16 bits, compris entre -32.768 et 32.767 ;
- le type `long` des entiers 64 bits ;
- le type `float` des nombres à virgule flottante pouvants être représentés sur 32 bits ;
- le type `double` des nombres à virgule flottante pouvants être représentés sur 64 bits ;
- le type `char` des caractères, plus précisément des caractères Unicode 16 bits.

On remarque qu'il existe des conversions canoniques entre certains types numériques ; par exemple un `int` peut être converti en `long`, un `long` ou un `int` en `float`, un `float` en `double`. Ces fonctions de conversions sont insérées automatiquement par le compilateur, ce qui permet d'utiliser un `int` comme un `double` comme dans le code suivant :

```
int n;
double x;
n = 4;
x = n + 0.5;
```

Ces fonctions de conversions sont parfois appelées coercions ; on parle alors de *coercions implicites*.

1.1.8 À la frontière entre expressions et instructions

Pour être tout à fait précis, on peut faire la remarque que nous donnons ici une présentation un peu restrictive de Java, en imposant une frontière rigoureuse entre expressions et instructions. Dans la pratique, Java permet de considérer certaines instructions comme des expressions, et vice-versa. Il s'agit largement de l'héritage de pratiques anciennes venues de C. Par exemple l'instruction `x = e;` peut également être utilisée comme expression. Elle vaut alors `e`, donc la nouvelle valeur de `x`. Cela permet d'écrire des choses comme `x = y = e;` qui va affecter la valeur de `e` à la fois à `x` et `y`.

Nous n'encourageons pas l'utilisation intensives de telles constructions, au moins dans un premier temps. Nous les décrivons un peu plus dans la partie 3.6.

1.2 Nos premiers programmes

1.2.1 Structure générale

Les programmes Java sont structurés en classes. Pour l'instant, nous utilisons une seule classe, mais remarquons déjà que :

> Les noms des classes Java commencent avec une majuscule.

Les variables globales sont déclarées au début de la classe avec le mot-clé `static`. Voici un exemple de programme :

```
class Factorielle {
    static int n = 5;
    static int i;

    public static void main(String[] arg) {
        i = n - 1 ;
        while (i > 0) {
            n = n * i;
            i = i - 1;
        }
        System.out.println(n);
    }
}
```

Le code principal est regroupé dans la fonction `main` ; c'est celui qui est exécuté lorsqu'on lance le programme. Ce code peut faire appel à d'autres fonctions, dont on décrit la syntaxe dans le chapitre 3. On verra au chapitre 5 ce que signifie le mot-clé `static`.

1.2.2 Le fichier source

Le programme donné ci-dessus est placé dans un fichier. En Java, il doit y avoir concordance entre le nom de la classe et le nom du fichier, il s'appellera donc `Factorielle.java`.

Si l'on utilise un environnement de programmation intégré, comme Eclipse ou netbean, se dernier se chargera de nommer le fichier en fonction du nom de la classe. Si on utilise un éditeur de texte générique comme emacs, il faudra prendre garde de faire concorder nom de la classe et du fichier.

1.2.3 Compilation

Le compilateur Java va analyser le fichier source. Comme la plupart des compilateurs, on peut sommairement décrire son travail comme composé de trois passes :

1. Vérification de la syntaxe, c'est-à-dire vérifier qu'il est bien composé de constructions comme celles que nous venons de décrire (affectation, boucle, test...). Une erreur dans cette passe donne le fameux message "syntax error".
2. Vérification du typage, c'est-à-dire que le programme respecte bien la discipline des types de Java.
3. Génération du code compilé, c'est-à-dire la version exécutable du programme. Chaque fichier source est ainsi traduit en un fichier correspondant, muni du suffixe `.class`. Ici on obtiendra un fichier `Factorielle.class`.

Ici aussi, on peut appeller le compilateur à l'intérieur d'Eclipse ou de netbean. On peut aussi l'appeler de manière plus élémentaire, dans un environnement de type *shell* d'un système d'exploitation unix (comme linux, openBSD ou Mac OS) dans un terminal avec la commande `javac` (pour *Java compiler*).

```
javac Factorielle.java
```

Que contient le fichier `Factorielle.class` ? Le principe de la compilation est de transformer un programme écrit par le programmeur en code exécutable. Comme on l'a indiqué dans l'introduction, ce code est écrit dans le langage de la machine abstraite JVM (*Java Abstract Machine)*. On exécute donc le programme en appelant l'interpréteur de bytecode JVM (commande `java`) sur ce programme :

```
java Factorielle
```

Dans le chapitre suivant, nous montrons comment compiler le noyau impératif de Java vers une machine abstraite.

1.3 Autres constructions simples

1.3.1 Les tableaux

On a indiqué que le noyau impératif présenté ci-dessus (affectation, test, boucle) est "Turing-complet", c'est-à-dire qu'il permet de programmer tous les algorithmes connus. En pratique, il est nettement plus confortable de pouvoir utiliser des tableaux.

En Java, le type des tableaux d'entiers sera noté `int[]`, celui des tableaux de booleens `boolean[]`, etc. . . On peut ainsi déclarer :

```
int[] t;
```

Mais avant d'utiliser le tableau `t` il faut encore fixer sa taille. Ce qu'on fait avec l'instruction suivante :

```
t = new int[10];
```

qui fera de `t` un tableau de taille 10, c'est-à-dire indicé de 0 à 9. Comme dans la plupart des langages, `t[0]`, `t[1]`, . . . `t[9]` sont alors chacune une variable de type `int`. On peut accéder à la longueur du tableau `t` par la syntaxe `t.length` (qui vaudra ici 10).

Il est possible d'initialiser un tableau dès sa déclaration, par exemple par

```
int[] t = new int[10];
```

Pour être tout à fait précis, on verra que formellement `t = new int[10]` est une instruction et pas une déclaration (par exemple on peut changer la longueur d'un tableau dans le programme). Mais il serait prématuré d'entrer plus dans ces détails ici.

1.3.2 L'argument de la fonction `main`

On peut à ce stade indiquer à quoi correspond l'argument de la fonction `main` qui contient le corps du programme :

```
public static void main(String[] args) { ... }
```

Ici on l'appelle `args` mais on pourrait choisir un autre nom. On voit que cet objet `args` est un tableau de chaines de caractères (type `String`).

Quand on appelle le programme avec la commande `java`, on peut passer des arguments supplémentaires au programme. Si on fait :

```
java NomDeLaClasse je suis un argument
```

on va exécuter le code contenu dans la fonction `main` de la classe `NomDeLaClasse` dans lequel `args` sera un tableau dont le premier élément vaudra `"je"`, le deuxième `"suis"`, etc. . .

1.3.3 Variantes de l'affectation

Java propose un certain nombre de syntaxes concises pour les affectations qui consistent à modifier la valeur d'une variable :

`a++;`	est une abréviation de	`a = a + 1;`
`a--;`	—	`a = a - 1;`
`a += 3;`	—	`a = a + 3;`
`a -= 4;`	—	`a = a - 4;`

Ces abréviations[2] ont été reprises du langage C (Kernighan and Ritchie, 1988).

1.3.4 Boucles `for`

Les boucles **`for`** sont une variante des boucles **`while`** présentes dans la quasi-totalité des langages de programmation. Il s'agit s'une boucle munie d'une variable qui sert de compteur.

Le programme suivant affiche les entiers de 1 à 10 :

```
int i;
for (i = 1; i < 11; i++) System.out.println(i);
```

Il est strictement équivalent au programme suivant :

```
int i;
i = 1;
while (i < 11) {
   System.out.println(i);
   i++;
}
```

Le programme suivant affiche uniquement les entiers pairs :

```
int i;
for (i = 2; i < 11; i +=2) System.out.println(i);
```

Remarquons qu'il est possible de déclarer la variable comme locale à l'instruction de boucle. Le programme suivant est lui aussi strictement équivalent au premier :

```
for (int i = 1; i < 11; i ++) System.out.println(i);
```

2. On peut mieux retenir ces abréviations en comprenant que s'il a été choisi d'écrire `a -= 2` et non pas `a =- 2` c'est parce que la seconde version serait ambigüe avec `a = (-2)`.

1.3.5 Variables finales

Lorsque l'on déclare une variable, que ce soit une variable globale `static`, ou une variable locale, on peut ajouter le mot-clé `final`. Dans ce cas, on interdit tout changement ultérieur de la valeur de la variable (qui peut donc être considérée comme une constante). Par exemple :

```
static final double pi = 3,14159265358979323846264338327 9;
```

Remarquons qu'il s'agit là d'une pure restriction, c'est-à-dire que le programme fonctionnera exactement pareil si l'on supprime le mot-clé `final`. Mais cela permet d'imposer une certaine discipline et d'éviter des erreurs ou des malentendus.

Chapitre 2

Compilation du noyau impératif

Pour expliquer les mécanismes du langage de programmation, et particulièrement ce qu'il advient effectivement dans la mémoire de l'ordinateur au cours de l'exécution d'un programme, nous allons voir comment ce programme Java est compilé, c'est-à-dire traduit dans un langage de bas niveau. Plus généralement, comprendre un tel mécanisme de compilation est intéressant sur plusieurs plans :

- pour détailler, comme indiqué, ce qui se passe lors de l'exécution d'un programme,
- cela permet aussi de présenter des traits essentiels de l'architecture d'un ordinateur (mémoire, pile, registres. . .),
- c'est également l'occasion de comprendre ce qu'est un langage de bas niveau, proche du langage machine,
- cela nous conduit à manipuler explicitement les programmes comme des constructions,
- enfin, de manière plus générale, s'intéresser aux questions de compilation permet d'avoir un angle de vue pertinent et global sur tout ce qui touche à la programmation.

Comme mentionné précédemment, en pratique, les programmes Java sont compilés vers une machine abstraite appelée la JVM *(Java Abstract Machine)*. D'autres langages comme C ou C++ sont compilés directement en langage machine. Ici, nous allons utiliser une machine définie pour l'occasion, que nous appellerons XVM. Cette machine est une version simplifiée d'une machine utilisée en pratique comme la JVM. Soulignons aussi qu'elle n'est finalement pas si différente, dans le principe, d'un processeur matériel, et donc les principes de compilation ne sont pas fondamentalement différents pour produire du code natif.

Nous présentons les instructions de la XVM au fur et à mesure qu'on les utilise pour compiler des mécanismes de plus en plus avancés de Java, dans ce chapitre et les suivants. On trouve un tableau récapitulatif complet détaillant l'ensemble des instructions dans l'appendice A.

2.1 Une machine à pile

2.1.1 Opérations

Comme toute machine abstraite ou tout processeur, la XVM exécute des instructions, l'une après l'autre. Les premières instructions que nous détaillons sont identiques à celles qu'on peut trouver sur une simple calculatrice :

instruction	pile avant	pile après
ADD	$x; y; \ldots$	$x + y; \ldots$
SUB	$x; y; \ldots$	$y - x; \ldots$
MUL	$x; y; \ldots$	$x \times y; \ldots$
DIV	$x; y; \ldots$	$y/x; \ldots$

FIGURE 2.1 – Les instructions arithmétiques

ADD pour l'addition,

SUB pour la soustraction,

MUL pour la multiplication,

DIV pour la division.

Ces opérations sont effectuées sur des nombres qui sont placés sur une *pile*. Cette pile sera au cœur de la machine.

Nous allons noter [] pour la pile vide, [2; 3] pour l'état de la pile où 2 est en haut de la pile et 3 au niveau en dessous. De même [2; 3; 4] désignera l'état ou de plus on trouve un troisième élément à l'étage inférieur, etc... On s'autorisera à écrire [2; 3; 4; ...] pour désigner les états où l'on a au moins trois éléments, ici 2, 3 et 4 en haut de la pile, et éventuellement d'autres éléments en dessous.

Les opérations binaires, comme ADD, effectuent l'opération sur les deux éléments les plus hauts de la pile, et les remplacent par le résultat de l'opération. Par exemple si l'on exécute ADD sur la pile [2 ;3 ;4], on obtient l'état de pile [5 ; 4]. Si l'on effectue l'opération MUL, on obtient l'état [6 ; 4]. Ces opérations font donc diminuer la hauteur de la pile de 1 ; l'exécution de ces opérations alors qu'il y a moins de deux éléments sur la pile donnera lieu à une erreur d'exécution.

La soustraction et la division n'étant pas commutatives, il faut préciser les rôles respectifs des deux opérandes. On le fait par le tableau de la figure 2.1 (remarquez que dans de tels tableaux, on omet les crochets [et] dans les descriptions de la pile).

2.1.2 Instructions agissant sur la pile

Il faut évidemment pouvoir charger des valeurs sur la pile. L'instruction PUSH(n) ajoute la valeur numérique n en haut de la pile, dont la hauteur s'accroit donc de 1. On se donne également, même si l'on n'en aura pas besoin tout de suite, une instruction POP qui diminue de 1 la hauteur de la pile, en effaçant donc le contenu du registre supérieur.

Avec les instructions présentées jusqu'ici, on dispose déjà d'une calculatrice quatre opérations, qui fonctionne en notation dite "polonaise inverse". Pour calculer 5-3 on effectue la séquence d'instructions suivante : PUSH(5); PUSH(3); SUB. A noter que, de ce point de vue, POP correspondrait à la touche d'effacement "C" ou "DEL" des calculatrices.

Solution page 183. **Exercice 2.1.1** Donnez une séquence d'instructions pour calculer $(5 - 3) \times 9 - 2$.

L'instruction FETCH permet de recopier un élément de la pile. Plus exactement, FETCH(i) prende le i-ème élément de la pile (en partant du haut, 0 désignant l'élément le plus haut) et le recopie en haut de la pile. Par exemple si on part de la pile [10; 8; 6; 4] exécuter FETCH(0) donne [10; 10; 8; 6; 4]. Alternativement, en partant toujours de la même pile, FETCH(1) donne [8; 10; 8; 6; 4], FETCH(2) donne [6; 10; 8; 6; 4], FETCH(3) donne [4; 10; 8; 6; 4], et FETCH(4) cause une erreur d'exécution.

2.1.3 Programmes et instructions de saut

Toutefois, à la différence d'une calculatrice simple, notre machine ne doit pas attendre qu'un utilisateur lui indique la prochaine instruction par la pression d'une touche. Elle doit exécuter un programme, c'est-à-dire une séquence d'instructions. Cela veut dire que :

1. Une partie dédiée de la mémoire contient une séquence d'instructions.
2. Comme toute partie de la mémoire, on accède à ces instructions par leurs adresses. Comme il s'agit d'une partie contigüe de la mémoire, ces adresses se suivent.
3. La machine est munie d'un registre particulier, appelé PC (pour *program counter*). A chaque étape, la machine exécute l'instruction qui se trouve à l'adresse mémoire indiquée par PC.
4. En général, la machine passe ensuite à l'instruction suivante, c'est-à-dire que PC est incrémenté de 1.

Une instruction spéciale STOP arrête l'exécution.

On dispose de plus d'instructions de *saut*. L'instruction de saut inconditionnel, GTO(i), (pour *go to*) stipule que la prochaine instruction qui sera exécutée sera l'instruction à l'adresse i (et non pas l'instruction de l'adresse suivante en mémoire). Autrement dit, GTO(i) donne au registre PC la valeur i.

Par exemple, les programmes suivant vont empiler puis dépiler 2 sur la pile, puis recommencer à l'infini.

adresse	instruction
0	PUSH(2)
1	POP
2	GTO(0)

adresse	instruction
77	PUSH(2)
78	POP
79	GTO(77)

Instruction	avant		après	
	pile	PC	pile	PC
ADD	$x; y; z \ldots$	n	$x+y; z \ldots$	$n+1$
SUB	$x; y; z \ldots$	n	$y-x; z \ldots$	$n+1$
MUL	$x; y; z \ldots$	n	$x \times y; z \ldots$	$n+1$
DIV	$x; y; z \ldots$	n	$y/x; z \ldots$	$n+1$
GTO(a)	$x; y; \ldots$	n	$x; y; \ldots$	a
GTZ(a)	$0; y; \ldots$	n	$y; \ldots$	a
GTZ(a)	$x; y; \ldots$ avec $x \neq 0$	n	$y; \ldots$	$n+1$

2.1.4 Présentation des programmes en langage machine

On voit que les deux programmes précédents sont essentiellement identiques et se différencient seulement par leurs emplacements en mémoire.

On se permettra donc de présenter un programme de la XVM indépendamment de son emplacement précis en mémoire en utilisant des étiquettes, ou *labels* pour indiquer les destinations possibles des sauts. Dans le cas précédant, cela donnera :

```
(α)  PUSH(2)
     POP
     GTO(α)
```

Dans le cas de programmes courts comme celui-ci, on s'autorisera également à l'écrire sur une ligne : (α)PUSH(2); POP; GTO(α).

2.1.5 Saut conditionnel

Une variante est l'instruction de saut conditionnel GTZ (pour *go to if zero*). Cette instruction teste la valeur en haut de la pile : si cette valeur est 0, alors GTZ(i) saute à l'adresse i. Sinon, on passe simplement à l'instruction suivante. Dans les deux cas, on "consomme" la valeur en haut de la pile (c'est-à-dire que la hauteur de la pile diminue est décrémentée).

Solution page 183. **Exercice 2.1.2** Ecrivez un programme XVM qui décrémente de 1 la valeur en haut de la pile, jusqu'à ce qu'elle soit égale à 0. ◇

2.1.6 Lire et écrire en mémoire

Les instructions de saut permettent des programmes de structure complexe. Pour donner à la machine les capacités d'un ordinateur, il faut encore ajouter la possibilité d'écrire et de lire des informations en mémoire.

La mémoire peut être vue comme un vaste tableau, c'est-à-dire une séquence d'emplacements mémoire. Chaque emplacement contient un mot mémoire. On accède à cet emplacement par son *adresse*. Cette adresse est elle-même un mot. On peut voir ces adresses comme des entiers successifs, même si les instructions de la XVM sont conçues pour éviter de faire des opérations arithmétiques sur les adresses.

Si d est une adresse, on notera mem(d) la valeur courante de la case mémoire d'adresse d.

Ces adresses mémoire peuvent elles aussi être stockées sur la pile. L'instruction READ permet de lire la mémoire. Elle suppose que le haut de la pile est une adresse mémoire d ; l'instruction lit la valeur correspondante en mémoire, c'est-à-dire mem(d), et la place en haut de la pile, à la place de d.

La hauteur de la pile est inchangée. On passe d'une pile $[d; \ldots]$ à la pile $[\text{mem}(d); \ldots]$. Pour placer en haut de la pile la valeur mem(d) on utilisera donc typiquement la séquence PUSH(d); READ.

Inversement, l'instruction WRITE permet de modifier la mémoire. Plus précisément, si la pile est de la forme $[d; n; \ldots]$, alors WRITE va placer la valeur n à l'adresse d. Ces deux éléments sont par ailleurs effacés de la pile, c'est-à-dire que la hauteur de la pile est décrémentée de 2.

Dans les chapitres suivants, nous présenterons encore quelques instructions supplémentaires de notre machine abstraite. Mais les instructions décrites ci-dessus suffisent à traduire le noyau impératif de Java. La figure de l'annexe A résume la description du comportement des opérations de la machine.

2.1.7 Tests

On se souvient que Java manipule aussi des valeurs booléennes (`true`, `false`) et que ces valeurs sont en particulier le résultats d'opérations de test : `==` pour l'égalité, `!=` pour la différence, `<`, `<=`, etc... pour les tests d'inégalité. On munit pour cela la XVM d'opérations correspondantes ; par exemple l'instruction EQ vérifie si les deux éléments au sommet de la pile sont égaux ou pas. On détaille en 2.3 le comportement de EQ et LEQ. Pour les autres instructions similaires on peut se reporter à l'annexe décrivant complètement la XVM à la fin de l'ouvrage.

2.2 Première phase : adressage des variables

Une première phase de la compilation d'un programme consiste à repérer les variables et à attribuer à chacune une adresse mémoire. Par exemple, on va assigner à la variable `x` l'adresse mémoire adr(`x`). Si adr(`x`) vaut 3, alors la valeur de `x` sera toujours stockée à l'adresse 3 de la mémoire.

Cette phase est très simple, puisque toutes les variables sont mentionnées au début du programme.

Si on considère le programme suivant :

```
class ExempleAdressage {
  static int n;
  static boolean pair;
  static int m;

  public static void main(String[] args) {
    n = 51;
    pair = n % 2 == 0;
    if (pair) {
            ...
    }
  }
}
```

Un résultat possible de la phase d'adressage est la fonction adr telle que :

$$\begin{aligned} \text{adr}(\texttt{n}) &= 2178 \\ \text{adr}(\texttt{pair}) &= 2179 \\ \text{adr}(\texttt{m}) &= 2180 \\ \text{adr}(x) & \text{ indéfini pour les autres identificateurs} \end{aligned}$$

Bien sûr, les valeurs des adresses ont été choisies arbitrairement ici. En pratique, le compilateur placera les variables dans une zone libre de la mémoire. On remarque que si, en général et comme dans cet exemple, les variables seront rangées les unes à côté des autres, avec des adresses mémoires consécutives, c'est juste pour une raison pratique ; ce n'est pas nécessaire pour le bon fonctionnement de la compilation.

Deux points importants à noter sont :

- Il s'agit bien d'une phase de *compilation*. C'est-à-dire que ces adresses sont choisies *avant* d'exécuter le programme. Elles sont même choisies lorsqu'on génère le programme compilé.
- L'adresse de chaque variable est donc fixée et ne changera pas au cours de l'exécution du programme. C'est pour cela qu'on parle de variable *statiques*. On verra plus tard des constructions faisant intervenir des variables dynamiques, faisant intervenir des adresses mémoires qui changent au cours de l'exécution.

2.3 Compilation des expressions

Une fois l'adressage des variables résolu, on peut compiler les programmes. Pour cela, on va d'abord décrire comment on compile les expressions Java (comme définies en 1.1.1, c'est-à-dire par exemple `2+x`). On regardera la compilation des instructions dans la section suivante.

On traduit chaque expression `e` en une séquence d'instructions qu'on note ⟦`e`⟧. Le principe est que l'exécution de ⟦`e`⟧ va placer en haut de la pile la valeur correspondant à `e`.

Constante numérique

Si l'expression est une constante numérique n, la forme compilée est l'instruction PUSH(n). Par exemple l'expression 7 est compilée vers PUSH(7).

Variable

Si l'expression est une variable `x`, la forme compilée est PUSH(adr(`x`)) READ. C'est-à-dire la séquence d'instructions qui place en haut de la pile la valeur se trouvant à l'adresse mémoire adr(`x`).

Opérations arithmétiques

Si l'expression est de la forme `e1 + e2`, alors la forme compilée est la séquence ⟦`e1`⟧ ⟦`e2`⟧ ADD.

On comprend que l'exécution de la séquence ⟦`e1`⟧ va placer en haut de la pile la valeur correspondant à `e1`. Ensuite la séquence ⟦`e2`⟧ va placer au dessus la valeur correspondant à `e2`. Enfin, ADD va remplacer ces deux valeurs par le résultat de l'addition, qui est le résultat attendu.

Les autres opérations arithmétiques ou booléennes sont évidemment traitées de manière similaire :

- la forme compilée de `e1` − `e2` est : ⟦`e1`⟧ ⟦`e2`⟧ SUB,
- la forme compilée de `e1` ∗ `e2` est : ⟦`e1`⟧ ⟦`e2`⟧ MULT,
- la forme compilée de `e1/e2` est : ⟦`e1`⟧ ⟦`e2`⟧ DIV.

Opérations de comparaison

On dispose de deux instructions permettant de compiler les expressions booléennes comme `x == y` ou `x < y`. Ces instructions sont :

EQ (pour *equal*) qui teste l'égalité des deux éléments en haut de la pile,

LEQ (pour *less or equal*) qui teste si le deuxième élément de la pile est inférieur ou égal au premier.

instruction	pile avant	pile après	condition
EQ	$x; x; \ldots$	$1; \ldots$	
EQ	$x; y; \ldots$	$0; \ldots$	si $x \neq y$
LEQ	$x; y; \ldots$	$1; \ldots$	si $y \leq x$
LEQ	$x; y; \ldots$	$0; \ldots$	si $x < y$

La forme compilée de `e1 == e2` est[1] ⟦`e1`⟧ ⟦`e2`⟧ EQ. Celle de `e1 <= e2` est ⟦`e1`⟧ ⟦`e2`⟧ LEQ.

1. On pourrait aussi prendre ⟦`e2`⟧ ⟦`e1`⟧ EQ ; mais si on veut tout à fait respecter les spécifications de Java, il faut, lors de l'exécution du programme, évaluer `e1` avant `e2`. On traite cette question d'ordre d'évaluation plus en détail pour les arguments de fonction en 3.3.

Négation booléenne

On voit ci-dessus qu'on représente `false` par l'entier 0. On décide que `true` peut être représenté par tout entier non nul. On dispose d'une instruction NOT correspondant à la négation booléenne, qui transforme `true` en `false`, et vice-versa, en haut de la pile :

instruction	pile avant	pile après	condition
NOT	$0; y; \ldots$	$1; y; \ldots$	
NOT	$x; y; \ldots$	$0; \ldots$	si $x \neq 0$

Exercice 2.3.1 Proposez une manière de compiler une expression `e1 < e2`. ◇ Solution page 183.

Exercice 2.3.2 Proposez une manière de compiler l'opérateur unaire du nombre opposé `-e`. ◇ Solution page 184.

2.4 Compilation des programmes

On peut maintenant décrire la compilation des programmes. C'est-à-dire que chaque instruction ou séquence d'instructions Java sera traduite en une séquence d'instructions XVM. On garde la même notation que pour les expressions : si `p` est une instruction ou séquence d'instructions Java, elle est traduite en ⟦p⟧ qui est une séquence d'instructions XVM.

2.4.1 Compilation de l'affectation

On a déjà l'essentiel des éléments pour traduire en langage machine une affectation `x = e`. La séquence d'instructions ⟦e⟧ place en haut de la pile la valeur correspondant à `e`. Il suffit alors de la placer au bon endroit de la mémoire par PUSH(adr(x)) WRITE. Notons que la pile est ensuite dans le même état qu'avant l'exécution de ⟦x = e⟧.

Au final, on a donc :

$$\llbracket \texttt{x = e} \rrbracket \equiv \begin{array}{l} \llbracket \texttt{e} \rrbracket \\ \text{PUSH(adr(x))} \\ \text{WRITE} \end{array}$$

2.4.2 Compilation de la séquence

C'est assez évident : la forme compilée d'une séquence d'instructions $\{p_1\ p_2\}$ est obtenue en enchainant les deux formes compilées.

$$\llbracket p_1\ p_2 \rrbracket \equiv \llbracket p_1 \rrbracket \llbracket p_2 \rrbracket.$$

On remarque que, du coup, compiler $\{p_1\ \{p_2\ p_3\}\}$ et $\{\{p_1\ p_2\}\ p_3\}$ donne effectivement exactement le même code.

2.4.3 Compilation du choix conditionnel

On calcule l'expression sur lequel se fait le choix (`e`) puis le saut conditionnel permet d'exécuter la bonne branche du programme. Le code ⟦`if` (`e`) p_1 `else` p_2⟧ est le suivant :

```
    ⟦e⟧          ; on place la valeur de e sur la pile
    GTZ(α)       ; 0 correspond au cas false
    ⟦p1⟧
    GTO(β)
(α) ⟦p2⟧
(β) ...          ; ici se situe la suite du programme
```

2.4.4 Compilation de la boucle

Le code ⟦`while (e) p`⟧ est :

```
(α) ⟦e⟧          ; la condition de boucle
    GTZ(β)       ; si elle est fausse on sort de la boucle
    ⟦p⟧          ; sinon on exécute le corps de la boucle
    GTO(α)       ; et on recommence
(β) ...          ; ici la suite du programme
```

Solution page 184. **Exercice 2.4.1 (recommandé)** Java propose une seconde version de la boucle, où le test est effectué après une première exécution de la boucle. La syntaxe est : `do p while (e)`. Proposez une compilation de cette construction. ◇

2.4.5 Un cas particulier : les expressions booléennes

En Java comme dans la plupart des langages de programmation modernes, il y a une optimisation importante dans la manière dont sont évaluées les expressions booléennes. Plus précisément les expressions de la forme conjonction `e1 && e2` et disjonction `e1 || e2`. Dans les deux cas, l'expression de gauche `e1` est évaluée en premier. Si le résultat de cette évaluation donne `false`, alors `e1 && e2` est directement évalué également vers `false`, sans que la valeur de `e2` ne soit calculée. Inversement, si `e1` s'évalue vers `true`, alors `e1 || e2` vaut également `true` sans que l'on ait besoin de calculer `e2`.

En d'autres termes, le calcul de `e1 && e2` est effectué comme suit :

1. On calcule la valeur de `e1`. Si celle-ci vaut `false`, le résultat final vaut aussi `false`.

2. Sinon on calcule la valeur de `e2`. Le résultat final sera alors la valeur de `e2`.

Cela veut dire qu'on ne peut pas compiler les opérateurs booléens `&&` et `||` en utilisant simplement une opération de la machine, comme on le fait pour les opérateurs arithmétiques. On peut en revanche compiler l'expression `e1 && e2` de la manière suivante :

```
    ⟦e1⟧         ; on calcule la valeur de e1
    GTZ(α)       ; si elle vaut false on saute le calcul de e2
    ⟦e2⟧         ; on place la valeur de e2 sur la pile
    GTO(β)
(α) PUSH(0)      ; on remet false sur la pile - il a été consommé par GTZ
(β) ...          ; suite du programme
```

Solution page 184. **Exercice 2.4.2** Proposez une manière de compiler `e1 || e2`.

2.5 Comment fonctionne la pile

2.5.1 Le pointeur de pile (SP)

Pour les chapitres suivants, il est utile de décrire plus précisément comment fonctionne la pile d'une machine de "bas niveau" comme la XVM. Pour résumer, la pile est gérée en utilisant des mécanismes simples et rapides.

Le contenu de la pile est stocké dans une partie de la mémoire réservée à cet effet. La taille maximale de la pile est fixée lors du lancement de la machine et ne peut être changée ensuite. Si on appelle t cette taille maximale, la pile sera stockée sur t adresses contigües de la mémoire. Appelons ls la première adresse de la pile ; celle-ci sera donc stockée sur les adresses ls à $ls+t-1$.

L'élément le plus bas de la pile sera stocké en ls, c'est-dire la première adresse, l'élément suivant en $ls+1$ et ainsi de suite.

Un registre spécial de la machine, appelé SP pour *stack pointer* indique où se trouve le haut de la pile. On peut prendre comme convention que SP est l'adresse de la première adresse libre de la pile. Donc si la pile est vide, SP vaut ls, si la pile contient un élément, SP vaut $ls+1$, si la pile contient deux éléments, SP vaut $ls+2$ etc. . .

Autrement dit, la pile correspond au contenu des adresses mémoires ls, $ls+1$, . . . SP -1. Le contenu des adresses SP à $ls+t$ est sans importance.

On peut alors mieux détailler ce que font les instructions qui agissent sur la pile :

- PUSH(n) commence par écrire n à l'adresse SP, puis incrémente SP de 1.
- POP va simplement décrémenter SP de 1.
- FETCH(i) va recopier la valeur se trouvant à l'adresse SP $-1-i$ à l'adresse SP, puis incrémenter SP de 1.

Exercice 2.5.1 Décrivez de la même façon les actions des opérations comme ADD. ◇ Solution page 184.

2.5.2 Débordements

On peut également préciser les erreurs d'exécutions dues à une pile trop pleine *(overflow)* ou trop vide *(underflow)*. La première a lieu si une instruction devrait donner lieu à une valeur de SP supérieure à $ls+t$, par exemple une séquence de $t+1$ PUSH. La seconde si une instruction devrait donner lieu à une valeur de SP inférieure à ls ; par exemple le programme composé de la seule instruction POP.

Dans les deux cas, une telle erreur est fatale et l'exécution du programme se termine avec un message d'erreur. On verra dans les chapitres suivants des programmes Java donnant lieu à un débordement de pile.

2.6 Différences entre la XVM et un processeur réel

Pour répondre complètement à cette question, il faudrait décrire précisément un processeur physique, ce qui sortirait tout-à-fait des objectifs de ce cours. Mais peut donner un certain nombre de points communs et de différences.

2.6.1 Instructions et horloge

Un point commun est que le processeur, tout comme la XVM, possède un *jeu d'instructions*, c'est-à-dire un ensemble fini d'instructions (ADD, GTO...). Mais ce jeu d'instruction varie grandement suivant le modèle de processeur. Remarquons toutefois que le jeu d'instruction de la XVM est particulièrement réduit pour simplifier la présentation, alors qu'un processeur réel, tout comme d'ailleurs une machine virtuelle comme la JVM qui est utilisée dans un cadre industriel, aura un jeu d'instructions plus important pour favoriser la vitesse d'exécution.

La *fréquence d'horloge* donne une indication de la vitesse à laquelle le processeur exécute les instructions. Comme vous le savez peut-être, les processeurs modernes tournent à des fréquences de 3 à 4 GHz, voire plus s'ils sont très refroidis. Les processeurs des années 1970 tournaient à quelques MHz. A noter toutefois que le nombre exacts de tics d'horloge nécessaires par instruction varie suivant l'architecture du processeur et l'instruction.

2.6.2 Registres

Une différence importante est qu'un vrai processeur dispose de plus de registres, c'est-à-dire d'emplacements mémoires propres, que la XVM. En particulier, les opérations arithmétiques sont en général effectuées sur les registres et pas sur la pile. Le nombre de ces registres varie grandement d'une architecture de processeur à l'autre. Les processeurs sont souvent caractérisés par la taille de ces registres ; c'est pourquoi on parle de processeur 8, 16, 32 ou 64 bits. De fait, on utilise aujourd'hui des processeurs extrèmenent divers ; les processeurs modernes 64 bits (core I7 Intel, gamme Bulldozer chez AMD...) comportent plus de un ou deux milliards de transistors. D'un autre côté, des appareils ménagers ou des calculatrices utilisent des versions modernisées de processeurs 8 bits conçus dans les années 1970 ; la calculatrice TI-84, par exemple, utilise un eZ80 qui est basé sur le Zilog Z80 de 1976 composé de 8500 transistors.

2.6.3 Registres flottants

Depuis les années 1990, les modèles avancés de processeurs manipulent non seulement des nombres entiers, mais aussi de nombres à virgule flottante, c'est-à-dire des nombres décimaux dans le format mantisse et exposant. Le comportement des opérations arithmétiques sur ces nombres, c'est-à-dire comment les opérations d'arrondi doivent se faire, est précisément défini par l'IEEE [2]. Une anecdote célèbre est que l'une des premières générations de microprocesseurs Intel Pentium en 1995 ne respectait parfois pas cette norme pour l'opération de division. Cela était peu ou pas visible dans la plupart des conditions d'utilisation mais pouvait être dramatique dans certains cas. Plus généralement, le domaine de l'arithmétique des ordinateurs est un domaine tout-à-fait intéressant, important et surprenant ; voir par exemple (Muller et al., 2010) pour une présentation détaillée de ce domaine, ou le *bug* du missile Patriot (Arnold, 2000) pour un exemple frappant de conséquence d'erreur d'arrondi.

2.6.4 Cœurs

Les processeurs modernes sont souvent annoncés comme ayant deux, quatre, ou plus de *cœurs*. Cela signifie essentiellement qu'ils sont composés d'autant de processeurs unitaires tournant en parallèle, même si ceux-ci partagent certaines ressources. Cette évolution a été rendue nécessaire car, à cause de l'accélération des fréquences horloge et de l'augmentation de

2. *Institute of Electrical and Electronics Engineers*, société savante qui effectue en particulier le travail de normalisation.

la densité des circuits, on se heurte à des limites physiques, en particulier dues à des difficultés de dissipation de chaleur.

Ces cœurs multiples peuvent servir soit à traiter en parallèle des tâches indépendantes (faire un calcul Java tout en regardant une vidéo), soit à traiter plus vite un problème particulier. Mais dans ce cas, ce problème aura d'abord dû être parallèlisé. Le domaine également vaste du calcul parallèle et réparti est abordé dans le cours INF431 en deuxième année.

2.6.5 Mémoire cache

L'accès en lecture et écriture à la mémoire principale (RAM) de l'ordinateur prend du temps. Pour limiter cela, les processeurs disposent depuis longtemps d'une "mémoire cache". C'est-à-dire qu'une partie de la RAM est recopiée dans cette mémoire cache qui se trouve dans le processeur. L'avantage est que le temps d'accès au cache est bien plus réduit que pour la RAM. La manière dont est gérée cette mémoire cache est une source de questions et techniques algorithmiques très diverses. Les processeurs modernes possèdent plusieurs niveaux de caches (jusqu'à 12 méga-octets pour le niveau 3 d'un processeur haut de gamme).

2.7 Dr. *while* et M. *Goto*

Même si le noyau impératif décrit dans ces deux premiers chapitres est très rudimentaire, il porte néanmoins un premier choix structurant : celui de se restreindre aux boucles *while* et de renoncer aux instructions de saut. En effet, l'instruction GTO (ou *Go to*) n'a pas d'équivalent direct dans le langage source. Si une machine de bas niveau comme la XVM ou un processeur matériel a naturellement une telle instruction, la plupart des langages de programmation modernes en sont dépourvus. Cela n'a pas toujours été le cas, et cette instruction est présente dans la plupart des langages anciens comme Fortran ou Basic. On peut y écrire des programmes comme :

```
10      A = A+1
20      GOTO 10
```

Si l'on a renoncé à conserver des instructions de sauts dans la plupart des langages sources, c'est qu'il est très difficile de raisonner à propos de programmes comportant des sauts arbitraires, et donc également difficile de les corriger ou débugger. L'un des premiers à réclamer la suppression du GOTO a été le grand informaticien Edsger Dijkstra, dans une célèbre lettre *Go To Statement Considered Harmful* (Dijkstra, 1979) publiée en originellement en 1968 par les Communications de l'ACM. A l'époque, ce point de vue a largement été considéré comme provocateur, et il a été long et difficile de convaincre des programmeurs habitués à utiliser des langages avec GOTO.

On peut, en revanche, raisonner à propos de programmes comportant des boucles *while*. Une notion clé est alors la notion d'*invariant*. Prenons un exemple simple ; soit le programme suivant où `n` est un entier positif (`Math.pow(x,y)` calcule `x` puissance `y`) :

```
int i = 1;
int r = 0;
while (i <= n) {
   r = r + 1 / Math.pow(2, i);
   i++;
 }
```

Les valeurs successives de `r` sont donc : 0, $\frac{1}{2}$, $\frac{1}{2}+\frac{1}{4}$, $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}$, *etc.*...

On veut montrer qu'à la fin de l'exécution de ce programme, la valeur de `r` est strictement inférieure à 1. Il faut pour cela identifier la propriété `r == 1 - 1 / Math.pow(2,i-1)` (ou pour être tout à fait précis la propriété `r == 1 - 1 / Math.pow(2,i-1) && i > 0`). En effet cette propriété vérifie deux points :

– Elle implique le résultat recherché, c'est-à-dire `r < 1`.

– Elle est préservée par l'exécution du corps de la boucle : si elle est vraie elle reste également après `r = r + 1 / Math.pow(2, i); i++;`.

C'est évidemment à cause du deuxième point qu'on appelle une telle propriété un *invariant de boucle*. Nous ne rentrons pas dans les détails ici, mais on peut comprendre qu'un invariant est en fait une propriété de récurrence.

L'exemple ci-dessus est simple, mais pour certains programmes, il peut être difficile de trouver un invariant de boucle correct. En fait, c'est parce qu'il n'existe pas de méthode universelle pour trouver les invariants de boucle que la preuve de programmes est indécidable. C'est essentiellement la même raison qui fait qu'il n'existe pas de méthode universelle pour trouver des preuves mathématiques (ou pour décider si une propriété mathématique est vraie ou fausse).

Nous n'approfondissons pas ici ce sujet, qui n'est pas celui du cours. Pour découvrir quelques-uns des nombreux liens entre informatique et logique mathématique, mentionnons à nouveau le cours INF412 d'Olivier Bournez (Bournez, 2017), le cours de troisième année de Samuel Mimram (Mimram, 2020), le livre de vulgarisation d'Hofstadter (Hofstadter, 1979), celui, plus précis, de Gilles Dowek (Dowek, 2015) ou son ancien cours à l'X (Dowek, 2010) ou la très bonne bande dessinée *Logicomix* (Doxiadis et al., 2009).

Chapitre 3

Fonctions

3.1 Généralités et syntaxe

On a indiqué dans le chapitre 1 que le noyau impératif suffisait pour coder toutes les machines de Turing. Toutefois, afin de pouvoir programmer de manière confortable, il est essentiel d'étendre le langage de programmation ; le premier ajout que nous traitons est celui de fonctions.

3.1.1 Fonctions sans argument

Une première utilité des fonctions dans les langages de programmation, est d'éviter de dupliquer du code. Gilles Dowek (Dowek, 2008) donne un exemple particulièrement simple. Supposons qu'on veuille, à plusieurs reprises afficher trois sauts de lignes. On peut le faire de manière élémentaire en écrivant :

```
System.out.println();
System.out.println();
System.out.println();
```

Il est plus concis et élégant de regrouper ce code dans une fonction, en déclarant dans la classe. Notons que, dans ce chapitre, les déclarations de fonctions sont aussi munies du mot clé `static` :

```
static void sauterTroisLignes() {
   System.out.println();
   System.out.println();
   System.out.println();
  }
```

On peut alors remplacer les trois sauts de lignes par l'unique instruction :

```
sauterTroisLignes();
```

Cet exemple est bien sûr quasi-trivial, mais illustre très précisément la première chose qu'on attend des fonctions en programmation : rendre les programmes plus concis et plus lisibles. Si on peut se passer de fonctions, essentiellement en remplaçant tout appel de fonction par le corps de celle-ci, cela se fait au coup d'une explosion de la taille du programme, qui serait intolérable en pratique.

3.1.2 Fonctions avec argument(s)

Dans l'exemple précedent, on peut trouver plus élégant la version suivante :

```
static void sauterTroisLignes() {
  for (int i = 0; i < 3; i++)
        System.out.println();
  }
```

Mais surtout, on peut souhaiter ne pas se limiter au saut de trois lignes, et paramétrer la fonction par un argument. Cela donne :

```
static void sauterNLignes(int n) {
    for (int i = 0; i < n; i++)
              System.out.println();
  }
```

On récupère la fonction précédente avec l'instruction `sauterNLignes(3);` mais on peut plus généralement faire `sauterNLignes(e);` où `e` est une expression de type `int`.

Voici un exemple de fonction prenant deux arguments :

```
static void afficherNfois(int n, String s) {
  for (int i = 0; i < n; i++)
            System.out.println(s);
  }
```

On remarque que, comme lors d'une déclaration de variable, il faut mentionner explicitement le type de chaque argument. Contrairement, par exemple, à Caml, le compilateur Java ne peut pas "deviner" le type d'une variable ; techniquement, on dit que Java n'effectue pas d'*inférence de type*.

Un point technique est qu'il est autorisé de changer la valeur des arguments dans le corps de la fonction. Autrement dit, les arguments sont vus comme des variables locales au corps de la fonction. Voici deux variantes de la fonction précédente qui exploitent cette possibilité :

```
static void afficherNfois(int n, String s) {
   while (n > 0) {
       System.out.println(s);
       n--;
     }
  }

static void afficherNfois(int n, String s) {
    for (n = n; n > 0; n--)
          System.out.println(s);
  }
```

3.1.3 Fonctions renvoyant une valeur

Dans les exemples ci-dessus, le mot-clé `void`, qu'on peut traduire peu ou prou par *vide*, signifie que les fonctions déclarées ne rendent pas de résultat. Leur intérêt repose uniquement dans l'effet qu'elle produisent à chaque fois qu'elles sont appelées (ici d'afficher des sauts de ligne). Mathématiquement, une fonction est quelque chose qui, à des arguments, associe un résultat. Voici, en Java, une telle fonction simple :

```
static int factorielle(int n) {
    int r = n;
    while (n > 1) {
```

```
        n--;
        r = r * n;
     }
    return(r);
  }
```

Les points nouveaux sont le type de retour au début de la déclaration de fonction (`static int factorielle`) et l'instruction **`return`** qui permet de renvoyer l'argument.

On remarque également que cet exemple montre que l'on peut, à l'intérieur de la fonction, utiliser l'argument comme une variable locale : la valeur de `n` est modifiée au cours du calcul de la fonction. Si l'on veut empêcher cela, on peut déclarer un argument de fonction comme `final`.

L'instruction **`return`** peut être invoquée partout dans le corps de la fonction. Elle interrompt l'exécution du corps de la fonction, voire de structures comme des boucles. Par exemple, la fonction suivante calcule le produit des valeurs contenues dans un tableau d'entiers. On voit qu'on peut utiliser **`return`** pour sortir prématurément à la fois de la boucle et de la fonction lorsqu'on rencontre un zéro :

```
static int produitTab(int[] t) {
   int r = 1;
   for (int i = 0; i < t.length; i++) {
     if (t[i] == 0) return(0);
     r = r * t[i];
    }
  return(r);
}
```

Dans le cas de fonctions sans résultat, on a vu qu'il n'est pas nécessaire d'utiliser **`return`**. On peut toutefois l'utiliser, sans argument (syntaxe : **`return;`**), pour sortir plus rapidement de la fonction.

3.2 Fonctions et variables globables

Les variables globales, c'est-à-dire les variables statiques déclarées "au début" de la classe sont visibles depuis les fonctions. Par exemple ici la fonction `reset()` va remettre à 0 la variable `n` et le programme va afficher 0 :

```
class Cache {
    static int n = 1;

    static void reset() { n = 0; }

    public static void main(String[] args) {
        reset();
        System.out.println(n);   }
}
```

En revanche, si on remplace le corps de `main` par les trois lignes suivantes, `reset` va toujours mettre à 0 la variable globale `n`, alors que c'est la variable locale qui sera affichée :

```
        int n = 2;
        reset();
        System.out.println(n);
```

Exercice 3.2.1 (recommandé) Que se passe-t-il avec le programme suivant ? Solution page 184.

```
 class Cache {

    static void reset() { n = 0; }

    public static void main(String[] args) {
        int n = 3;
        reset();
        System.out.println(n);
    }
}
```

3.3 L'évaluation des arguments

Considérons la fonction suivante, qui affiche une chaine de caractères puis rend l'argument en résultat :

```
static int f (int x) {
        System.out.println("coucou");
        return(x);
}
```

ainsi que la fonction suivante qui utilise deux fois son argument :

```
static int double(int x) {
    System.out.println("bonjour");
    return(x + x);
}
```

On voit qu'on a, dans les deux cas, des fonctions qui produisent un effet de bord (afficher une chaine de caractères) puis rendent un résultat. Observons ce qui se passe lorsqu'on évalue le programme `double(f(2))`. L'ordinateur va afficher :

```
coucou
bonjour
```

On voit que `coucou` est affiché avant `bonjour`, c'est-à-dire que l'argument `f(2)` est calculé avant que l'on n'exécute le corps de la fonction `double`. Aussi, l'argument `f(2)` n'est calculé qu'une fois, même si `double` fait référence à son argument plus d'une fois.

L'ordre d'évaluation des arguments

Regardons maintenant une fonction à plusieurs arguments :

```
static int somme(int x, int y) { return(x + y); }
```

L'exécution du programme `somme(double(2), f(3))` va afficher :

```
bonjour
coucou
```

Cela veut dire que `double(2)` a été calculé avant `f(3)`. En effet, Java spécifie que les arguments d'une fonction doivent être évalués de gauche à droite.

Exercice 3.3.1 Que se passe-t-il si on change la définition de la fonction `somme` en remplaçant `x + y` par `y + x` ? ◇ Solution page 184.

D'autres langages de programmation, comme Caml, ne précisent pas dans quel ordre les arguments d'une fonction doivent être évalués [1]. C'est pourquoi on conseille parfois d'éviter d'écrire des programmes dont le comportement dépend de l'ordre d'évaluation des arguments.

3.4 La fonction `main`

Dans les chapitres précédents, on a présenté les programmes Java élémentaires comme des classes composées de :

- la déclaration des variables globales,
- le corps du programme, placé dans une fonction nommée `main`.

Plus précisément, la déclaration de `main` est :

```
public static void main(String[] args) { ... }
```

On comprend donc qu'il s'agit d'une fonction nommée `main` ne rendant pas de résultat. On verra plus tard la signification du mot-clé `public`. L'argument de la fonction est donc un tableau de chaînes de caractères. Il s'agit d'arguments qui peuvent être passés lorsque l'on appelle le programme.

Par exemple la commande unix suivante :

```
java NomProgramme bonjour bonsoir
```

Va appeler la fonction `main` de la classe `NomProgramme` en lui passant en argument un tableau de taille 2, contenant dans l'ordre les chaînes de caractères `"bonjour"` et `"bonsoir"`.

On peut donc tout-à-fait construire des classes Java ne comportant pas de fonction `public static void main(String[] a)`. Juste, on ne pourra pas appeler ces classes par la commande `java`. En revanche, elle peuvent comporter du code et des données utilisés par la classe principale. Cela apparaitra progressivement dans les chapitres suivants.

Exercice 3.4.1 Ces deux programmes produisent-ils des résultats différents ? Solution page 184.

```
class C1 {
  static int n = 3;
  static void f() {
     n = n * 2;
  }
  public static void main(String[] a) {
    f();
    f();
    System.out.println(n);
  }
}

class C2 {
  static int n;
  static void f() {
     n = 3;
     n = n * 2;
```

1. Voir paragraphe 7.7.1 de (Leroy et al., 2017).

```
  }
  public static void main(String[] a) {
    f();
    f();
    System.out.println(n);
  }
}
```

3.5 La surcharge

Il est possible de définir, en Java, plusieurs fonctions portant le même nom, à condition que les types ou le nombres de leur arguments soient différents. Par exemple :

```
static int somme(int n, int m) {
    return(n+m);
}
static int somme(int n, int m, int p) {
    return(n+m+p);
}
```

Dans ce cas, le code suivant appellera d'abord la fonction à deux arguments, puis la fonction à trois arguments, et affichera donc 7 puis 20 :

```
System.out.println(somme(3,4));
System.out.println(somme(10,8,2));
```

On comprend que c'est le nombre d'arguments, ou éventuellement leur type, qui permet de déterminer quelle fonction doit être appelée.

Cette possibilité de Java est appelée la surcharge, ou *overloading*. Ici c'est le nom de fonction `somme` qui est surchargé.

Il est important de noter que la résolution *de la surcharge est effectuée au moment de la compilation et non pas au moment de l'exécution du programme. C'est le compilateur, qui, en fonction des informations de typage, choisit la "bonne" fonction* `somme`. Ce point distingue la surcharge de l'héritage que nous découvrirons au chapitre 6.

À noter que le type du résultat de la fonction n'est pas pris en compte pour la résolution de la surcharge. Par exemple, le compilateur ne saura pas utiliser la surcharge pour différencier les deux fonctions suivantes et refusera ce code :

```
static String f(int x) { ... }
static float f(int x) { ... }
```

Solution page 184. **Exercice 3.5.1** Que produit le programme suivant ?

```
class C {
static float f (int n) {
   return(n + 1);
}
static float f (float n) {
  return(n * 2);
}
public static void main(String[] a){
  int m = 4;
  System.out.println(f(f(m)));
}
}
```

Exercice 3.5.2 Pouvez-vous deviner ce qui se passe quand on compile puis lance le programme suivant ? Solution page 185.

```
class MultiMain {
    public static void main(String a) {
        System.out.println("bonjour");
    }
    public static void main(int s) {
        System.out.println("bonsoir");
    }
    public static void main(String[] a) {
        System.out.println("bonne nuit");
    }
}
```

3.6 Instructions, expressions et vice-versa

On a indiqué en 1.1.8 que certaines instructions, comme `x = e;` peuvent être utilisées comme des expressions. Avec l'introduction de fonctions, on est amené à permettre d'utiliser des expressions comme des instructions. Considérons une fonction qui rende un résultat mais donc l'évaluation produise un effet. Par exemple :

```
static int f(int x) {
  a = a + x;
  System.out.println(a);
  return(a);
  }
```

où `a` est une variable statique.

On peut alors vouloir exécuter, par exemple, `f(4)`, sans utiliser le résultat de la fonction. On peut faire cela en utilisant `f(4);` comme une instruction : l'expression est calculée puis le résultat est effacé.

Exercice 3.6.1 Comment serait compilée une expression comme `f(3);` lorsqu'elle est utilisée comme une expression ? ◇ Solution page 185.

Inversement, on peut donner un peu plus de détail sur les instructions qui peuvent être également utilisées comme des expressions. On le fait ici, sans chercher à être exhaustif :

- L'instruction `x = e;` peut être utilisée comme expression. Elle vaut alors `e`, donc la nouvelle valeur de `x`. Cela permet d'écrire des choses comme `x = y = e;`
- Comme on l'a dit, l'instruction `++x;` est équivalente à `x = x+1;` aussi elle incrémente la variable `x` et, si elle est utilisée comme expression, vaut la nouvelle valeur de `x`.
- En revanche, l'instruction `x++;` incrémente également `x` mais vaut *l'ancienne* valeur de `x` si elle est utilisée comme expression.

Exercice 3.6.2 Proposez des manières de compiler `x++` et `++x` lorsqu'elles sont utilisées comme des expressions (par exemple lorsque `x` est une variable statique). ◇ Solution page 185.

Si l'on regarde les solutions des deux exercices ci-dessus, on voit que l'utilisation de ces instructions comme des expressions, peut permettre, dans certains cas, d'aboutir à un code compilé un peu plus efficace. En particulier, on limite l'utilisation d'instructions READ et WRITE en

leur substituant une seule instruction FETCH(0). C'est cela qui a historiquement justifié l'introduction de telles constructions. Aujourd'hui, c'est moins utile, car les compilateurs modernes effectuent de nombreuses optimisations du code qu'ils génèrent, ce qui rend ces constructions un peu superflues. On parle un peu plus des optimisations de compilation en 8.8.

Chapitre 4

La compilation des fonctions

4.1 Principes généraux

L'une des raisons d'être des fonctions est donc d'éviter de devoir écrire plusieurs fois le même code dans un programme. On va retrouver cette préoccupation dans le code compilé. Prenons une fonction `f` et désignons `p` le code de son corps. Supposons qu'elle prenne un argument et rende un résultat de type **`int`** :

```
static int f(int n) {
  p
  }
```

On va compiler le corps de cette fonction et placer ce code ⟦`p`⟧ en un endroit de la mémoire, qu'on appellera adr(`f`).

Il nous faut donc comprendre comment compiler les appels à cette fonction, c'est-à-dire déterminer les instructions machine qui composent ⟦`f(e)`⟧. Il faudra pour cela des mécanismes pour :

1. Appeler ⟦`p`⟧ depuis différents endroits du code (les appels `f(3)` et `f(5)` dans la figure 4.1) et revenir au point d'appel.
2. Passer à ⟦`p`⟧ la valeur de l'argument `n`.
3. Récupérer le résultat de la fonction après l'exécution de ⟦`p`⟧.

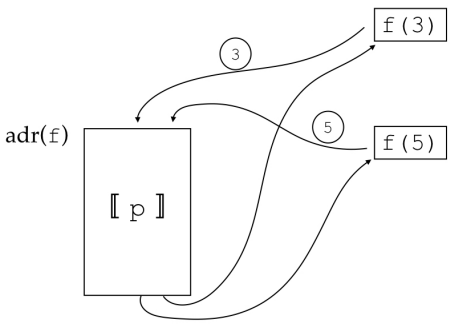


FIGURE 4.1 – Ce qu'on attend du code compilé d'une fonction

Nous présentons les mécanismes correspondants dans cet ordre. On va voir que pour les deux premiers points, c'est une utilisation astucieuse de la pile qui nous permettra de réussir.

4.2 Fonction sans argument et sans résultat

Le premier exemple de fonction que nous avions donné dans le chapitre précédent, `sauterTroisLignes` en 3.1.1, permet simplement d'exécuter une portion de code en tout endroit du programme. Appeler une telle fonction dans un programme Java, c'est juste exécuter le code de cette fonction, puis continuer l'exécution du programme principal ; c'est-à-dire revenir à l'instruction du programme principal qui suit immédiatement l'appel de la fonction.

La machine abstraite nous fournit deux instructions qui permettent cela très facilement : GSB(pour *Go to Subroutine*) et RET (pour *return*). La caractéristique commune des ces deux instructions est qu'elles rangent des adresses dans la pile :

- L'instruction GSB(i) est identique à GTO(i), à la différence qu'avant d'effectuer le saut à l'adresse i, elle recopie en haut de la pile l'adresse de la prochaine instruction.
- L'instruction RET effectue un saut inconditionnel à l'adresse stockée en haut de la pile (et cette adresse est ôtée de la pile).

On va voir plus bas que les deux instructions sont en fait un petit peu plus compliquées. Mais cette version simplifiée permet de saisir comment une partie du code peut être partagée.

Instruction	avant l'instruction		après l'instruction	
	PC	pile	PC	pile
GSB(i)	n	$x; y; \ldots$	i	$n+1; x; y; \ldots$
RET	n	$i; x; y; \ldots$	i	$x; y; \ldots$

En d'autre terme, GSB permet d'aller exécuter une séquence de code. Si celle-ci se termine par RET, on revient au point de départ.

Il est alors très facile de compiler une fonction sans argument ni résultat, comme la fonction `sauterTroisLignes()` du chapitre précédant :

- Avant de compiler le corps principal du programme (la fonction `main`), on compile le corps de la fonction.
- Ce code compilé est placé quelque part dans la mémoire, à une adresse qu'on appellera α (c'est-à-dire que α est l'adresse où se trouve la première instruction du code).
- Dans le code de la fonction, l'instruction **`return`** est compilée par RET. On ajoute également une instruction RET à la fin du code compilé de la fonction.

L'appel à cette fonction sera alors compilé vers le code GSB(α). Autrement dit $[\![\texttt{f ();}]\!]$ = GSB(α).

Si l'on prend le cas général d'une fonction sans argument ni résultat, elle est de la forme :

```
static void f() {
  p
  }
```

où p est une séquence, arbitrairement longue, d'instructions Java.

On va alors :

1. Placer à un emplacement mémoire adr(f) le code $[\![\texttt{p}]\!]$.

2. ajouter à la fin de ce code l'instruction RET.

3. Enfin, on peut compiler les appels à la fonction `f()` en GSB(adr(f)). Ou autrement dit : ⟦f()⟧ = GSB(adr(f)).

On voit que pour que cela fonctionne, il est essentiel qu'à la fin de l'exécution de ⟦p⟧, l'adresse de retour se retrouve en haut de la pile. En fait c'est une propriété du code compilé que pour toute séquence d'instructions Java p, l'exécution de ⟦p⟧ restitue la pile dans l'état dans lequel elle était avant l'exécution (à condition que p ne contienne pas d'instruction `return`).

4.3 Le passage des arguments

Considérons maintenant une fonction prenant trois arguments et ne rendant pas de résultat :

```
static void f(int e1, int e2, int e3){ ... }
```

Il nous faut maintenant comprendre comment passer des arguments entre différentes parties du code. Nous allons encore, pour cela, utiliser la pile. Le principe est simple : les valeurs des arguments sont placées sur la pile avant d'exécuter l'instruction GSB qui lance l'exécution du code de la fonction.

Prenons l'instruction `f(e1, e2, e3);` qui est donc un appel à la fonction `f`. Elle est compilée comme :

$$
\llbracket \texttt{f(e1, e2, e3)} \rrbracket \equiv \begin{cases} \llbracket \texttt{e1} \rrbracket & \text{; place la valeur du premier argument sur la pile} \\ \llbracket \texttt{e2} \rrbracket & \text{; place la valeur du deuxième argument sur la pile} \\ \llbracket \texttt{e3} \rrbracket & \text{; place la valeur du troisième argument sur la pile} \\ \mathsf{GSB}(\mathrm{adr}(\texttt{f})) & \text{; exécute le code de la fonction} \\ \mathsf{POP} & \text{; après l'exécution de la fonction, on} \\ \mathsf{POP} & \text{; efface les trois arguments de la pile} \\ \mathsf{POP} & \end{cases}
$$

On va voir maintenant comment, dans le code de la fonction, on peut accéder aux valeurs de ces arguments.

4.4 Le mécanisme de cadre

Le mécanisme de cadre va permettre au code de la fonction d'accéder aux arguments dans la pile. On voit que du point de vue du corps de la fonction, les arguments vont être stockés dans la pile juste en dessous de l'adresse de retour. Pour les "trouver", on va utiliser une fonctionnalité supplémentaire de la machine, le pointeur de cadre, souvent désigné par son appellation anglaise *frame pointer* ou FP.

4.4.1 Le pointeur de cadre (FP)

On a vu en 2.5 que la XVM possédait un registre SP *(stack pointer)* désignant l'adresse du haut de la pile. Elle possède un second registre semblable FP, pour *frame pointer*. Ce terme est parfois traduit par "pointeur de cadre", "pointeur de page" ou de manière hybride "pointeur de frame".

Le registre FP contient donc une adresse de la pile. En pratique, on aura toujours $\mathsf{FP} \leq \mathsf{SP}$ puisque SP pointe sur le haut de la pile [1] . L'instruction RFR, pour *Read in FRame*, permet de lire une valeur dans la pile correspondant à la position du pointeur de page. Plus exactement, RFR(i) lit la valeur de la pile se trouvant à l'adresse FP, avec un décalage (*offset*) de i. C'est-à-dire :

- RFR(0) va recopier en haut de la pile la valeur se trouvant à l'emplacement FP de la pile.
- RFR(1) va recopier en haut de la pile la valeur se trouvant à l'emplacement immédiatement au-dessus de FP.
- RFR(−1) va recopier en haut de la pile la valeur se trouvant immédiatement en-dessous de FP.
- Et ainsi de suite pour RFR(2), RFR(−2), RFR(3) ...

L'instruction duale de RFR(i) est WFR(i), pour *Write in FRame* qui va écrire la valeur en haut de la pile à l'emplacement FP avec le décalage i.

Notation On peut décrire cela de manière plus synthétique en notant $st(p)$ la valeur se trouvant à l'adresse p dans la pile. On peut dire alors :

- RFR(i) recopie et empile la valeur $st(\mathsf{FP} + i)$,
- WFR(i) déplace la valeur en haut de la pile en $st(\mathsf{FP} + i)$,
- FETCH(i) recopie et empile la valeur $st(\mathsf{SP} - i - 1)$.

4.4.2 Le comportement détaillé de GSB et RET

On peut maintenant préciser le comportement de GSB et RET.

Tout d'abord, GSB(i) va empiler la valeur de FP avant d'empiler celle de PC. De plus, elle va recopier la valeur de SP en FP avant d'effectuer le saut à l'instruction en i.

Du coup, on peut considérer que lorsqu'on exécute RET, l'adresse de retour se trouve en $st(\mathsf{FP})$.

Voici le comportement précis de ces deux instructions.

Instruction	avant l'instruction				après l'instruction			
	PC	SP	FP	pile	PC	SP	FP	pile
GSB(i)	n	s	f	$x; y; \ldots$	i	$s+2$	s	$n+1; f; x; y; \ldots$
RET	n	s	f	$\ldots$	$st(f+1)$	f	$st(f)$	$\ldots$

On voit que, dans le corps de la fonction, FP pointe vers l'endroit de la pile où se trouvent les arguments. Plus précisément, le dernier argument, dans notre exemple `e3` se trouve en $\mathsf{FP} - 1$. L'avant-dernier argument, dans notre exemple `e2`, se trouve en $\mathsf{FP} - 2$, et ainsi de suite jusqu'au premier argument.

On va donc compiler les appels aux arguments de la fonction par :

$$\begin{aligned} [\![\texttt{e1}]\!] &\equiv \mathsf{RFR}(-3) \\ [\![\texttt{e2}]\!] &\equiv \mathsf{RFR}(-2) \\ [\![\texttt{e3}]\!] &\equiv \mathsf{RFR}(-1) \end{aligned}$$

1. Plus exactement, on peut écrire du code machine pathologique qui aboutirait à une situation où l'on aurait plus $\mathsf{FP} \leq \mathsf{SP}$; mais le code généré par le compilateur va bien préserver cet invariant.

Exercice 4.4.1 Voyez-vous pourquoi, dans la pile, on range le premier argument en bas et le dernier en haut ? ◇

Solution page 185.

On remarque que si GSB recopie la valeur courante de FP dans la pile, c'est uniquement pour pouvoir la restaurer après l'exécution de la fonction.

4.5 Un autre exemple simple

Prenons l'exemple suivant suffisamment simple pour pouvoir être traité entièrement :

```
class Global {
  static int a;

  int ajouteVara (int b) { a = b + a; }

  public static void main(String[] args) {
     a = 3;
     ajouteVara(2);
     System.out.println(a);
   }
}
```

Cet exemple est choisi pour être sans malice : l'appel à la fonction augmente de 2 la valeur de `a` et le résultat affiché est 5.

Intéressons-nous d'abord à la compilation du corps de la fonction ; on voit qu'il y a dans le corps de `ajouteVara`, et en particulier dans l'expression `a + b`, deux variables de natures différentes : la première est globale, la seconde est un argument de fonction et donc locale. La variable globale `a` est stockée dans un emplacement choisi statiquement, comme décrit dans le chapitre 2. La valeur de l'argument `b` est stockée dans la pile.

Détaillons ce que donne la compilation du programme ci-dessus. Le code correspondant à la fonction `ajouteVara` sera placé à une certaine adresse mémoire que nous appellerons α.

L'instruction `a = 3;` est compilée comme décrit dans le chapitre précédent.

L'instruction `ajouteVara(2);` est compilé comme suit :

- 2 est placé sur la pile,
- GSB(α) exécute le code de la fonction,
- au terme de l'exécution de ce code, la valeur 2 est effacé de la pile par POP.

Le code de `main` sera donc :

```
PUSH(3)          ; la valeur 3...
PUSH(adr(a))
WRITE            ; est placée dans l'emplacement mémoire de a
PUSH(2)          ; l'argument de la fonction
GSB(α)           ; appel de la fonction
POP              ; on efface l'argument de la pile
```

Le code de la fonction, à l'adresse α, est la version compilée de l'instruction `a = b + a;` comme défini dans le chapitre 2, c'est-à-dire $[\![$a = b + a$]\!]$. La seule différence est la manière dont est traduit l'appel à la variable `b`. Ici, comme la valeur se trouve sur la pile, la valeur de `b` est obtenue par l'instruction RFR(−1) ; en effet, cette valeur se trouve dans la pile juste sous le pointeur de *frame*. Enfin, l'instruction RET termine le code. Au final, on aura donc à l'adresse α le code suivant :

```
(α)  RFR(−1)        ; on recopie l'argument en haut de la pile
     PUSH(adr(a))
     READ           ; on met la valeur de a en haut de la pile
     ADD            ; on calcule b + a
     PUSH(adr(a))
     WRITE          ; on affecte la nouvelle valeur de a
     RET
```

A retenir :

> *Les variables globales sont, comme on l'a souligné dans les chapitres précédents,* statiques, *c'est-à-dire que l'emplacement mémoire correspondant est choisi une fois pour toute au moment de la compilation.*
>
> *Dans le corps d'une fonction, on doit traiter également les arguments de la fonction. Ces arguments sont stockés dans la pile. Au moment de la compilation, on ne connait pas exactement cet emplacement. En revanche, on connait leur emplacement dans la* page mémoire *ou* frame*; c'est-à-dire que pour chaque argument, on connait son emplacement relativement au* frame pointer FP*. Ces variables sont donc statiques* dans la page *(frame).*
>
> *On verra dans les chapitres suivant des données qui seront véritablement non-statiques, ou* dynamiques*.*

4.6 Compilation du passage de résultat

Il nous reste à proposer une manière de compiler les fonctions qui rendent un résultat. Par exemple :

```
static int somme(int a, int b) { return(a + b); }
```

Une fois cette fonction définie, on peut utiliser `somme(2, 3)` comme une expression de type `int`. On se rappelle que le code compilé d'une expression doit placer en haut de la pile la valeur correspondante. Lorsqu'on a affaire à une fonction rendant un résultat, il faut donc :

- exécuter le corps de la fonction, comme dans les cas précédents,
- compiler l'instruction `return(...)` pour que le résultat puisse être placé en haut de la pile après le retour de l'appel.

Il y a différentes manières de procéder. Nous proposons ici une procédure relativement simple.

On utilise un registre supplémentaire de la machine abstraite. Ce registre est appelé R (comme "résultat") avec deux instructions correspondantes :

- PXR (*push* X *to* R) qui pousse le haut de la pile vers le registre R.
- PRX (*push* R *to* X) qui recopie le registre R et place sa valeur courante sur le haut de la pile.

Instruction	avant l'instruction		après l'instruction	
	pile	R	pile	R
PXR	$x; y; \ldots$	r	$y; \ldots$	x
PRX	$x; y; \ldots$	r	$r; x; y; \ldots$	r

On compile donc `return(e)` comme :

```
⟦e⟧    ; place la valeur correspondant à e sur la pile
PXR    pousse cette valeur dans R
RET    termine l'appel de la fonction
```

Par ailleurs, l'appel à une fonction comme `somme(a,b)` est compilé en :

```
⟦a⟧        ; place la valeur de a sur la pile
⟦b⟧        ; pareil pour b
GSB(α)     ; où α est l'adresse du code de double
POP        ; efface les valeurs des arguments de la pile
POP
PRX        ; place la valeur du résultat en haut de la pile
```

4.7 Fonctions récursives

Une caractéristique importante et puissante des fonctions d'un langage de programmation est la possibilité de faire appel à une fonction dans le corps de celle-ci, c'est-à-dire de définir des fonctions récursives.

En Java, il n'est pas nécessaire de signaler les définitions récursives (contrairement, par exemple à Caml avec sa construction `let rec`). Voici l'exemple classique, la version récursive de la fonction factorielle :

```
static int fact(int i) {
  if (i < 2) return(1);
  return(i * fact(i - 1));
}
```

Le schéma de compilation décrit dans la section précédente reste valable pour les fonctions récursives et n'a pas besoin d'être modifié ou étendu. A chaque appel récursif, le corps de la fonction ne "voit" dans la pile que la valeur de l'argument qui correspond à l'appel courant.

On voit sur un exemple comme la factorielle, qu'une fonction récursive va beaucoup solliciter la pile : la calcul de `fact(25)` donne lieu à un appel de `fact(24)`. Dans le calcul de `fact(24)` on a un appel à `fact(23)`, dans lequel on a un appel à `fact(22)` et ainsi de suite jusqu'à `fact(1)`. Donc la calcul de `fact(n)` donne lieu à `n` appels de fonction emboités. Et chacun de ses appel va, au moins, ajouter sur la pile :

- l'argument de la fonction,
- le pointeur FP,
- l'adresse de retour.

Donc le calcul de `fact(n)` va "utiliser" au moins $3 \times$ `n` niveaux dans la pile. C'est pourquoi, lorsqu'on appelle une fonction récursive avec un argument trop grand il peut arriver que l'espace mémoire réservé à la pile soit épuisé. On a alors une erreur de débordement de pile à l'exécution. Plus précisément, cette erreur se produit lorsque le compteur de pile SP dépasse la valeur maximale prévue.

On peut facilement observer une telle erreur avec une fonction qui devrait en principe donner lieu à une infinité d'appels récursifs emboités, comme :

```
static int foo(int n) {
  return(foo(n));
}
```

Solution page 185. **Exercice 4.7.1** Voici deux versions récursives, presque identiques, de la fonction factorielle. Décrivez dans chacun des deux cas l'état de la pile au moment où sa hauteur est maximale lorsqu'elle est la plus haute au cours de l'exécution de respectivement `fact1(4)` et `fact2(4)`. On peut appeler respectivement adr(`f`) et adr(`c`) les adresses du code de la fonction factorielle et celle depuis laquelle le premier appel est effectué.

```
static int fact1(int i) {
  if (i < 2) return(1);
  return(fact1(i - 1) * i);
 }

static int fact2(int i) {
  if (i < 2) return(1);
  return(i * fact2(i - 1));
 }
```

4.8 Variables locales

On peut déclarer des variables locales à l'intérieur du corps de fonctions. Par exemple dans cette fonction de calcul de puissance, les variables `i` et `r` sont locales :

```
static int power(int a, int n) {
  int i = 0;
  int r = 1;
  while (i <= n) {
      r = r * a;
      i++;
   }
  return(r);
}
```

On voit que les variables locales sont liées au code d'une fonction (ou une partie du code d'une fonction). Il est donc naturel ranger les valeurs de ces variables dans la pile. A la différence des arguments, elles vont être rangées au-dessus du pointeur de page.

On rappelle que :

- FP – 1 correspond au dernier argument de la fonction,
- FP pointe vers l'emplacement où se trouve la précédente valeur de FP,
- FP + 1 correspond à l'emplacement ou est stockée l'adresse de retour.
- Donc au début de l'exécution du corps de la fonction, SP vaut FP + 2.

La première variable locale, ici `i`, va donc être stockée en FP + 2, puis `r` en FP + 3 etc.

Le code compilé de `power` est donné en figure 4.2.

A retenir :

> *Tout comme les arguments des fonctions, les variable locales sont rangées dans la pile, et comme elles, leur emplacement dans la page courante est connu lors de la compilation.*

4.9 Les différents niveaux de la pile

On voit donc que la pile contient des éléments divers :

```
        PUSH(0)    ; on pousse la valeur initiale de i sur la pile
        PUSH(1)    ; on pousse la valeur initiale de r sur la pile
(α)     RFR(2)     ; la valeur courante de i
        RFR(−1)    ; la valeur courante de n
        LEQ        ; calcule i <= n
        GTZ(β)
        RFR(3)     ; la valeur courante de r
        RFR(−2)    ; la valeur de a
        MULT
        WFR(3)     ; la valeur de r est mise à jour
        RFR(2)     ; on reprend i
        PUSH(1)    ; pour l'incrémenter
        ADD
        WFR(2)
        GTO(α)     ; fin du corps de la boucle
(β)     RFR(3)
        PXR        ; la valeur de r est copiée en R
        RET        ; en restaurant la frame précédente, on efface aussi les variables locales
```

FIGURE 4.2 – Le code compilé de la fonction `power`.

- Des valeurs intermédiaires pour l'évaluation d'expressions.
- Les valeurs des arguments de fonctions.
- Les valeurs des variables locales.
- Les adresses de retour des appels de fonctions.
- Les valeurs des pointeurs de pages.

Pour illustrer et résumer cela, considérons la fonction suivante. Elle est volontairement aussi simple que possible, tout en utilisant variables locales et arguments :

```
static int ds (int a, int b) {
   int c = a + 1;
   int d = b + 1;
   return(1793 + c * d);
}
```

Supposons que cette fonction soit invoquée à travers un appel `ds(24, 43)`, et que cet appel ait lieu à l'adresse $\mathcal{R}$ du programme.

La figure 4.3 montre l'état de la pile correspondant au moment où va être calculé la multiplication. On voit qu'on a, de haut en bas :

1. Les valeurs 1793, 25 et 44 qui ont été empilées pour permettre de calculer la valeur de `1793 + c * d`.
2. Les valeurs des variables locales `c` et `d`.
3. L'adresse de retour $\mathcal{R} + 1$.
4. La valeur de FP au moment de l'appel.
5. Les valeurs des arguments de la fonction.
6. En dessous on trouve ce qu'il y avait sur la pile au moment de l'appel de fonction.

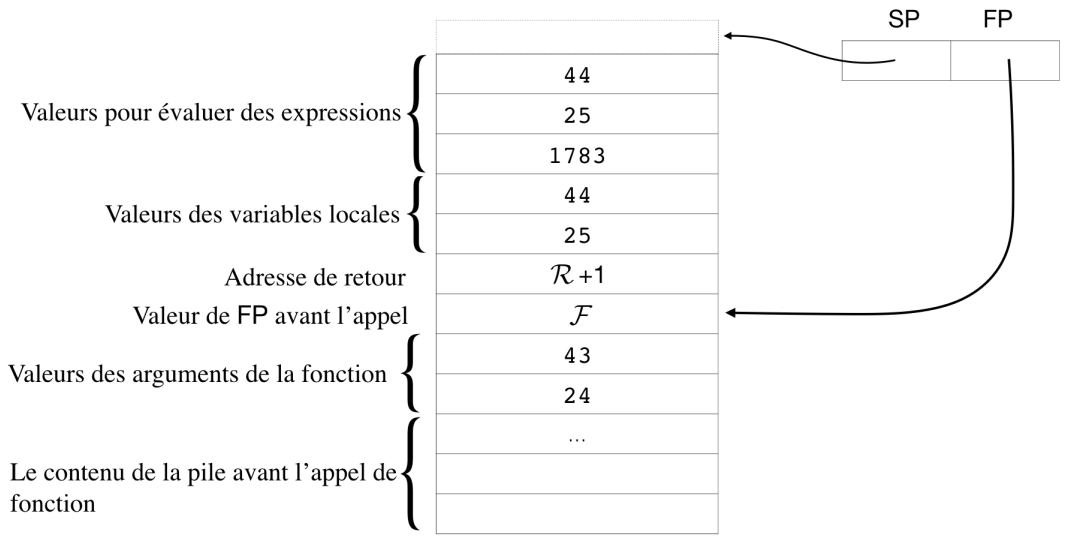


FIGURE 4.3 – Un état de la pile

On indique également les positions vers lesquelles pointent SP et FP : pour le premier c'est, par définition, le haut de la pile. Pour le second, c'est la valeur de SP après avoir empilé les arguements au moment de l'appel de fonction.

On voit donc que les valeurs correspondants aux cas 2 et 5 ci-dessus sont accessibles grace au pointeur FP. Elles correspondent donc à la page locale (ou cadre local) de la machine.

4.10 Une optimisation de compilation : les appels terminaux

Cette partie est un peu technique et il est raisonnable de ne pas l'aborder en première lecture. Elle peut toutefois plaire à ceux qui apprécient se pencher sur le code machine. EIle donne aussi une idée d'une technique plus générale, importante dans les compilateurs évolués, à savoir les optimisations de programme. En général, une optimisation est une transformation effectuée par le compilateur sur le code machine produit, pour le rendre plus efficace. On donne d'autres exemples d'optimisation à la compilation avec l'exercice 2.4.1 puis dans la section 8.8.

L'optimisation des appels de fonction terminaux est un exemple classique d'optimisation de code. Considérons une fonction `g(int n)`. Dans le corps de cette fonction, on trouvera donc un ou plusieurs appels de forme `f(e)`, à une fonction `f` qui peut, ou pas, être identique à `g`. Un appel de fonction est dit *terminal* si il est de la forme `return(f(e))`, c'est-à-dire que le résultat de la fonction `g` est aussi le résultat de l'appel à la fonction `f`.

Par exemple, la définition habituelle de la fonction factorielle n'est pas terminale :

```
static int fact(int i) {
  if (i < 2) return(1);
  return(i * (fact(i - 1)));
}
```

En revanche, l'unique appel récursif de la fonction suivante est terminal :

```
static int aux(int i, int r) {
  if (i < 2) return(r);
```

```
  return(aux(i - 1, r * i));
}
```

On remarque qu'on peut obtient bien la factorielle de `i` en calculant `aux(i, 1)`. La seconde fonction est donc une version terminale de la factorielle.

L'intérêt de repérer les appels terminaux est qu'il est possible d'optimiser leur traitement à la compilation. Considérons un tel appel terminal `return(f(e))`. Une fois calculée la valeur de `e`, on n'a plus besoin de connaître la valeur de l'argument `n`. Le principe de l'optimisation est le suivant ; on compile `return(f(e))` par un code qui :

- Calcule `e`,
- place la valeur dans la pile à l'emplacement qui contenait la valeur de l'argument (en fait FP − 1),
- on appelle le code de `f` par un saut simple GTO(adr(`f`)) à la place du GSB des appels habituels.

Voici les deux versions de la compilation de `return(f(e))` avec optimisation de la récursion terminale (à droite) et sans (à gauche) :

⟦`e`⟧		⟦`e`⟧	; l'argument est mis sur la pile
GSB(adr(`f`))	; appel récursif	WFR(−1)	;puis placé au bon endroit sur la pile
POP	; argument effacé	GTO(adr(`f`))	
RET	; fin de la fonction		

On a donc, essentiellement, remplacé une séquence GSB RET par un GTO. L'avantage est d'une part que les opérations GSB et RET peuvent prendre du temps, car elles sont complexes. Mais surtout, on évite de faire grandir la pile.

Remarque 4.10.1 Notez que cette description de l'optimisation est pour le cas où il n'y a pas de variables locales dans la fonction appelante. Dans le cas général, il faut repérer combien de variables locales sont stockées en haut de la pile à l'endroit correspondant à l'appel, et les effacer en insérant le même nombre de POP avant le GTO. Sans cela, les variables locales de la fonction appelée seraient décalées. ◇

On peut vérifier si un compilateur effectue l'optimisation de la récursion terminale en regardant le comportement d'un programme effectuant une boucle infinie d'appels récursifs, comme :

```
static foo (int n) { return(foo(n)); }
...
   System.out.println(foo(3));
```

Vous pouvez faire l'essai et constater que Java n'effectue pas cette optimisation ; en effet le programme échoue avec `StackOverflowError`. En revanche, le programme similaire en Caml va boucler indéfiniment :

```
let rec foo n = foo n;;
foo 3;;
```

Dans le cas de C, on peut donner au compilateur des options qui lui demanderont d'effectuer ou pas certaines optimisations comme la récursion terminale. Le comportement par défaut sera en général d'effectuer l'optimisation.

Nous nous sommes ici restreint à la description de l'optimisation pour une fonction à un argument, mais on peut généraliser à des fonctions prenant plus d'arguments (comme la version terminale de la factorielle ci-dessus).

On peut aussi remarquer que si cette optimisation est possible pour tous les appels de fonction, même non-récursifs, pourvu qu'ils soient terminaux, c'est-à-dire de la forme[2] `return(f(..));` l'optimisation est effectivement particulièrement utile pour des fonctions récursives ou mutuellement récursives. On peut alors espérer avoir une auteur de pile bornée, la où la version non-optimisée de la fonction récursive utilisera une hauteur de pile qui augmente avec la valeur de l'argument. C'est pourquoi on parle parfois de "dé-récursivation" pour décrire cette optimisation.

2. Au moins si la fonction appelante et la fonction appelée ont le même nombre d'arguments ; on laisse le lecteur intéressé réfléchir à ce point.

Chapitre 5

Les enregistrements et les méthodes

Jusqu'à maintenant, nous n'avons pas traité les aspects orientés objet de Java. Plus exactement, les seuls "objets", au sens conventionnel, que nous avons manipulés, sont des nombres et des booléens, éventuellement regroupés dans des tableaux. L'étape suivante est de regrouper de tels objets primitifs dans des structures un peu plus complexes. Dans un premier temps nous présentons les aspects simples de ces entités que nous n'appellons pas encore "objets". Il faut toutefois souligner que l'introduction des ces structures est important car :

- Elles sont les premières à faire appel à une gestion dynamique de la mémoire.
- En munissant, dans les chapitres suivant, ces structures de *méthodes*, nous arriverons à la programmation orientée-objet proprement dite.

5.1 Les champs d'un enregistrement

Imaginons que nous écrivions un logiciel qui gère un parc automobile. Dans ce logiciel, un véhicule est décrit par un certain nombre de caractéristiques : marque, immatriculation, année de fabrication. Cela peut correspondre à deux chaînes de caractères (marque, immatriculation) et un nombre entier (année de fabrication). On peut regrouper ces trois données en créant une classe spécifique `Vehicule` :

```
class Vehicule {
// les variables statiques sont globales
  static int anneeCourante = 2021;

// les champs propres à chaque instance de la classe
  String marque;
  String immatriculation;
  int anneeFabrication;

// enfin le constructeur
  Vehicule(String m, String i, int f) {
      marque = m;
      immatriculation = i;
      anneeFabrication = f;
   }
```

Ce code définit la classe `Vehicule`. Cette classe va jouer un rôle différent de ce que nous avons vu jusqu'ici. La nouveauté est que les variables `marque`, `immatriculation` et `anneeFabrication`

ne sont pas munies du mot-clé `static` ; on ne parle plus du coup de variable mais de *champs* de la classe. Du coup :

- La variable statique `anneeCourante` est une variable statique comme on l'a décrit dans les chapitres précédents. Elle occupe un (et un seul) emplacement statique dans la mémoire.
- En revanche, on peut créer plusieurs *instances* de la classe `Vehicule`, qui correspondront chacune à un véhicule particulier. C'est-à-dire que chaque instance sera composée par un triplet `String`×`String`×`int`.

On crée des instances de la classe en utilisant le *constructeur*. Dans le code ci-dessus, il prend en argument, dans l'ordre, la marque, l'immatriculation et l'année de fabrication.

Voici un programme simple qui crée et manipule de telles instances :

```
Vehicule maDodoche;
Vehicule voitureDirecteur;

  maDodoche = new Vehicule("2cv", "99-DODO-75", 1966);
  voitureDirecteur = new Vehicule("Delahaye", "69-GV-92", 1936);

  System.out.println("Ma voiture est une "+ maDodoche.marque);
  System.out.println("Elle a "
                       + (anneeCourante - maDodoche.anneeFabrication)
                       + " ans");

  if (maDodoche.anneeFabrication > voitureDirecteur.anneeFabrication)
       System.out.println("la voiture du directeur est plus ancienne");
```

On voit que les champs de chaque instance se comportent comme des variables des types attendus. Il est en général possible de modifier la valeur d'un champ :

```
  maDodoche.immatriculation = "X-2021-TOP";
```

Remarquons enfin qu'il est possible de surcharger le constructeur. Par exemple, si un grand nombre de véhicules traités sont de la marque "Chaika" [1], on pourra ajouter un second constructeur comme :

```
Vehicule( String i, int f) {
    marque = "Studebaker";
    immatriculation = i;
    anneeFabrication = f;
 }
```

Dans ce cas, les deux lignes ci-dessous seront strictement équivalentes :

```
v = new Vehicule("Studebaker", "Indiana-1", 1954);

v = new Vehicule("Indiana-1", 1954);
```

5.2 Fonctions et méthodes

Supposons que nous voulions définir une fonction qui calcule l'âge du véhicule par rapport à la variable statique, donc globale, `anneeCourante`. Il y a deux possibilités. La première relève des constructions déjà présentées dans les chapitres précédents : nous définissons une fonction statique `age` qui prend en argument le véhicule :

1. Je n'utilise que des marques qui n'existent plus depuis longtemps. Delahaye était un fabriquant français de voitures de luxe en particulier dans les années 1930, Studebaker une ancienne marque américaine, etc. . .

```
static int age(Vehicule v) {
    return(anneeCourante - v.anneeFabrication);
}
```

On appelle alors cette fonction avec, par exemple, `age(maDodoche)`.

Mais il est également possible de définir age comme une *méthode*. Pour cela, il faut ôter le mot-clé **`static`** :

```
int age() {
    return(anneeCourante - this.anneeFabrication);
}
```

On appelle alors cette méthode pour un objet, par exemple, avec `maDodoche.age()` ou `voitureDirecteur.age()`.

Le principe de ces *méthodes dynamiques* est le suivant : une méthode qui n'est pas déclarée comme statique prend un argument caché appelé **`this`**. Cet argument est une instance de la classe de cette méthode (ici `Vehicule`) ; il est donc important de déclarer cette méthode à l'intérieur de la classe `Vehicule`. On *invoque* cette méthode par la syntaxe `v.age()` où `v` est un `Vehicule`. Alors est exécuté le code de la méthode, dans lequel l'argument caché **`this`** est lié à `v`.

Dans l'exemple ci-dessus, l'exécution des deux solutions présentées, fonction statique et méthode dynamique, sont donc parfaitement équivalentes. En revanche, dans le cas des méthodes, la syntaxe donne l'impression que le code calculant l'âge est "porté" par l'objet véhicule. C'est le premier trait orienté-objet de Java que nous rencontrons.

Remarquons que **`this`** peut souvent être omis dans la syntaxe. Si, dans une méthode, on utilise un nom de champ seul, le compilateur comprend qu'il s'agit du champ de l'objet **`this`**. En d'autres termes, la définition suivante de la méthode `age()` est strictement équivalente à la précédente :

```
int age() {
    return(anneeCourante - anneeFabrication);
}
```

5.3 Public et privé

Un premier intérêt de cette syntaxe orientée-objet des méthodes apparaît quand on utilise une autre possibilité, qui est de rendre privés des champs des objets.

Supposons par exemple que l'on ne veuille pas rendre publique l'immatriculation des véhicules. On peut pour cela déclarer ce champ *privé* :

```
class Vehicule {

  String marque;
  private String immatriculation;
  int anneeFabrication;
```

Mais on peut vouloir vérifier si un véhicule possède bien telle immatriculation. Pour que ce soit possible, on ajoutera à la classe la méthode suivante :

```
boolean verifImmat(String i) {
    return(this.immatriculation.equals(i));
}
```

(La méthode `equals` sert simplement à vérifier l'égalité de deux `String`; cette question est traitée dans la section 6.8.1.)

On pourra alors par exemple faire `v.verifImmat("K-67-J")` pour vérifier que la voiture `v` est immatriculée K-67-J. On arrive ainsi à ne révéler qu'une information partielle sur l'immatriculation.

Remarque 5.3.1 Il n'est pas nécessaire de rentrer dans ces détails pour l'instant, mais si l'on est tout à fait précis, un champ n'est réellement public que lorsqu'il est déclaré avec le mot-clé `public`. Par défaut, la visibilité est restreinte au *package* de la classe. Pour le détail des différents niveaux de visibilité, voir 9.1.5. ◇

5.4 L'encapsulation

Les méthodes de la classe peuvent également être publiques ou privées, c'est-à-dire accessibles depuis l'extérieur de la classe ou non. Ceci permet ce qu'on appelle l'encapsulation.

En particulier, il y a une forme de modularité. L'auteur de la classe choisit quels champs et quelles classes sont visibles de l'extérieur (donc depuis d'autres classes). On peut donc ensuite changer l'implémentation interne de ces méthodes et la manière dont l'information est structurée à l'interieur de la classe, et rester compatible avec le reste du code.

Ce qui est visible de l'extérieur, c'est-à-dire les champs et les méthodes publics, forme *l'interface* de la classe.

Dans la syntaxe Java, si on a un objet `v` instance de, par exemple, la classe `Vehicule`, on accède donc aux composantes du module, en l'occurrence de la classe, à travers `v` : `v.marque`, `v.age()`. La syntaxe est aussi uniforme que possible entre champs et méthodes. Le fait que l'on accède à ces composants à travers un objet relève de l'aspect *orienté-objet* du langage.

5.5 Désigner les composantes statiques de la classe

On a vu qu'il peut y avoir des composantes statiques aux classes.

Des variables statiques comme dans l'exemple précédent, la variable `anneeCourante`. Ce sont donc des variables *globales* de la classe; il y a une seule copie de cette variable, partagée par toutes les instances de la classe (tous les `Vehicule`).

Des méthodes statiques qui sont des fonctions qui ne se rapportent pas à une instance particulière de la classe. On pourrait ainsi, dans l'exemple précédent, définir la méthode statique suivante :

```
public static void nouvelAn() {
       anneeCourante++;
}
```

Lorsqu'elles sont publiques, ces composantes statiques peuvent être adressées depuis l'extérieur de la classe. On utilise pour cela encore la syntaxe pointée, cette fois avec le nom de la classe : `Vehicule.anneeCourante` ou `Vehicule.nouvelAn()`.

On peut aussi atteindre ces composantes, en utilisant une instance de la classe au lieu du nom de la classe elle-même : `v.anneeCourante` ou `v.nouvelAn()` sont strictement équivalentes aux expressions précédentes.

5.6 Comportement en mémoire : le tas

Il est important de comprendre comment les objets sont représentés en mémoire. Jusqu'ici, nous avons vu que l'espace d'exécution de Java comporte au moins trois zones de mémoire :

- la pile,
- la zone où est stocké le code exécutable,
- l'espace où se trouve les valeurs des variables statiques.

L'information afférente aux objets est elle conservée dans un quatrième espace mémoire : le *tas* ou *heap*. Cette zone a la particularité d'être gérée dynamiquement. C'est-à-dire que les portions du tas sont attribuées au cours de l'exécution du programme, et non pas lors de la compilation.

Nous commençons par un exemple aussi simple que possible. Soit la classe suivante, qui correspond à des points de coordonnées entières :

```
class Point {
  int x;
  int y;
  Point(int a, int b) { x = a; y = b; }
}
```

Ajoutons maintenant à cette classe le code suivant (statique) :

```
  static Point p;
  static int a = 4;

  public static void main(String[] args) {
    p = new Point (3,a);
    p.x = 5;
    p = new Point (6,6);
  }
```

Le premier point important est qu'une variable de type `Point` comme `p` contient en fait un *pointeur*, c'est-à-dire une adresse mémoire. C'est à cette adresse que l'on trouvera les valeurs des champs de `p`.

A retenir

> *Du point de vue opérationnel, la valeur d'un objet est une adresse. À cette adresse on trouvera d'autres valeurs. C'est la différence essentielle entre classes et types primitifs :*
>
> - *Une variable du type primitif* `int` *contiendra le nombre entier lui-même.*
> - *Une variable dont le type est une classe comme* `Point` *contiendra une adresse.*
>
> *Ce point reviendra régulièrement, par exemple lorsque l'on verra comment tester l'égalité de deux objets, puis des traits orientés-objet plus avancés.*

La figure 5.1 illustre ce qui se passe :

1. La variable `p` est statique. Lors de la compilation on lui assigne donc un emplacement mémoire.
2. La variable `p` n'est pas initialisée. Elle a donc la valeur par défaut `null`.

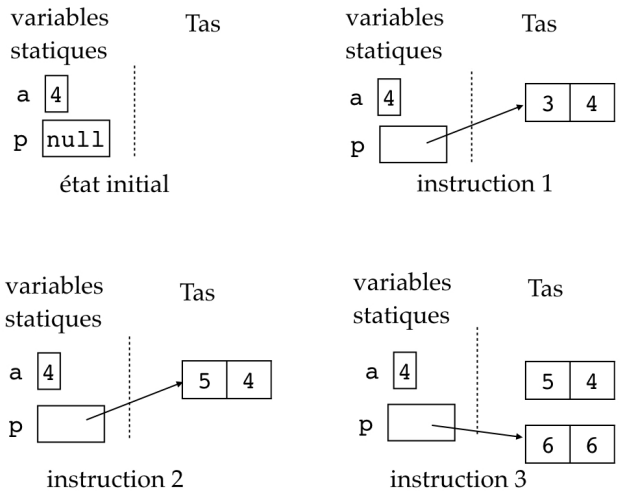


FIGURE 5.1 – Les quatre étapes de l'exécution

3. L'appel au constructeur de `Point` dans la première instruction de `main` va effectuer les opérations suivantes :
   - trouver une adresse mémoire libre dans le tas, on remarque que cette adresse est trouvée *dynamiquement*, au cours de l'exécution,
   - placer 3 et la valeur de `a` (c'est-à-dire 4), respectivement à cette adresse mémoire et à la suivante,
   - placer cette adresse mémoire dans la variable `p`.
4. La seconde instruction va modifier la valeur de la case mémoire correspondant à `p.x` dans le tas (donc remplacer 3 par 5).
5. La troisième instruction va effectuer une nouvelle allocation dans le tas, c'est-à-dire trouver une autre adresse, y placer les valeurs 6 et 6, et changer la valeur de `p`.

L'intérêt du tas comme espace de mémoire dynamique apparaitra d'une part avec les structures de données récursives (section 5.9), et ensuite avec l'héritage (chapitre 6). *Pour l'instant, retenons que le mot clé* **`new`** *combiné au constructeur est bien une* instruction *du langage de programmation, c'est-à-dire qu'il correspond à une* action *qui a lieu pendant l'exécution du programme, et non pas lors de la compilation.*

5.7 Le GC

L'abréviation GC vient de l'anglais *garbage collector*. L'abréviation est tellement utilisée qu'on traduit l'original parfois par glaneur de cellules. Il s'agit d'un mécanisme très important de l'environnement d'exécution des programmes Java, qui va repérer les parties (cellules) du tas qui ne sont plus utilisées et les rendre à nouveau disponibles.

Pour prendre à nouveau un exemple très simple :

```
static Vehicule v;
...
v = new Vehicule("Panhard", "666 WX 75", 1964);
v = new Vehicule("Tatra", "400 OM 13", 1958);
```

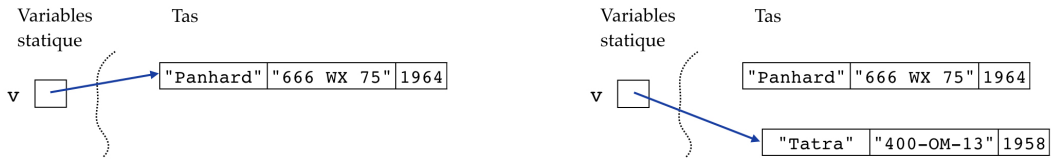


FIGURE 5.2 – Etats de la mémoire avant et après la ré-affectation de la variable `v`.

Les états de la mémoire après la première puis la seconde instruction sont représentés dans la figure 5.2. On y voit qu'au terme de l'exécution, la situation est la suivante : `v` pointe vers les cellules mémoires du tas décrivant la Tatra. En revanche, il n'y a aucun pointeur vers les cellules mémoires décrivant la Panhard. Cela veut dire que le contenu de ces cellules est devenu inutile : on ne retrouvera jamais leur adresse, donc on ne pourra plus ni les lire ni les modifier. On peut donc *libérer* cet espace mémoire, c'est-à-dire le rendre disponible pour de nouvelles allocations à venir. Mentalement on peut considérer qu'il s'agit donc d'effacer des informations devenues inutiles.

Ce travail consistant à repérer et libérer les cellules mémoires devenues inutiles dans le tas est effectué automatiquement par un programme appelé *garbage collector* ou GC. Ce programme fait partie de l'environnement d'exécution de Java, c'est-à-dire qu'il effectue son travail pendant que la JVM interprète le byte-code Java.

Plus précisément, le GC fait partie de la JVM, qui doit donc *partager* son temps entre l'interprétation du byte-code et des cycles de nettoyage de la mémoire.

Quelques points au sujet du GC :

- Java est loin d'être le seul langage à disposer d'un GC ; c'est par exemple aussi le cas de Caml, Python, LISP, Scheme, JavaScript. . . En revanche, C ou C++ n'en n'ont pas. C'est même l'une des différences importantes de ce dernier langage avec Java. Dans un langage sans GC, il faut libérer explicitement les espaces mémoire qui ne sont plus utilisés à l'aide d'une instruction dédiée.

- Les implémentations les plus simples du GC consistent à suspendre l'exécution du programme lorsqu'il n'y a plus d'espace mémoire disponible, de parcourir le tas pour rechercher et libérer les cellules orphelines puis de reprendre l'exécution du programme. On peut alors parfois observer visuellement l'arrêt temporaire du programme. Les techniques modernes de GC permettent de limiter ce défaut, en effectuant des phases de nettoyage très rapides, essentiellement en se concentrant d'abord sur les cellules allouées récemment. A noter que le GC de Ocaml est à ce titre souvent plus performant que celui des implémentations courantes de Java.

- Il n'est pas si facile de repérer quelles cellules sont orphelines et peuvent être effacées ou libérées par le CG. En particulier lorsqu'on a des types ou des classes récursives, des cellules du tas peuvent pointer vers d'autres cellules du tas. C'est-à-dire que le GC doit en fait effectuer un parcours de graphe dans la mémoire.

- Une conséquence du point précédant est que, dans certains cas, le GC peut considérer qu'une cellule est encore "vivante" alors que le programmeur pense qu'elle devrait être libérée par le GC. Dans ce cas, le programme va utiliser plus de mémoire que prévu. Dans des cas extrêmes, cela peut même conduire à l'échec du programme par manque de mémoire. Ce type de bug souvent particulièrement difficile à détecter et à corriger, est appelé une *fuite de mémoire*, ou *memory leak* en anglais.

5.8 Types énumérés

Bien qu'ils ne soient pas des enregistrements, nous présentons ici les types énumérés de Java, car ils sont surtout utiles en combinaison avec les enregistrements. Les types énumérés sont des constructions très simples ; ce sont des types ne pouvant prendre qu'un nombre fini de valeurs pré-déclarées. Par exemple, on peut vouloir définir un type pour les couleurs de nos véhicules :

```
enum Color {
    ROUGE, BLANC, NOIR, GRIS
    }
```

On aura souvent intérêt à définir ces types énumérés à l'intérieur de la classe où ils sont utilisés, par exemple :

```
class Vehicule {
  String marque;
  String immatriculation;
  int poids;
  Color couleur;

  public enum Color {
      ROUGE, BLANC, NOIR, GRIS
      }

  Vehicule(String m, String i, int p, Color c) {
      marque = m;
      immatriculation = i;
      poids = p;
      couleur = c;
   }
}
```

On peut effectuer des tests d'égalité == sur les éléments d'un type énuméré. Mais les deux constructions favorites sont d'une part la possibilité d'énumérer les éléments en utilisant la méthode `values()`, par exemple dans une boucle **`for`** et d'autre part le choix par **`switch`** ; par exemple :

```
enum Color {
    ROUGE, BLANC, NOIR, GRIS ;
    public static void main(String[] args) {
        for (Color c : Color.values())
            switch (c) {
            case BLANC : System.out.print("blanc ");
                break;
            case ROUGE : System.out.print("rouge ");
                break;
            case NOIR : System.out.print("noir ");
                break;
            default : System.out.print("autre");
            }
        System.out.println();
    }
}
```

va afficher : `rouge blanc noir autre`.

Il faut remarquer que :

– l'ordre d'énumération est celui des constructeurs fixé au début,

– on voit que un type `enum` est en fait une classe (on peut le munir d'une fonction `main`),

– la construction **`switch / case`** n'est en fait pas restreinte aux types énumérés (on peut l'utiliser pour des nombres par exemples),

– ce type de boucle **`for`** est également une instance d'un cas plus général que les types énumérés, on le comprendra au chapitre 6.

5.9 Listes et autres classes récursives

5.9.1 Listes

Les listes sont des structures courantes en informatique ; une de leurs caractéristiques premières est qu'elles peuvent être de taille variable. Si l'on considère, dans un premier temps, des listes de nombres entiers, on peut avoir : [] (la liste vide), [3; 8] (de longueur 2), [5; 0; 9; 7] (de longueur 4) etc. . .

On peut définir mathématiquement l'ensemble des listes de nombres entiers en posant que c'est le plus petit ensemble vérifiant les deux clauses suivantes :

– la liste vide est une liste,

– si l est une liste et n un nombre entier, alors on peut aussi construire la liste dont le premier élément est n et les éléments suivants sont ceux de l. Dans le langage Caml cette liste est notée $n :: l$.

On peut facilement transposer cette définition mathématique en Java en utilisant deux techniques :

– Prendre la convention que le pointeur **`null`** représente la liste vide,

– Utiliser un champ de type `Liste` dans la classe `Liste` (ce qui la rend de fait récursive).

Remarquez que nous choisissons l'orthographe française `Liste` car le nom `List` est déjà utilisé dans les bibliothèques Java. Cela donne la définition de classe suivante :

```
class Liste {
  int head;
  Liste tail;
  Liste(int i, Liste l) {
      head = i;
      tail = l;
   }
}
```

Construisons alors, par exemple, la liste correspondant à [3; 2; 1] :

```
Liste l = new Liste(3, new Liste(2, new Liste(1, null)));
```

On voit que cela correspond au schéma mémoire suivant :

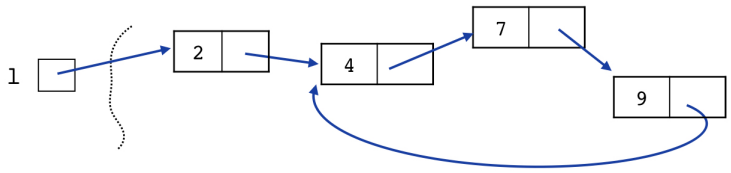
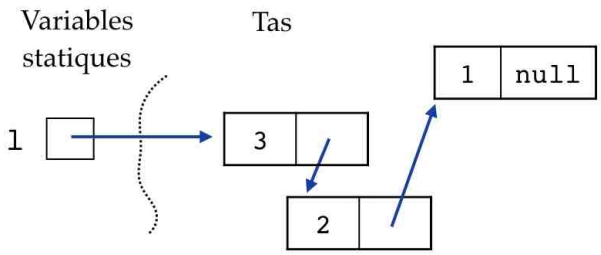


FIGURE 5.3 – Une liste “infinie”



Sur de telles structures, on a souvent le choix entre un style de programmation récursif et un style itératif. Voici deux versions de la fonction calculant la longueur d'une `Liste` :

```
static int lengthIter(Liste l) {
  int r = 0;
  while (l != null) {
      r++;
      l = l.tail;
  }
  return(r);
}

static int lengthRec(Liste l) {
  if (l == null) return(0);
  return(1 + lengthRec(l.tail));
}
```

De manière générale, il est plus *robuste* d'utiliser une version itérative, car une fonction récursive peut causer un débordement de pile lorsqu'elle est appliquée à une structure, ici une liste, trop grande. Ces erreurs d'exécution ne sont pas visible lors d'une première phase de test où l'on utiliserait que des exemples de petite taille.

Solution page 186. **Exercice 5.9.1** Définissez de la même manière deux versions de la fonction `somme` qui calculent la valeur de la somme des éléments d'une `Liste`. ◇

S'il est important de comprendre ce type de classes, il faut noter qu'il ne permet pas l'utilisation de méthodes dynamiques. On revient sur ce point dans la section 5.12

5.9.2 Cycles et “listes infinies”

Il est possible de créer des cycles dans les éléments de la classe décrite ci-dessus. Par exemple la séquence suivante va créer l'état illustré par la figure 5.3 :

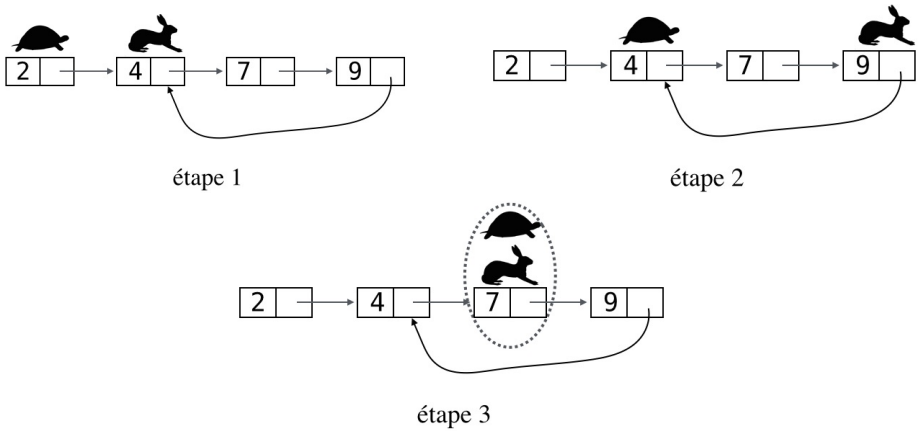


FIGURE 5.4 – Les étapes de l'algorithme du lièvre et de la tortue sur un exemple simple.

```
l = new Liste(2, new Liste(4, new Liste(7, new Liste(9,null));));
l.tail.tail.tail.tail = l.tail;
```

On alors créé un cycle dans l'objet. Une telle structure peut être vue comme une représentation de la liste, ou séquence, infinie $2;4;7;9;4;7;9;4;7;\ldots$

Suivant les situations, on peut choisir de s'autoriser de tels cycles. Nottons que :

- Si on s'autorise des cycles, il faut prendre garde à ce que des fonctions comme celles ci-dessus, calculant la longueur ou la somme des éléments d'une liste peuvent boucler (versions itératives) ou échouer sur des débordements de pile (version récursives).
- Si on rend le champ `tail` final, il n'est plus possible de créer de cycles. Vérifier précisément cet invariant de la structure n'est pas trivial.

Il existe un joli petit algorithme, dit "du lièvre et de la tortue", pour vérifier si une liste contient un cycle ou pas :

- On place deux pointeurs qui vont parcourir la liste. Le pointeur `hare` (lièvre en anglais) qui va avancer vite, et le pointeur `tortoise` (tortue) qui va avancer lentement. La tortue part du premier élément de la liste, le lièvre du deuxième élément.
- A chaque tour, `hare` avance de deux éléments dans la liste et `tortoise` avance d'un élément.
- On continue à faire avancer ces deux pointeurs jusqu'à ce que l'un des deux évènements suivants se produise :
  - on tombe sur un élément `null` ; dans ce cas la liste ne comporte pas de cycle.
  - `hare` et `tortoise` se trouvent sur le même élément ; dans ce cas la liste comporte un cycle.

La figure 5.4 illustre le déroulement de l'algorithme dans le cas de la liste avec le cycle de l'exemple précédent.

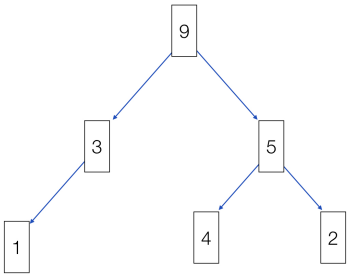
Exercice 5.9.2 Prouvez que cet algorithme termine. Quelle est sa complexité en temps ? Solution page 186.

Prouvez sa correction. ◇

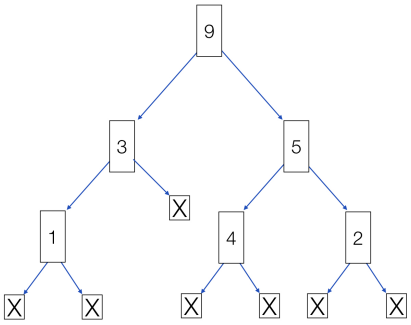
Exercice 5.9.3 Écrivez une fonction `boolean cycleP(Liste l)` qui implémente cet algorithme en rendant `true` si `l` comporte un cycle et `false` sinon. Solution page 187. ◇

5.10 Arbres binaires

La technique utilisée pour représenter les listes se généralise à des structures récursives plus complexes, par exemple les arbres binaires. Considérons des arbres binaires dont les nœuds portent des nombres entiers :



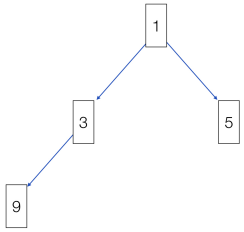
ou, si l'on affiche le même arbre avec les feuilles vides :



On peut définir la structure de données correspondante en réutilisant quasiment la classe des listes définies ci-dessus ; la seule différence étant qu'au lieu d'une seule sous-liste, on a deux champs pour les deux sous-arbres, gauche et droit.

```
class ArbreBin {
  int value;
  ArbreBin left;
  ArbreBin right;
  ArbreBin (int i, ArbreBin l, ArbreBin r) {
      value = i;
      left = l;
      right = r;
   }
}
```

Solution page 187. **Exercice 5.10.1** Donnez l'expression Java qui correspond à la construction, dans la classe `ArbreBin`, de l'arbre binaire suivant :



◇

Exercice 5.10.2 On dit qu'un arbre binaire est un *tas* si pour tout chemin depuis la racine jusqu'à une feuille, on rencontre des valeurs croissantes. Par exemple, l'arbre de l'exercice précédent est un tas car $1 \leq 3 \leq 9$ et $1 \leq 5$. Remarquez bien que malgré l'homonimie, cela n'a rien à voir avec l'espace mémoire appelé tas. Solution page 187.

Ecrivez une fonction qui vérifie si un arbre binaire est un tas :

```
static boolean isHeap(ArbreBin t) { ... }
```

Remarquez qu'il serait plus difficile ici d'écrire une version itérative. Vous pouvez donc vous autoriser la récursivité. ◇

On peut facilement modifier cette définition pour des arbres ternaires, quaternaires... On peut utiliser des listes dans la définition si l'on veut avoir une arité variable, c'est-à-dire un nombre arbitraire de sous-arbres à chaque nœud ; on donne un exemple plus bas.

Enfin, dans certains cas, on peut choisir une représentation très différentes pour les structures arborescentes, qui utilise l'héritage et les classes abstraites, notions présentées dans le chapitre 6. On utilisera une telle structure pour les arbres de syntaxe dans le chapitre 8.

5.11 Classes génériques

La classe `Liste` précédente définit des listes de nombres entiers `int`. Si l'on veut manipuler des listes de chaînes de caractères, il faudrait redéfinir un type similaire en changeant le type du champ `head`, et trouver un nouveau nom pour cette nouvelle classe (par exemple `ListString`.

Cela serait évidemment fastidieux. Java propose un mécanisme permettant de paramétrer une définition de classe par une (ou plusieurs) autre(s) classe(s). Cela s'appelle les *classes génériques*. Voici la définition de `Liste` comme une classe générique :

```
class Liste<E> {
  E head;
  Liste tail;
  Liste(E i, Liste l) {
      head = i;
      tail = l;
   }
}
```

Dans cette définition, `E` désigne une classe quelconque. On peut instancier ce paramètre, c'est-à-dire, par exemple, que `Liste<String>` est une classe valide.

On verra (chapitre 6) que ce mécanisme peut être raffiné grace à l'héritage.

Exercice 5.11.1 Utilisez cette classe générique pour définir une classe `Tree<E>` d'arbres dont : Solution page 187.

– Les nœuds contiennent un élément de classe `E`,

– les feuilles sont vides,

– chaque nœud peut avoir un nombre arbitraire de sous-arbres. ◇

5.12 Classes enveloppées ou containers

5.12.1 Types primitifs enveloppés

Dans l'exemple donné ci-dessus de la classe générique `Liste<E>`, on ne peut pas fabriquer la classe `Liste<int>`. La raison en est que **`int`** n'est pas une classe mais un type primitif [2]. Pour remédier à cela, Java fournit un certain nombre de *classes enveloppées* correspondants aux types primitifs. Par exemple la classe `Integer` dont les instances contiennent exactement un entier.

On peut donc obtenir des listes d'entiers avec la classe `Liste<Integer>`.

Dans le cas de ces classes enveloppées fournies par Java, on peut utiliser de manière quasi-indifférenciée des objets de type **`int`** et `Integer`. En effet, le compilateur Java va automatiquement insérer des fonctions de conversion traduisant l'un en l'autre. Le code suivant est donc valide :

```
int n = 0;
Integer m = 4;
m = m + n;
n = n + m;
```

On ne va pas donner le code précis définissant ces types enveloppés. A ce stade, on peut considérer qu'il ressemble à cela :

```
class Integer {
  final int content;
  Integer(int n) { content = n; }
  int getInt() { return(this.content); }
}
```

Et que le compilateur Java insère la fonction `getInt` et le constructeur `Integer` lorsque c'est nécessaire.

Parmi les types enveloppés fournis par Java, citons les classes `Float`, `Double` et `Long`, respectivement pour les types **`float`**, **`double`** et **`long`**.

Attention Lorsque l'on utilise des types enveloppés, il faut faire attention à la manière de vérifier l'égalité. En effet, sur des objets, l'opérateur `==` vérifie l'*égalité physique*, c'est-à-dire si l'adresse des représentations mémoires est la même. On donc avoir deux instances `n` et `m` de `Integer` qui représentent le même nombre mais à des emplacements différents du tas, par exemple dans la situation représentée figure 5.5. Dans ce cas `n == m` vaudra **`false`**.

La bonne méthode pour tester l'égalité de deux objets est d'utiliser la méthode `equals`, dans ce cas par `n.equals(m)`. Cette question est traitée dans la section 6.8.1.

2. Cela se comprend bien si l'on se rappelle la différence de représentation mémoire : une variable de type **`int`** correspond à un emplacement mémoire contenant effectivement un nombre entier ; en revanche une variable instance d'une classe contiendra toujours un pointeur vers une adresse du tas.

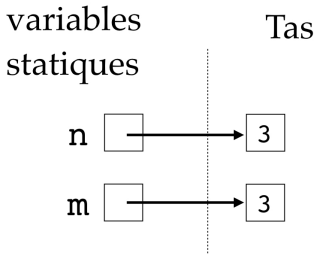


FIGURE 5.5 – Deux variables de types `Integer` représentant le même entier mais physiquement distinctes.

5.12.2 Classes récursives enveloppées

Revenons sur la définition des listes utilisée précédemment ; pour simplifier, on prend la version non-générique :

```
class Liste {
  int head;
  Liste tail;
  Liste(int i, Liste l) {
      head = i;
      tail = l;  }
}
```

Essayons maintenant de définir la fonction qui calcule la longueur d'une liste, mais cette fois comme une méthode dynamique, c'est-à-dire qu'on veut pouvoir l'invoquer par `l.length()` au lieu de `length(l)` comme c'est le cas pour la fonction statique définie précédemment. On est tenté d'ajouter à la classe précédente quelque chose comme :

```
  int length() {
    if (this == null) return(0);
    return(1 + this.tail.length());
  }
```

> **Attention**, car ce code ne fonctionnera pas ! En effet, l'objet **`null`** ne porte pas de méthodes. Autrement dit, le test **`this == null`** est absurde, car si on exécute la méthode, alors **`this`** est précisément différent de **`null`**.

C'est pourquoi, lorsque l'on veut utiliser la puissance de l'orienté-objet avec des types récursifs comme les listes, il faut utiliser deux classes :

- de manière interne, une classe récursive qui code le contenu des listes comme on l'a vu jusqu'ici,
- une classe qui enveloppe la précédente et peut contenir les méthodes.

Voici une implémentation possible. On commence donc par définir une classe récursive similaire à ce qu'on a déjà vu. Comme cette classe ne sera visible qu'à l'intérieur du container, on peut lui choisir un nom plus long :

```
class ListContent<E> {
  E head;
  ListContent<E> tail;
  ListContent(E i, ListContent<E> l) {
```

```
      head = i;
      tail = l;
   } }
```

On définit ensuite la classe `Liste` proprement dite. Les éléments de cette classe ne contiennent qu'un champ de type `ListContent`. Un élément de `Liste` sera vide si et seulement si le contenu de ce champ vaut `null`. Du coup, on a intérêt à donner deux constructeurs, un pour construire des listes vides, et un autre pour construire des listes non-vides :

```
class Liste<E> {
    ListContent<E> content;
    Liste() {
        content = null;
    }
    Liste(E i, Liste<E> l) {
        content = new ListContent<E>(i, l.content);
    }
```

On peut maintenant définir une méthode qui rend le premier élément d'une liste en utilisant donc la syntaxe `l.head()`. À noter que lorsque cette méthode est invoqué sur une liste vide, on peut invoquer une erreur d'exécution choisie :

```
    E head() {
        if (content == null) throw new Error("Empty list");
        return(content.head);
    }
```

On fait la même chose pour construire une méthode `tail()`. Remarquez qu'on choisit ici d'utiliser une troisième version de constructeur pour `Liste` :

```
    Liste(ListContent<E> l) { content = l; }

    Liste<E> tail() {
        if (content == null) throw new Error("Empty list");
        return(new Liste<E>(content.tail));
    }
```

On peut par exemple définir la méthode `length()` récursivement, en enchaînant les invocations de méthodes :

```
    int length() {
        if (this.content == null) return(0);
        return(1 + this.tail().length());
    }
```

Solution page 188. **Exercice 5.12.1** Pourquoi, à la suite des définitions précédentes, le code suivant échoue-t-il ? Comment le corriger ?

```
public static void main(String[] a) {
   Liste<String> l =
          new Liste<String>(
                     "a",
                      new Liste<String>("b", null));
   System.out.println(l.length());
   }
}
```

Remarquez qu'il est tout à fait naturel d'envelopper de la même manière d'autres classes récursives, comme celle définissant les arbres binaires 5.10 ou les arbres n-aires (exercice 5.11.1).

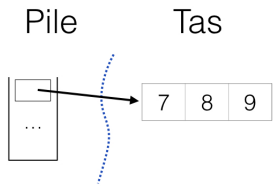


FIGURE 5.6 – La création d'un triplet dans le tas.

5.13 Compilation des enregistrements

5.13.1 Instructions XVM pour le tas

La gestion du tas est plus complexe que les tâches effectuées par les instructions de la XVM vues jusqu'à maintenant. Aussi, les instructions supplémentaires dont nous aurons besoin pour compiler les classes avec des champs dynamiques sont un peu plus évoluées. En particulier, elles ne correspondent pas à des instructions élémentaires de processeurs physiques.

Nous avons besoin d'allouer des blocs de mémoire dans le tas, puis d'y écrire et lire des informations. Les trois instructions correspondantes sont :

ALLOC qui va réserver dans le tas un bloc mémoire composé de x mots contigus, où x est la valeur en haut de la pile. Cette valeur x est remplacée, en haut de la pile, par l'adresse mémoire en question.

CREAD cette instruction, abréviation de *Computed address READ* permet de lire le contenu d'une adresse mémoire. Elle ressemble donc beaucoup à READ, mais elle utilise deux éléments de la pile. Si la pile est de la forme $x; y; \ldots$, elle va lire le contenu de l'adresse $x + y$. On dit que y est l'adresse, et x le décalage (*offset*). Cette possibilité d'ajouter un *offset* à une adresse correspond à ce qui existe dans des processeurs réels ; dans notre cas, cela nous permet d'accéder aux différents mots d'un bloc. Après l'exécution de CREAD, les valeurs x et y sont enlevées de la pile et remplacées par la valeur lue dans le tas. La hauteur de la pile a donc diminué de 1.

CWRITE cette instruction dont le nom est l'abréviation de *Computed address WRITE* et est la duale de la précédente. Si la pile est de la forme $x; y; z; \ldots$ elle va écrire la valeur z à l'adresse y avec l'*offset* x (donc à l'adresse $x + y$). Ensuite, la hauteur de la pile est décrémentée de trois.

En pratique, on utilisera les instructions CREAD en CWRITE pour lire et écrire à des adresses mémoires du tas.

Illustrons le fonctionnement de ces instructions sur un exemple simple.

On veut créer un triplet contenant les valeurs 7, 8 et 9 et placer son adresse en haut de la pile, et ainsi atteindre la situation de la figure 5.6 :

```
PUSH(3)
ALLOC    ; on alloue un bloc de 3 mots
PXR      ; on place cette adresse dans R
PUSH(7)  ; la première valeur qu'on va écrire
PRX      ; à l'adresse créée
PUSH(0)  ; à l'offset 0
CWRITE
PUSH(8)  ; on recommence pour la deuxième valeur
PRX      ; à la même adresse
PUSH(1)  ; cette fois à l'offset 1
CWRITE
PUSH(9)  ; la troisième valeur qu'on va écrire
PRX
PUSH(2)  ; à l'offset 2
CWRITE
PRX      ; on a fini et place l'adresse en haut de pile
```

Remarquons que l'on peut, dans la solution ci-dessus, remplacer la séquence PUSH(0); CWRITE par WRITE, ce qui est une petite optimisation simple.

Solution page 188. **Exercice 5.13.1** On remarque aussi, dans l'exemple ci-dessus, qu'on s'est autorisé à utiliser le registre R. Proposez une séquence équivalente qui n'utilise que la pile. ◇

5.13.2 Compilation des tableaux

Commençons par nous intéresser aux tableaux. On se rappelle que les tableaux sont créés à l'exécution par des instructions comme :

```
t = new int[50];
r = new String[20];
```

Plus précisément, il s'agit ici d'affectations ; ce qu'il nous faut compiler sont les expressions `new int[50]` et `new String[20]`.

Les tableaux sont alloués dans le tas. Pour un tableau de taille n, on va utiliser un bloc de $n + 1$ mots :

- le premier mot pour stocker la taille du tableau,
- les n suivants pour les n cases du tableau.

Donc l'expression `new T[e]`, où `e` est une expression de type `int`, pourra être compilée par :

```
⟦e⟧      ; calcule la valeur e (la taille du tableau) et la place sur la pile
FETCH(0) ; on copie la taille du tableau pour s'en souvenir après ALLOC
PUSH(1)
ADD      ; ajoute 1 pour obtenir la taille du bloc alloué
ALLOC    ; on a en haut de la pile : la nouvelle adresse puis la taille
PXR      ; on met l'adresse dans R pour s'en souvenir
PRX      ; on remet une copie de l'adresse sur la pile
PUSH(0)  ; offset pour écrire la taille
CWRITE   ; on écrit la taille au début du bloc
PRX      ; on remet le pointeur vers le tableau en haut de la pile
```

Remarquez que :

- Il s'agit bien de la compilation d'une expression : on place la valeur en haut de la pile. Ici la "valeur" du tableau est l'adresse de celui-ci dans le tas.

– On a un peu simplifié les choses ici en ne se préoccupant pas de l'initialisation des valeurs du tableau (en Java, les cases d'un nouveau tableau de `int` valent 0, un tableau de classe valent `null`, etc. . .).

Accès à un tableau

On accède au valeurs des cases d'un tableau par des expressions comme `t[4]`, ou plus généralement `t[e]` où `t` est une expression de type tableau, et `e` une expression de type `int`. La compilation d'une telle expression est remarquablement simple :

```
⟦t⟧        ; on place l'adresse du tableau sur la pile
⟦e⟧        ; on place la valeur de l'indice sur la pile
PUSH(1)
ADD        ; on ajoute 1 à l'indice car le premier mot est réservé à la longueur
CREAD      ; on récupère la valeur recherchée
```

Exercice 5.13.2 On a ici aussi fait une petite simplification par rapport au compilateur Java réel. Voyez-vous laquelle ? ◇ Solution page 188.

Exercice 5.13.3 Proposez une manière de compiler l'expression `t.length`. ◇ Solution page 188.

Exercice 5.13.4 Proposez une manière de compiler l'affectation d'une expression à une case d'un tableau, c'est-à-dire les instructions de la forme `t[e1] = e2;`. ◇ Solution page 188.

5.13.3 Classes sans méthode

Pour bien expliquer la compilation des classes et des méthodes, il faut tenir compte des mécanismes d'héritage qui sont traités dans le chapitre suivant. Pour l'instant, on va donc juste voir comment compiler des enregistrements simples sans méthodes dynamiques.

Représentation

Prenons une classe simple comme exemple :

```
class Exemple {
  static int an = 2021;
  final static int b = 4;

  float champFloat;
  int champInt;
  String champString;

  Exemple(float x, int y, String s) {
     champFloat = x;
     champInt = y;
     champString = s;
  }
  static int f ( ... ) { ... } // puis éventuellement
                               //d'autres fonctions statiques
}
```

Ici, on a regroupé les variables statiques d'abord, les champs ensuite et les fonctions statiques ensuite. Mais le fichier source pourrait les présenter dans un ordre différent, le traitement serait le même.

Les variables statiques et les fonctions statiques sont compilées comme présenté dans les chapitres précédents. On se concentre donc sur les champs dynamiques.

Le compilateur énumère et numérote les champs, un peu comme il assigne une adresse à chaque variable statique. La différence est que chaque champ se voit assigné un *numéro* et pas une adresse, en commençant par 1. Cela peut donc donner :

$$\begin{aligned} \text{num(}\texttt{champFloat}\text{)} &= 1 \\ \text{num(}\texttt{champInt}\text{)} &= 2 \\ \text{num(}\texttt{champString}\text{)} &= 3 \end{aligned}$$

Remarquez que cette numérotation (re-)commence à 1 pour chaque classe.

Les objets de classe `Exemple` correspondent donc à des triplets. Ils seront représentés dans le tas par des blocs de quatre mots :

- Les valeurs respectives des champs `champFloat`, `champInt` et `champString` dans les mots d'*offset* 1, 2 et 3.
- Le premier mot, à l'*offset* 0, est réservé pour le traitement des méthodes. On n'utilisera pas sa valeur ici (c'est-à-dire que sa valeur est indifférente pour les classes sans méthodes). On comprendra l'intérêt de ce mot supplémentaire au chapitre 6.

Autrement dit, un objet créé par `new Exemple(6.6, 7, "abc")` sera représenté dans le tas par un bloc de quatre mots, de la forme :

0	6.6	7	p

où p est un pointeur vers la représentation de la chaîne `"abc"`.

On voit qu'un objet de classe `Exemple` ressemble beaucoup à un tableau de taille 3. On accède aux champs avec leurs numéros, attribués lors de la compilation et donc invisibles pour le programmeur. Une différence est, bien sûr, que les types des champs ne sont pas tous identiques.

Compilation de l'accès aux champs

Si `x` est un objet de classe `Exemple`, on sait déjà que ⟦`x`⟧ est un code machine qui va placer sur la pile l'adresse à laquelle se trouve la représentation de l'objet.

On peut donc compiler l'expression `x.champInt` par :

```
⟦x⟧       ; met l'adresse sur la pile
PUSH(2)   ; car num(champInt) vaut 2
CREAD     ; récupère la valeur du champ dans le tas
```

Les accès aux autres champs sont évidemment compilés de manière similaire.

Compilation de l'affectation des champs

Supposons que `e` soit une expression de type `int`. On peut compiler l'instruction d'affectation `x.champInt = e` par :

```
⟦e⟧       ; calcule la valeur de e et la met sur la pile
⟦x⟧       ; met l'adresse sur la pile
PUSH(2)   ; car num(champInt) vaut 2
CWRITE    ; écrit la valeur du champ au bon endroit
```

Là encore, les cas des autres champs sont quasi-identiques.

Compilation des constructeurs

En général, un constructeur est de la forme `Exemple(T1 x1, ..., Tn xn){ p }`, où `p` est une suite d'instructions, qui sont typiquement des affectations des champs.

Détailler complètement le code serait fastidieux, mais on peut en comprendre le principe essentiel :

- Le constructeur est vu comme une fonction, en particulier les arguments `x1`... `xn` sont traités comme les arguments d'une fonction habituelle.
- Le code compilé du constructeur commencera par effectuer une opération ALLOC pour obtenir l'adresse d'un nouveau bloc de la bonne taille dans le tas. Cette adresse sera retournée comme résultat par la fonction.
- Le code `p` sera compilé de la manière habituelle. À l'intérieur de ce code, l'expression **`this`** correspondra à la nouvelle adresse.
- Les affectations des valeurs des champs dans `p` pourront alors être traitées comme décrit dans la paragraphe précédent.

Au final, le constructeur est traité quasiment comme une fonction habituelle. C'est juste dans le code source Java qu'il est appelé avec le mot clé **`new`**.

Chapitre 6

Héritage

6.1 Sous-classes

Le premier trait orienté-objet de Java réside donc dans la possibilité pour les objets de porter des méthodes dynamiques. Ce trait prend toute son importance quand il est renforcé par l'héritage.

La meilleure manière d'expliquer l'héritage est de prendre un exemple simple, souvent lié, justement, à des objets du monde réel. Reprenons le thème du chapitre précédent, qui traitait de véhicules.

Tous les véhicules partagent certaines propriétés :

- ils ont chacun un numéro d'immatriculation,
- ils ont chacun une année de mise en circulation.

Mais d'autres propriétés ne sont partagées que par des sous-ensembles de véhicules :

- un nombre maximum de passagers pour les voitures,
- parmi les voitures, les voitures à essence ont une consommation moyenne, ainsi qu'un niveau de pollution crit'air (de 1 à 5),

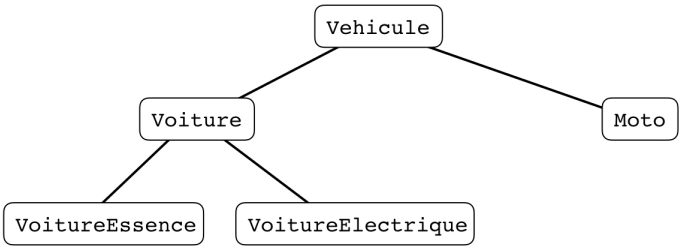


FIGURE 6.1 – Une petite hiérarchie de classes

– en revanche, pour les voitures électriques, on voudra connaitre leur autonomie (en km).

On a donc une structure d'arbre (figure 6.1) : la catégorie générale des véhicules de l'entreprise, à l'intérieur de celle-ci deux sous-ensembles, motos et voitures, ces dernières étant à nouveau partagées entre voitures à essence et électriques.

L'héritage permet exactement cela : avoir des objets des classes `Voiture` ou `VoitureElectrique` qui soient aussi des instances de `Vehicule`. On dit que `Voiture` est une *sous-classe* de `Vehicule`.

On dit aussi que `Voiture` étend `Vehicule`, puisqu'elle en possède tous les champs et méthode, en plus d'autres qui lui sont propres.

Pour clarifier l'exemple, repartons d'une définition très simple de la (sur-)classe des véhicules :

```
class Vehicule {
    public String immatriculation;
    public final int anneeCirculation;

    Vehicule(String i, int a) {
        immatriculation = i;
        anneeCirculation = a;
    }
}
```

Dans un autre fichier, on peut maintenant définir la (sous-)classe des voitures :

```
class Voiture extends Vehicule {
    public int nombrePlaces;

    Voiture(String i, int a, int n) {
        super(i,a);
        nombrePlaces = n;
    }
}
```

Le mot-clé important est **`extends`** qui indique donc que `Voiture` étend `Vehicule`. Cela veut dire :

– Tout objet instance de `Voiture` est également instance de `Vehicule`,

– la classe `Voiture` hérite des champs de `Vehicule` ; c'est-à-dire que `anneeCirculation` et `immatriculation` sont également des champs de la nouvelle classe, sans qu'il soit nécessaire de le rappeler dans la définition.

Nous expliciterons la définition du constructeur au paragraphe 6.4. Pour l'instant vérifions que l'héritage fonctionne. On peut définir une fonction prenant un `Vehicule` en argument :

```
static void afficheImmat(Vehicule v) {
    System.out.println(v.immatriculation);
}
```

et appliquer cette fonction à une `Voiture` :

```
Voiture w = new Voiture("XX", 1987, 5);
afficheImmat(w);
```

6.2 Hierarchie de sous-classes

La relation de sous-classe est transitive, on peut donc définir des sous-classes de `Voiture` qui seront également des sous-classes de `Vehicule` :

```
class Electrique extends Voiture {
    int autonomie;

    Electrique (String i,  int an, int p,  int au) {
        super(i, an, p);
        autonomie = au;
    }
}

class Essence extends Voiture {
    int consommation;

    Essence (String i,  int a, int p,  int c) {
        super(i, a, p);
        consommation = c;
    }
}
```

On voit donc qu'un objet peut avoir plusieurs classes (par exemple `Electrique`, `Voiture` et `Vehicule`. Il n'a par contre qu'une seule *classe principale*. La classe principale d'un objet est la classe la plus petite d'un objet, c'est-à-dire la plus précise. La classe principale d'un objet est la classe correspondant au constructeur avec lequel l'objet est créé.

6.3 Sur-classes et sur-classe directe

En Java, il n'y a pas d'héritage multiple ; c'est-à-dire qu'une sous-classe n'hérite directement que d'une seule classe. Ou autrement dit, le mot clé **`extends`** ne peut être utilisé qu'une fois dans une déclaration de classe.

C'est pour cela que l'on peut parler de l'arbre d'héritage (figure 6.1). Chaque classe a au plus une sur-classe directe, qui est son père dans l'arbre d'héritage. La sur-classe directe de `Electrique` est `Voiture`, la sur-classe directe de `Voiture` est `Vehicule`.

A l'intérieur d'une classe, on peut faire référence à la sur-classe directe par le mot-clé **`super`**. En général, on utilise indifféremment les termes sur-classe et super-classe.

6.4 Constructeurs des sous-classes

On peut maintenant expliquer la syntaxe des constructeurs des sous-classes. Le principe est simple :

- les champs qui sont spécifiques à la sous-classes sont initialisés explicitement par le constructeur,
- en revanche, les champs qui sont hérités de la super-classe doivent être initialisés par un appel au constructeur de la super-classe,
- on fait référence à ce constructeur par le mot-clé **`super`**

Dans les exemples précédents, on voit donc que :

- le champ `autonomie` des instances de `Electrique` est initialisé par le constructeur `Electrique`,
- les autres champs sont initialisés par **`super`**`(i, an, p)` qui appelle de constructeur de `Voiture`,

– dans le constructeur de `Voiture`, on trouve à nouveau un appel au constructeur de `Vehicule` pour initialiser les champs `immatriculation` et `anneeCirculation`.

On rappelle que le constructeur de la super-classe peut être surchargé. Si aucun appel au constructeur de la super-classe n'apparaît, alors le compilateur cherche à appeler la version sans argument du constructeur de la super-classe (dans l'exemple ici ce serait `Voiture();`). S'il n'existe pas de telle version du constructeur de la super-classe, alors le compilateur engendre un message d'erreur.

6.5 Héritage et redéfinition de méthodes

La sous-classe hérite non seulement des champs, mais aussi des méthodes de sa sur-classe. C'est la caractéristique qui donne toute sa puissance à l'approche orientée-objet.

6.5.1 Principe

Ajoutons à la classe `Vehicule` une méthode qui affiche l'immatriculation :

```
public affiche() {
    System.out.println("Vehicule : "+immatriculation);
}
```

Si on ne fait rien de plus, les classes `Voiture`, `Essence`, etc vont hériter de cette méthode. C'est-à-dire qu'on pourra écrire :

```
Voiture v = new Voiture("XX", 2017, 5);
v.affiche();
```

Ce qui affichera `Véhicule : XX`.

Mais on peut aussi, dans la définition de la sous-classe, choisir de *re-définir* la méthode - en anglais *override the method*. Cela permet de la spécialiser pour la sous-classe. Concrètement, on ajoutera pour cela dans la classe `Voiture` quelque chose comme :

```
class Voiture extends Vehicule {
    ...
    public affiche() {
        System.out.println("Voiture : "+immatriculation+" a "
       +places+" places");
    ...
    }
```

Alors, l'appel de la méthode `affiche` pour les instances de `Voiture` donneront lieu à l'affichage plus complet avec le nombre de places.

6.5.2 Annoter les redéfinitions

En anglais, la redéfinition de méthode se dit *override*. Il est possible de signaler qu'une méthode est bien la redéfinition d'une méthode de la super-classe en ajoutant l'annotation `@Override`. Le code suivant est équivalent au précédent :

```
class Voiture extends Vehicule {
    ...
    @Override
    public affiche() {
```

```
        System.out.println("Voiture : "+immatriculation+'' {\`a}
        "+places+" places");
    ...
    }
```

En revanche, l'utilisation de l'annotation `@Override` sur une méthode qui n'est pas une redéfinition déclenche une erreur lors de la compilation. L'intérêt de signaler ainsi qu'il s'agit d'une redéfinition est d'éviter un certain nombre d'erreurs courantes.

6.5.3 Utiliser `super` dans une redéfinition

Lorsqu'on redéfinit une méthode, il est possible de faire référence à la méthode de la super-classe avec le mot-clé `super`. Par exemple :

```
class Adresse {
  String rue;
  int numero;
  Adresse(String r, int n) { rue = r; numero = n; }
  void affiche() { System.out.println(" "+numero+" "+rue); }
}

class AdresseDetaillee extends Adresse {
   int etage;
   AdresseDetaillee(String r, int n, int e) { super(r, n); etage = e; }

   @Override
   void affiche() {
       super.affiche();                          // affiche l'adresse
       System.out.println("etage " + etage);     // affiche l'étage
   }
}
```

6.5.4 Redéfinition interdite : `final`

Si l'on veut interdire à d'éventuelles sous-classes de redéfinir une méthode, il suffit d'utiliser le mot-clé **`final`** là où elle est déclarée. On peut également déclarer toute la classe **`final`**, auquel cas on interdit toute spécialisation de la classe. Autrement dit, on ne peut pas définir de sous-classe d'une classe **`final`**. De même, les méthodes statiques ne peuvent pas être redéfinies.

6.5.5 Liaison dynamique

Une caractéristique importante de l'héritage est que l'appel de méthode est dynamique. Pour reprendre l'exemple ci-dessus, si `v.affiche();` apparaît quelque part dans le code, le choix du code exécuté sera de la responsabilité de l'objet `v` et n'est pas connu lors de la compilation. Cela veut dire que les objets Java portent avec eux les adresses du code de leurs méthodes. On précise cela avec le schéma de compilation en 6.9.

C'est aussi la différence essentielle entre l'héritage de méthodes et la surcharge vue en 3.5 qui elle peut-être résolue au moment de la compilation.

6.6 Classes abstraites et interfaces

Toujours dans l'exemple des véhicules, on peut trouver qu'il n'est pas vraiment utile de pouvoir créer des objets `Vehicule` qui ne soient ni `Voiture` ni `Moto`, ou des `Voiture` qui ne soient ni `Essence` ni `Electrique`. On peut facilement interdire cela, en transformant `Vehicule` et `Voiture` en *classes abstraites*.

Un classe abstraite reste un nœud dans l'arbre de la hiérarchie des classes, mais ne contiendra pas d'instances. En revanche, on peut préciser un certain nombre de propriétés de la classe — ou des instances de ces sous-classes :

- présence de certains champs,
- présence de certaines méthodes.

Dans le cas des méthodes, on ne va pas, le plus souvent, donner le code de ces méthodes. Ce sont alors des *méthodes abstraites*. Voici une version abstraite des classes `Vehicule` et `Voiture` puis une implémentation de `Essence` :

```
abstract class Vehicule {
    public String immatriculation;
    public int anneeCirculation;

    abstract void affiche();
}

abstract class Voiture extends Vehicule {
    public int nombrePlaces;

    public void affiche() {
        System.out.println("Voiture " +immatriculation
                        +" a "+nombrePlaces
                        + " places");
    }
}

class Essence extends Voiture {
    int consommation;

    Essence (String i,  int a, int p,  int c) {
        immatriculation = i;
        anneeCirculation = a;
        nombrePlaces = p;
        consommation = c;
    }
}
```

On voit qu'il est possible de proposer une implémentation d'une méthode dans une classe abstraite, ici avec `affiche()` dans `Voiture`. On voit egalement que les classes abstraites n'ont évidemment pas de constructeur puisqu'elles n'ont pas d'habitant. Aussi, lorsque la superclasse est une classe abstraite, on ne va pas faire appel au constructeur **`super`**.

De manière générale, les classes abstraites permettent donc de *spécifier une interface*, c'est-à-dire de préciser les caractéristiques d'une classe : champs et méthodes. Nous utilisons ici le mot "interface" pour désigner ce qui est partagé entre l'intérieur et l'extérieur d'une classe. Le mot-clé **`interface`** en Java a un sens plus précis et plus restreint, il s'agit essentiellement d'une alternative aux classes abstraites.

Alors que dans une classe abstraite certaines méthodes peuvent être définies, ce n'est pas le

cas avec les interfaces où les méthodes sont indéfinies et où il n'y a pas de méthodes statiques ou de champs de données modifiables.

Autrement dit, une interface ne comporte que des méthodes abstraites et publiques.

Alors que les classes peuvent étendre d'autres classes (`extends`) pour les interfaces on dit que :

- Une interface peut *spécialiser* une autre interface (`extends`).
- Une classe peut *implémenter* une ou plusieurs interfaces (`implements`).

C'est le dernier point qui justifie particulièrement l'existence des interfaces en plus des classes abstraites. Le fait qu'une classe puisse implémenter plusieurs interfaces permet de contourner l'absence d'héritage multiple.

6.7 Vérification dynamique de la classe

Le système de types et de classes de Java n'est pas parfait. Il est parfois nécessaire de forcer la main au compilateur, ou plutôt à la vérification de type faite au moment de la compilation. On présente ici deux fonctionnalités de Java, parfois indispensables mais qu'il vaut mieux éviter quand c'est possible.

6.7.1 Vérifier la classe

Dans certains cas, on souhaite déterminer si un objet est ou non instance d'une certaine classe. Voici un exemple (pas très profond, je l'admets) : on a une classe décrivant les personnels de l'école :

```
class Personne {
  int age;
  String nom;
  Personne (String n, int a) { nom = n; age = a; }
  String salut (Personne p) { return "Bonjour!"; }
}
```

La méthode `salut` donne la phrase de bienvenue ; c'est-à-dire que `a.salut(b)` est ce que `a` dit à `b`.

On définit une sous-classe décrivant les élèves. On voit que les élèves sont plus cools dans leur salut :

```
class Eleve extends Personne {
   int promotion;
   String sectionSport ;
   Eleve (String n, int a, int promo, String sec)
     { super(n, a); promotion = promo; sectionSport = sec; }
   @Override
   String salut(Personne p) { return("Salut !"); }
```

Supposons que l'on trouve que la définition de `salut()` ci-dessus rend les élèves trop peu respectueux de leurs aînés. On va alors paramétrer le salut par la personne à qui il est adressé :

```
class Eleve extends Personne {
```

```
  ...
  @Override
  String salut (Personne p) {
    if (p instanceOf Eleve)
      return "Salut!";
    else
      return super.salut(p);
    }
}
```

Ici, lorsque la personne saluée n'est pas un élève, on appelle la méthode de la sur-classe. La sur-classe est désignée par **`super`** et dans le cas présent se référera donc à `Personne`. Le salut adressée aux personnes qui ne sont pas des élèves sera donc `Bonjour!`.

Pour donner une règle générale : l'utilisation de **`instanceof`** est à éviter lorsque c'est possible. En revanche, **`super`** est une construction orienté-objet légitime et du meilleur goût.

Notons que l'opérateur **`instanceof`** fait intervenir un objet de classe `Class`. Cette classe permet de décrire la hiérarchie des classes de Java : chaque classe est représentée par une instance de `Class`. Mais nous éviterons de rentrer dans ces détails.

6.7.2 Casts

Dans certains cas, il faut demander à la vérification de types de voir un objet avec une classe différente que celle apparente à la vérification de types. On appelle généralement un *cast* cette opération en théorie des langages de programmation.

On verra des utilisations nécessaires de cette construction en 6.8.1 et 6.8.2. Donnons pour l'instant simplement sa syntaxe. Si un objet `a` est de classe `A` mais on veut le voir localement comme de classe `B`, la syntaxe est : `(B)a`.

Remarquons qu'il s'agit là uniquement d'une instruction donnée au vérificateur de types. Elle n'est pas traduite dans le code compilé [1].

Remarquons aussi que lorsqu'une (mauvaise) utilisation de cast crée une situation problématique lors de l'exécution (par exemple l'objet ne possède pas la classe invoquée), la JVM lance une erreur `ClassCastException` qui peut éventuellement être rattrapée par le mécanisme d'exceptions (voir le chapitre 7 pour les exceptions).

6.8 La hiérarchie des classes

La relation de sous-classe est transitive : une classe hérite de sa sur-classe, et par là aussi de la sur-classe et sa sur-classe, etc... La relation d'héritage définit en fait un ordre partiel sur l'ensemble de toutes les classes Java. Aussi, un objet peut appartenir à plusieurs classes. Parmi celles-ci, la classe qui le caractérise le plus précisément dans la hiérarchie d'héritage est appelée sa *classe principale*.

Cette relation d'ordre admet aussi un élément maximal, qui est la classe `Object`. Cette classe est prédéfinie.

Lorsque qu'une classe est définie sans utiliser le mot-clé **`extends`**, sa sur-classe est donc `Object`. La signature de `Object` contient en particulier :

```
public boolean equals (Object o);
```

1. Ou, pour être tout à fait précis : elle peut influencer indirectement le code compilé en influençant la manière dont sera résolue la surcharge (*overloading*).

```
public int hashCode ();
public void finalize ();
```

Tout objet, quelle que soit sa classe principale, dispose donc de ces trois méthodes. Les méthodes `equals` et `hashCode` sont particulièrement importantes. Décrivons rapidement la troisième.

`finalize()` est appelée lors de l'effacement de l'objet par le GC. Tant qu'elle n'est pas redéfinie, cette méthode ne fait rien. Elle est assez délicate à utiliser, mais peut-être utile pour débugger certains programmes, par exemple pour détecter des fuites de mémoire.

Exercice 6.8.1 (un peu idiot) Soit la classe `Eleve` suivante : Solution page 188.

```
class Eleve {
   String nom;
   int age;
   Eleve(String n, int a) { nom = n; age = a}
}
```

On veut que tous les élèves se comportent silencieusement lorsque le GC les efface de la mémoire de l'école. Seul un élève turbulent, appelons-le "Arthur" veut afficher un message d'adieu avant de disparaître. Comment peut-on arriver à ce résultat ? ◇

6.8.1 Tester l'égalité : `equals`

La méthode `equals` est importante. Elle permet de traiter le point déjà abordé dans la section 5.12.1 de l'égalité physique et l'égalité structurelle. Il y a souvent, en informatique, plusieurs manières de considérer l'égalité. Dans le cas des objets, cela demande un peu d'explication.

Egalité physique

Supposons que l'on ait défini une classe `Point` :

```
class Point {
  int x, y;
  Point (int x, int y) { this.x = x; this.y = y; }
}
```

Supposons de plus que l'on dispose de deux variables `p1` et `p2` dont le type est `Point`. Il est donc permis d'effectuer le test `p1 == p2`. Cette expression produit un résultat booléen. Quel sera le résultat de ce test ?

Au niveau du code machine, la valeur d'un objet est une *adresse* mémoire dans le tas. Aussi, le résultat de `p1 == p2` sera **`true`** si et seulement si les variables `p1` et `p2` contiennent la même adresse.

Donc, si `p1 == p2` est vrai, on aura `p1.x == p2.x` et `p1.y == p2.y` ; on dit qu'on a alors égalité physique entre `p1` et `p2`. Bien sûr, l'inverse n'est pas vrai : les égalités `p1.x == p2.x` et `p1.y == p2.y` signifient que les objets `p1` et `p2` ont le même contenu, mais n'impliquent pas que `p1` et `p2` sont un seul et même objet (c'est-à-dire sont représentés à la même adresse mémoire).

Egalité structurelle

Le test d'égalité physique est rapide, puisqu'il revient à vérifier l'égalité de deux adresses mémoire, mais il est souvent insuffisant. En particulier, il ne tient pas compte du *contenu* des objets comparés. L'égalité structurelle entre deux objets est définie comme l'égalité entre les contenus des champs.

Pour la classe `Point` ci-dessus, deux points sont structurellement égaux si et seulement si leurs coordonnées `x` et `y` sont identiques.

Java ne fournit pas de test d'égalité structurelle. Il incombe au programmeur de le définir de façon appropriée pour chaque classe.

Cela demande un peu de travail, mais permet d'ajuster le test d'égalité de manière plus fine ou plus efficace pour certaines classes. Par exemple, deux individus pourront être déclarés égaux s'ils ont le même numéro de sécurité sociale, sans qu'il soit nécessaire de comparer leurs noms.

Par convention, la méthode `equals` est censée effectuer un test d'égalité *approprié*. Son implémentation par défaut, définie dans la classe `Object`, effectue simplement un test d'égalité physique. Si on souhaite munir une sous-classe d'une autre notion d'égalité, par exemple une forme d'égalité structurelle, alors on doit redéfinir la méthode `equals` pour implémenter l'égalité voulue. Par exemple, si on souhaite munir les points d'une notion d'égalité structurelle, on écrira :

```
class Point {
  final int x, y;
  Point (int x, int y) { this.x = x; this.y = y; }
  @Override
  public boolean equals (Object o) {
    try {
      Point that = (Point) o;
      return this.x == that.x && this.y == that.y;
    } catch (ClassCastException e) {
      return false;
    }
  }
}
```

On pourra ensuite effectuer un test d'égalité entre deux points `p` et `q` en appelant `p.equals(q)` ou `q.equals(p)`. On explicite l'utilisation de **`try`** et **`catch`** dans la solution de l'exercice suivant :

Solution page 189. **Exercice 6.8.2 (Recommandé)** Pourquoi n'écrit-on pas plus simplement ceci ?

```
class Point {
  final int x, y;
  Point (int x, int y) { this.x = x; this.y = y; }
  public boolean equals (Point that) {
    return this.x == that.x && this.y == that.y;
  }
}
```

Ce code étant lui aussi accepté par le compilateur Java... ◇

Solution page 190. **Exercice 6.8.3** Pourquoi a-t-on ajouté au passage le mot-clef **`final`** dans la déclaration des champs `x` et `y` ? ◇

6.8.2 Le cas des tableaux génériques

Pour des raisons techniques assez obscures, Java interdit les tableaux génériques. C'est-à-dire qu'à l'intérieur d'une classe générique `Exemple<E>`, on ne peut pas utiliser le type `E[]`, le type des tableaux dont les éléments sont de classe `E`.

Une manière de contourner cela est de déclarer ces tableaux comme des tableaux de `Object`. Là où on voudrait écrire :

```
E[] t = new E[50];
```

on écrit :

```
Object[] t = new Object[50];
```

Il faut alors utiliser un *cast* lorsqu'on lit des éléments du tableau. Lorsqu'on veut faire, par exemple, `f(t[i])`, où `f` est une fonction prenant en argument une instance de `E`, on doit écrire `f((E)t[i])` pour garantir que `t[i]` est bien une instance de `E`.

6.9 Compilation des méthodes et de l'héritage

6.9.1 Principe

Dans la section 5.13 nous avons vu comment on peut compiler des classes contenant des champs dynamiques, mais sans méthode et sans héritage. Nous pouvons maintenant regarder comment on compile les méthodes dynamiques, et ce d'une manière compatible avec l'héritage.

Saisir le schéma général permet de mieux comprendre le fonctionnement réel des mécanismes d'héritage.

Prenons à nouveau un exemple simple : une classe avec deux champs et deux méthodes :

```
class Point {
  int x;
  int y;

  Point(int a, int b) { x = a; y = b; }

  int s() { return(x + y); }
  int d() { return( x * x  + y * y); }
}
```

et une sous-classe qui ajoute un champ et une méthode supplémentaires et redéfinit une des deux méthodes hérités :

```
class ColorPoint extends Point {
  String color;

  ColorPoint(int a, int b, String c) {
     super(a, b);
     color = c;
   }

  String color() { return(color); }

  @Override                                  //exemple sans
  int d() { return( 3 * x + y * y + 1); } // signification particulière
}
```

Quelles sont les informations qui composent alors un objet de la classe `Point` ? Plusieurs choses :

- Tout d'abord, les valeurs de ses champs `x` et `y`. Cela correspond à ce que nous avons vu en 5.13 : un objet contient le n-uplet des valeurs de ses champs.
- Mais si cet objet appartient également à la sous-classe `ColorPoint`, il contiendra le champ supplémentaire `color`. Ce n-uplet ne sera pas un couple mais un triplet.
- Enfin l'objet doit indiquer quelles méthodes lui sont associées. Par exemple, suivant que la classe principale de l'objet est `Point` ou `ColorPoint`, le code de la méthode `d()` ne sera pas le même.

6.9.2 Sauts avec adressage indirect

Pour pouvoir compiler les méthodes dynamiques, nous allons utiliser une version un peu modifiée des instructions GTO et GSB de la XVM. Dans cette version, l'adresse à laquelle est effectuée le saut ne fait pas partie de l'instruction, mais est lue sur la pile : `s`

Instruction	pile avant	pile après	PC avant	PC après
GTO	$x\,;\,y\,;\,z\,;\ldots$	$y;z;\ldots$	n	x
GSB	$x\,;\,y\,;\,z\,;\ldots$	$n+1;y;z;\ldots$	n	x

On peut remarquer que les instructions GTO(n) et GSB(n) que nous utilisions jusqu'à maintenant peuvent être comprises simplement comme des abréviations des séquences respectives PUSH(n); GTO et PUSH(n); GSB. Le détail du comportement de l'instruction GSB avec la gestion du pointeur FP est donné dans le tableau récapitulatif de l'appendice A.

6.9.3 La table d'adressage des méthodes

Les deux méthodes `s()` et `d()` de la classe `Point` sont essentiellement des fonctions qui prennent chacune un argument `Point this`. Elles sont compilées en tant que telles :

- Leurs codes seront placés respectivement à des adresses adr(Point, s) et adr(Point, d). Remarquez que ces adresses sont fixées dès la compilation.
- Ces méthodes peuvent donc être appelées respectivement par PUSH(adr(s)); GSB; POP; PRX et PUSH(adr(d)); GSB; POP; PRX. Mais il faudra auparavant avoir poussé l'adresse de l'objet `this` sur la pile.

On peut détailler le code de la méthode `s`, qui se trouve donc en adr(Point, d). Rappelons encore que c'est une fonction qui prend un unique argument `this` :

```
RFR(-1)    ; recopie l'adresse this en haut de la pile
PUSH(1)    ; le numéro du champ x
CREAD      ; lit la valeur de this.x
RFR(-1)    ; recopie l'adresse this en haut de la pile
PUSH(2)    ; le numéro du champ y
CREAD      ; lit la valeur de this.y
ADD        ; on a maintenant x+y sur la pile
PXR        ; on pousse ce résultat en R
RET        ; fin de la méthode
```

Le compilateur va numéroter les méthodes d'une classe en commençant par 0. Par exemple `s` sera la méthode 0 et `d` la méthode 1 :

$$\begin{aligned} \text{num}(\text{Point}, s) &= 0 \\ \text{num}(\text{Point}, d) &= 1 \end{aligned}$$

Pour chaque classe, un emplacement particulier de la mémoire est utilisé pour stocker les adresses des méthodes ; c'est la *table d'adressage de cette classe*. Cette table comporte autant de mots que la classe comporte de méthodes (deux dans notre exemple[2]. Chaque mot est utilisé pour stocker l'adresse du code de la méthode correspondante.

Dans le cas de notre exemple, la table d'adressage de la classe `Point` est donc :

adr(Point,s)	adr(Point,d)

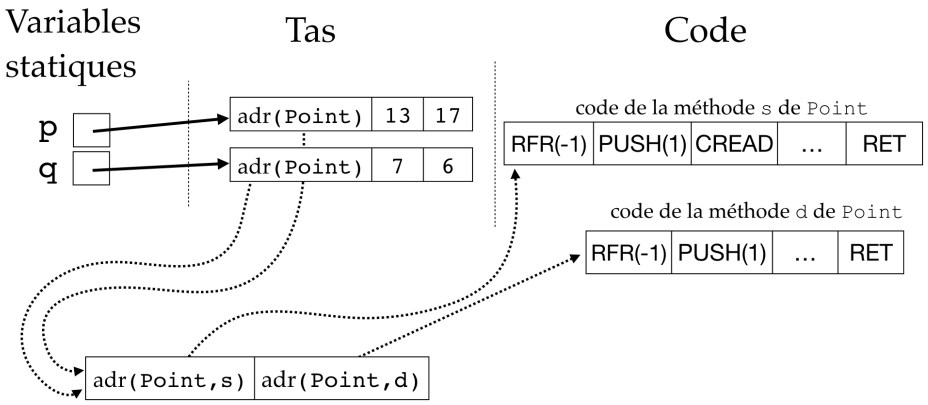
On appellera adr(Point) l'emplacement mémoire où est stockée cette table.

6.9.4 Représentation machine des objets

On a expliqué dans le chapitre précédent que les objets de la classe `Point` seront représentés par des blocs de 3 mots mémoire ; les deux derniers de ces trois mots contenant les valeurs des deux champs `x` et `y`. On peut maintenant préciser à quoi sert le premier mot mémoire : il va contenir adr(Point), c'est-à-dire l'emplacement de la table d'adressage de la classe de l'objet.

Autrement dit l'instruction suivante donnera le schéma mémoire ci-dessous :

```
static Point p = new Point(13,17);
static Point q = new Point(7,6);
```



6.9.5 Appel des méthodes

Le compilateur sait donc que `s` est la première méthode de `Point` et `d` la seconde. Si on veut compiler l'instruction `p.s()` il faut une séquence d'instructions qui place sur la pile :

– en haut l'adresse de la méthode adr(s),

2. On simplifie un peu ici, car en pratique la classe devrait hériter des méthodes de `Object`. Mais cela ne change absolument pas le principe décrit.

– en dessous l'argument, c'est-à-dire l'adresse de l'objet `p`.

Le code ⟦`p`⟧ va nous donner l'adresse de `p`. Le "truc" est qu'à partir de cette adresse, on peut retrouver l'adresse de la méthode, puisque :

– le triplet donne l'adresse de la table d'adressage,

– la table d'adressage contient l'adresse de la méthode.

L'adresse de la méthode est à la position correspondant à l'offset 3 du mot. On retrouve cet offset en additionant 0 (le numéro de la méthode) avec le contenu du premier mot du quintuplet (3).

Si on voulait obtenir l'offset de la méthode `d`, on additionnerait son numéro (1) avec le premier mot du quintuplet.

La séquence suivante est une compilation correcte de l'instruction `p.s();`.

```
⟦p⟧          ; place l'adresse de l'objet p en haut de la pile
FETCH(0)     ; copie une deuxième fois cette adresse en haut de la pile
             ; celle du haut va servir à trouver l'adresse de la méthode
             ; celle du bas sera l'argument this de la méthode
READ         ; on a maintenant l'adresse de la table d'adressage en haut de la pile
PUSH(0)      ; l'offset correspondant à la première méthode
CREAD        ; place l'adresse de la méthode en haut de la pile
GSB          ; appel du code de la méthode
POP          ; efface l'adresse de p de la pile
PRX          ; place le résultat de la méthode sur la pile
```

En fait, dans ce qui précède, on pourrait remplacer PUSH(0);CREAD par READ. Mais CREAD est nécessaire pour accéder aux autres méthodes. Par exemple pour `p.d();` il suffit adapter l'offset pour changer de méthode :

```
⟦p⟧          ; on place deux copies de l'adresse de p sur la pile
FETCH(0)
READ         ; on a maintenant l'adresse de la table d'adressage en haut de la pile
PUSH(1)      ; offset correspondant à la deuxième méthode
CREAD        ; adresse de la 2e méthode sur la pile
GSB          ; appel du code de la méthode
POP          ; efface l'adresse de p de la pile
PRX          ; place le résultat de la méthode sur la pile
```

Plus généralement, la compilation d'un appel de méthode `x.m();` où la classe de `x` est `C` sera donc :

```
⟦x⟧              ; place l'adresse de x en haut de la pile
FETCH(0)
READ             ; on a maintenant l'adresse de la table d'adressage en haut de la pile
PUSH(num(C,m)    ; l'offset correspondant à la méthode
CREAD            ; place l'adresse de la méthode en haut de la pile
GSB              ; appel du code de la méthode
POP              ; efface l'adresse de p de la pile
PRX              ; place le résultat de la méthode sur la pile
```

Remarquons que lorsque la méthode ne renvoie pas de résultat, l'instruction PRX sera omise.

6.9.6 Traitement des sous-classes

Regardons maintenant ce qui se passe pour la sous-classe `ColorPoint`. Sa définition contient le code de deux méthodes : la nouvelle méthode `color()` et la redéfinition de `d`. Ces deux fonctions vont être compilées comme les méthodes de `Point`. Appelons respectivement adr(ColorPoint, color) et adr(ColorPoint, d) les adresses où seront stockés ces codes.

La manière dont est construite la table d'adressage de `ColorPoint` est particulièrement importante :

- Tout d'abord, les méthodes qui sont héritées de la sur-classe `Point`, ou qui sont redéfinies, sont aux mêmes positions que dans la table de la sur-classe. C'est-à-dire que

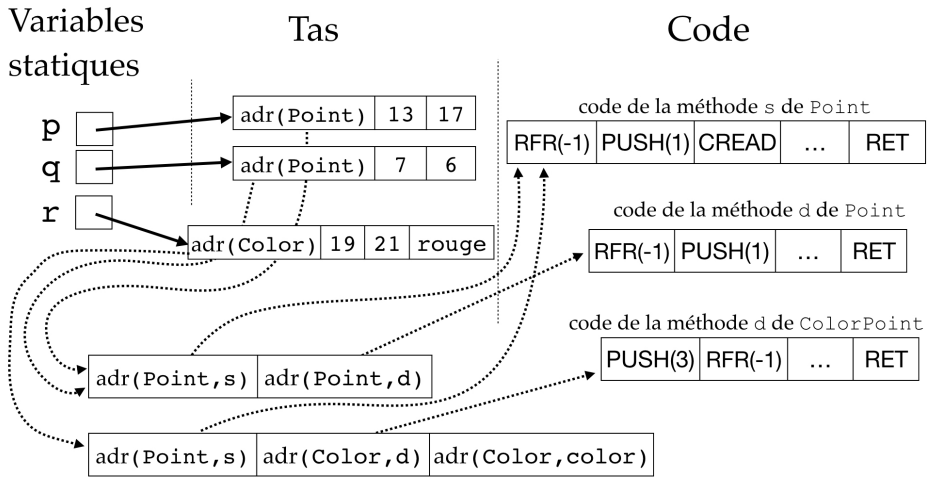
$$\begin{aligned} \text{num}(\text{ColorPoint}, \text{s}) &= 0 \\ \text{num}((\text{ColorPoint}, \text{d}) &= 1 \end{aligned}$$

- Les méthodes qui sont propres à la sous-classe (ici `color()`) viennent ensuite.
- Bien sûr, lorsque la méthode est héritée sans redéfinition, comme pour `s`, on pointe sur le même code. C'est-à-dire ici : adr(ColorPoint, s) = adr(Point, s).

Les objets instances de `ColorPoint` seront représentés de manière similaire aux instances de `Point`, en faisant attention aux points suivants :

- Les champs hérités de `Point` seront placés aux mêmes positions que pour les instances de `Point`;
- ensuite viendront les champs supplémentaires (ici `color`).

Aussi, `static ColorPoint r = new ColorPoint(19, 21, "rouge");` donnera lieu à la construction suivante en mémoire :



On voit que ce schéma de compilation respecte l'héritage : on peut compiler `q.d()` de la même manière que dans la section précédente, c'est-à-dire de la même manière pour toutes les instances de `Point`. Si `q` est une instance de `ColorPoint`, alors c'est le code de la méthode redéfinie qui sera exécuté.

On comprend aussi pourquoi Java interdit l'héritage multiple. Si une classe héritait de deux sur-classes, les tables d'adressages de celles-ci risqueraient de se chevaucher.

Solution page 190. **Exercice 6.9.1** Comment serait compilée la méthode `d()` de la classe `ColorPoint` si elle était re-définie par :

```
@Override
int d() { return(33 + super.d()); }
```

6.9.7 Méthodes avec arguments

La question de comment est compilé la passage de l'argument **`this`** est assez indépendante de l'héritage. On a en fait déjà décrit tous les mécanismes nécessaires.

```
class ExMeth {
    int x;
    int y;
    ExMeth(int a, int b) { x = a; y = b; }
    int f(int c) { return (x + y + c); }
}
```

La méthode `f(int c)` est compilée comme une fonction prenant deux arguments. L'argument explicite **`int`** `c` et un argument implicite **`this`** de classe `ExMeth`.

On considèrera que l'argument **`this`** est évalué avant les autres, donc que sa valeur est placée sous les autres dans la pile.

On compilera donc un code `p.f(e)` comme :

```
⟦p⟧          ; p est évalué et l'adresse placée sur la pile
⟦e⟧          ; la valeur de l'agument e est aussi placée sur la pile
FETCH(1)     ; on recopie l'adresse de p
READ         ; on la remplace par l'adresse de la table d'adressage
PUSH(0)      ; le numéro de la méthode dans la classe
CREAD        ; l'adresse du code de la méthode
GSB
POP          ; on efface les valeurs des argument c
POP          ; et this
PRX          ; on recopie le résultat de la méthode sur la pile
```

On laisse le code de la méthode comme exercice.

Chapitre 7

Erreurs et exceptions

7.1 Généralités et historique

Quasiment depuis l'origine, les langages de programmation sont munis d'instructions permettant d'interrompre l'exécution du programme lorsqu'une situation indésirable est détectée : lorsque le réseau ne répond pas, lorsque les données ne respectent pas le format prévu, lors d'une division par zéro etc. . .

Ces instructions ont ensuite évolué pour permettre un traitement plus fin des erreurs et situations exceptionnelles. Le langage LISP a été l'un des premiers à proposer de telles fonctionnalités dès les années 1960. Ada, vers 1980, puis Modula2+ ont proposé ensuite un ensemble de possibilités de traitement des exceptions qu'on retrouve dans beaucoup de langages modernes, dont Java ou Caml. Les trois points essentiels étant :

- La possibilité d'interrompre le déroulement habituel du programme en "lançant" (*throw*) une exception.
- La possibilité de traiter, ou rattraper (*catch*) une exception à l'interieur du programme.
- Enfin la possibilité d'associer une information à une exception, ce qui permet un traitement plus fin lorsqu'elle est rattrapée.

Ceci fait des exceptions un vrai outil de programmation qui va au-delà du simple traitement d'erreurs. Enfin, pour les langages typés (dont Java, Caml, C++) les exceptions sont traitées par le système de typage ; dans le cas de Java, le traitement des exceptions utilise les mécanismes de classes et d'héritage.

7.2 Déclencher une erreur

Une erreur correspond à une interruption irrémédiable du programme. Elles peuvent être explicitement déclenchées par le programmeur ; par exemple, voici une fonction qui cherche le maximum parmi les valeurs d'un tableau et échoue si le tableau est vide :

```
static int max(int[] t) {
  if (t.length == 0) throw new Error("tableau vide");
  int i = 0;
  for (int j = 1; j < t.length; j++)
      if (t[j] > t[i]) i = j;
  return(t[i]);
}
```

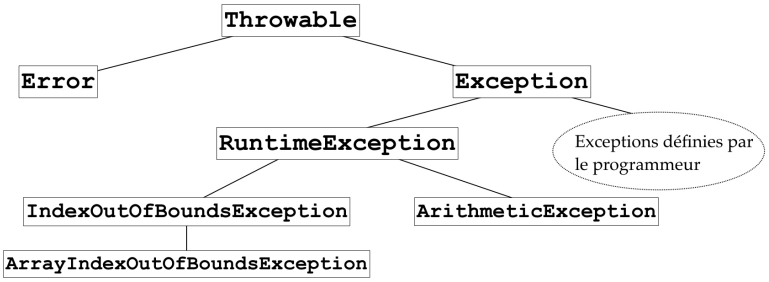


FIGURE 7.1 – Une partie de la hiérarchie des sous-classes de `Throwable`

Appliquer cette fonction `max` à un tableau vide interrompera l'exécution du programme avec une erreur munie de la chaîne de caractères `"tableau vide"`. On peut voir qu'on trouvera également quelques informations sur l'endroit du code où a été déclenchée l'erreur.

On voit que l'instruction pour déclencher ou "lancer" une erreur est **`throw`**. On voit également que cette instruction prend en argument un *objet* de la classe `Error` qui est typiquement, comme ici, fabriqué par un constructeur. Ici on utilise le constructeur `Error(String s)` qui permet d'ajouter une chaîne de caractères à l'erreur, ce qui va donner des informations utiles. Il est également possible d'utiliser un constructeur `Error()` qui ne rajoute pas d'information.

7.3 Erreurs standard

Un certain nombre d'opérations standard de Java peuvent aussi interrompre le programme. Par exemple une division par zéro déclenche une `ArithmeticException` munie de la chaîne de caractères `"/ by zero"` et l'accès à un tableau en dehors des bornes déclenche une : `ArrayIndexOutOfBoundsException`.

Formellement, on peut remarquer qu'il s'agit là d'objets de sous-classes de `Exception` et pas de `Error`. On éclaire cette petite différence ci-dessous.

7.4 Définir une exception spécifique

De manière générale, l'instruction **`throw e;`** est possible si `e` est une instance de la classe `Throwable`. Les classes `Error` et `Exception` sont toutes deux des sous-classes de `Throwable`. On donne en 7.1 une représentation d'une petite partie de la hiérarchie des sous-classes de `Throwable` disponible par défaut en Java.

Lorsque l'on veut utiliser une exception pour traiter un cas qui ne correspond pas à une erreur dans un programme, on va donc définir une sous-classe de `Exception` dédiée au cas traité.

Un exemple simple et classique est celui d'une fonction qui calcule le produit des éléments d'un tableau. L'idée est que si on rencontre un élément nul, on veut arrêter les calculs puisque le résultat est alors forcément zéro :

```
class ZeroFound extends Exception {}

static int prod (int[] t) {
   int r = 1;
```

```
   for (int i = 0; i < t.length; i++) {
      if (t[i] == 0) throw new ZeroFound();
      r = r * t[i];
   }
   return r;
}
```

7.5 Rattraper une exception

Le code de la fonction prod ci-dessus ne fonctionne pas encore. Lorsque l'exception est lancée il faut la traiter, ou la "rattraper" pour rendre alors 0 comme résultat final. La construction qui permet de rattraper et de traiter une exception est la suivante :

```
class ZeroFound extends Exception {}

static int prod (int[] t) {
   int r = 1;
   try {
      for (int i = 0; i < t.length; i++) {
         if (t[i] == 0) throw new ZeroFound();
         r = r * t[i];
       }
      return r;
   }
   catch (ZeroFound e) { return 0; }
}
```

Le forme générale est donc `try {p1} catch (E e) {p2}` où `E` est une sous-classe de `Throwable`. L'exécution d'un tel programme correspond à :

- L'exécution de `p1`. Si cette exécution ne lève pas d'erreur ou d'exception on en reste là et passe à la suite.
- Si en revanche l'exécution de `p1` est interrompue par un `throw e` où `e` appartient à `E` (ou donc à une sous-classe de `E`), alors on exécute `p2`.

Exercice 7.5.1 Que donne l'exécution du programme suivant ? Solution page 190.

```
class E1 extends Exception {}
class E2 extends E1 {}

class Exo {
    public static void main(String[] args) {
        try {
            try { throw new E1(); }
            catch (E1 e) { System.out.println("A"); }
            try { throw new E2(); }
            catch (Throwable e) { System.out.println("B"); }
            try { throw new E1(); }
            catch (E2 e) { System.out.println("C"); }
        }
        catch (Exception e) { System.out.println("D"); }
    }
}
```

7.6 Exceptions et types des fonctions

Le compilateur analyse statiquement le programme pour établir dans quelles parties du code une exception ou un `Throwable` peut être lancé. Lorsque l'exécution d'une fonction peut donner lieu au lancement d'une exception, il faut le signaler dans le type. Par exemple dans le code suivant, où il faut signaler que la fonction `main` est susceptible de donner lieu à une exception :

```
class E extends Exception {}
class Type {
    public static void main (String[] args) throws E {
        if (Math.random() < 0.5) throw new E();
        System.out.println("ouf !");
    }
}
```

En revanche, les `Error` potentielles ne sont pas signalées dans les types. Ne sont pas non plus signalées les `RuntimeException` ; ce serait fastidieux s'il fallait signaler, par exemple, que tout code utilisant un opérateur de division est susceptible de donner lieu à une erreur de division par 0.

On dit que les `RuntimeException` (et leurs sous-classes) sont des exceptions *non-vérifiées* ou *unchecked*.

Remarquons que les versions plus anciennes de Java ne demandent pas de signaler les exceptions potentielles dans les types des fonctions.

7.7 Traiter plusieurs exceptions

On peut mettre plusieurs `catch` autour d'un même programme pour traiter spécifiquement différentes exceptions. Par exemple le programme suivant utilise une fonction aléatoire pour afficher soit `bleu` soit `rouge`.

```
class E1 extends Exception {};
class E2 extends Exception {};

class Rand {
    public static void main(String[] args) {
        try {
            if (Math.random() < 0.5) throw new E1();
            throw new E2();
        }
        catch (E1 e) {System.out.println("bleu"); }
        catch (E2 e) {System.out.println("rouge"); }
    }
}
```

7.8 Associer de l'information à une exception

Si on veut qu'un objet de classe `Exception` contienne plus d'information, il suffit d'ajouter des champs lorsqu'on crée la classe. Le programme suivant se comporte comme le précédent mais en n'utilisant qu'une classe d'exceptions :

```
class Ec extends Exception {
```

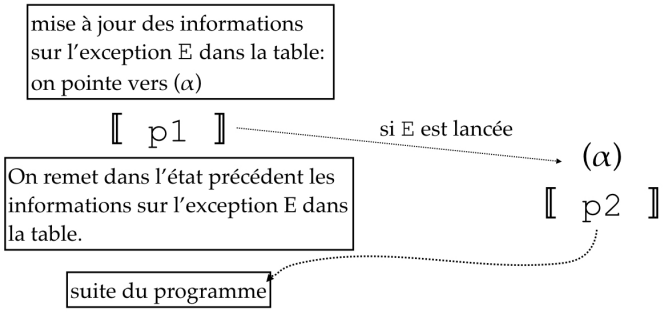


FIGURE 7.2 – Schéma du code compilé de `try {p1} catch (E e) {p2}`.

```
    String c;
    Ec(String s) { c = s; }
};

class Rand1 {
    public static void main(String[] args) {
        try {
            if (Math.random() < 0.5) throw new Ec("bleu");
            throw new Ec("rouge");
        }
        catch (Ec e) {System.out.println(e.c); }
    }
}
```

7.9 Comment sont compilées les exceptions

On ne va pas détailler le code machine correspondant aux constructions présentées dans ce chapitre, mais essayer d'en donner une idée. Une erreur ou une exception non-rattrapée interrompt simplement le programme. Plus intéressant est la compilation du mécanisme de `try ... catch` :

```
try { p1 }
catch (E e) { p2 }
```

Lorsque débute l'exécution du code `p1` la pile est dans un certain état qu'on note $\mathcal{S} = x; y; z; \ldots$ Si une exception `e` est lancée au cours de l'exécution de `p1`, la pile sera alors une *extension* de $\mathcal{S}$, c'est-à-dire quelque chose de la forme $x_0; x_1; \ldots; x; y; z \ldots$ Le code correspondant à `throw new E();` devra donc :

1. Remettre la pile dans son état $\mathcal{S}$,
2. puis effectuer un saut pour exécuter le code `p2`.

Le premier point est facile à effectuer puisqu'il suffit d'ôter une partie du haut de la pile, c'est-à-dire de réduire la valeur du pointeur SP. Il faut néanmoins se "souvenir", au moment du lancement de l'instruction, de la valeur à laquelle il faut décrémenter SP.

Une solution est de maintenir une table au cours de l'exécution, où pour chaque classe d'exceptions sont stockées les informations suivantes :

– est-ce que, à l'instant courant, il y a un mécanisme pour rattraper cette exception,

– si oui, alors à quelle valeur il faut décrémenter le compteur de pile si cette exception est lancée, et quelle est l'adresse du code qu'il faut exécuter ensuite.

Le code source `try { p1 } catch (E e) { p2}` est donc compilé comme suit :

– Des instructions qui mettent à jour les informations sur le traitement de `E` dans la table de traitement des exceptions (et qui vont notamment stocker la valeur courante de SP).

– Le code ⟦`p1`⟧.

– Des instructions pour remettre la table de traitement des exceptions dans son état précédent.

– Enfin, ailleurs, le code ⟦`p2`⟧.

On schématise cela dans la figure 7.2.

Ce qu'il faut retenir c'est que l'instruction **`throw`** et la construction **`catch`** peuvent être compilé vers du code n'effectuant que peu d'instructions et donc efficace.

Solution page 191. **Exercice 7.9.1** Pouvez-vous deviner ce que rend le code suivant :

```
class MonExc extends Exception {
    int val;
    MonExc(int n) { val = n; }
}

class Exc {
    static int f (int n) throws MonExc {
        if (n == 0) throw new MonExc(0);
        try {
            f(n - 1);
            return(0);
        }
        catch (MonExc e) { throw new MonExc(e.val + 1);}
    }

    public static void main (String[] args) {
        try {f(6);}
        catch (MonExc e) { System.out.println(e.val);}
    }
}
```

Chapitre 8

Architecture d'un compilateur

Dans ce cours nous expliquons les fonctionnalités de Java en montrant comment elles sont traduites en langage machine. En revanche, nous rentrons moins dans les détails de la construction du compilateur lui-même. L'écriture d'un compilateur est souvent considéré comme l'un des exercices les plus instructifs possibles pour un étudiant en informatique ; parce qu'on prend en compte toutes les caractéristiques à la fois du langage et de la machine, mais aussi parce qu'on utilise une large palette d'outils algorithmiques et de techniques de programmation. Un ouvrage de référence sur l'écriture de compilateurs est depuis plusieurs années le livre (Aho et al., 2007), souvent surnommé le *Dragon Book*, en référence à son illustration de couverture.

Dans ce chapitre on donne quelques éléments d'architecture de compilateur et on approfondit un point particulier, qui est la génération de code machine. Il y a deux raisons pour ce dernier choix :

- Cela permet de reprendre et de rendre concret la traduction du code source en code machine décrite dans les chapitres précédents,
- c'est un excellent exercice de programmation.

De fait, ce chapitre sert de support à un TD portant sur la compilation, où l'on va réaliser (en Java) la génération de code d'un compilateur pour un sous-ensemble de Java.

8.1 Le choix du langage et la possibilité du *Bootstrap*

Plus généralement, lorsque l'on réalise un compilateur pour un certain langage (disons Java) il faut choisir un langage dans lequel on écrira le compilateur lui-même. Le premier compilateur pour Java devait évidemment être écrit dans un autre langage pour lequel on disposait préalablement d'un compilateur, par exemple C. Mais une fois qu'on dispose d'un premier compilateur Java, on peut, en Java, écrire de nouveaux compilateurs Java plus puissants. Ces versions successives pourront produire du code plus efficace ou intégrer de nouvelles fonctionnalités. Un compilateur écrit dans le langage qu'il compile pourra donc se compiler lui-même. Cette action est généralement désignée par le terme anglais de *bootstrap*, qui repose sur la métaphore que l'on puisse décoller en tirant sur la boucle de ses chaussures ou bottes (*boots*).

A noter que le terme *to boot a computer* pour parler du processus de démarrage semble avoir la même origine.

Dans ce cours, nous ne décrivons la majorité des fonctionnalités de Java et nous nous intéressons à la compilation vers une machine simplifiée. Dans ce chapitre nous allons utiliser Java pour compiler un sous-ensemble nettement plus petit de Java. Cela nous permet d'utiliser

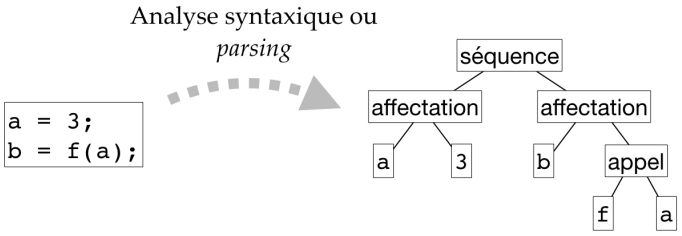
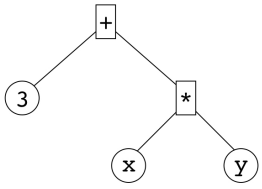


FIGURE 8.1 – La phase d'analyse syntaxique

des structures Java pour illustrer notre propos et à cette occasion présenter quelques techniques de programmation.

8.2 Analyse syntaxique

Le code source est un fichier texte, c'est-à-dire essentiellement une séquence de caractères. C'est donc une structure de données très simple et linéaire. En revanche, lorsque nous avons manipulé ou décrit les programmes sources dans les chapitres précédents, nous avons déjà indiqué que nous les voyons comme une structure *arborescente*. La suite de caractères est ce qu'on appelle la *syntaxe concrète* et la structure arborescente la *syntaxe abstraite*. Un exemple simple de syntaxe concrète d'expression Java est la suite de caractères `3 + x * y`. Celle-ci est transformée en un *arbre de syntaxe abstraite* qu'on peut graphiquement représenter comme ceci :



Cette première phase de transformation du code source s'appelle l'*analyse syntaxique*, ou *parsing* en anglais (figure 8.1). On peut remarquer que si l'analyse syntaxique transforme une séquence linéaire de caractères en une structure arborescente, le reste du processus de compilation va faire le contraire, à savoir "aplatir" cette structure arborescente en une séquence linéaire d'instructions machines.

L'analyse syntaxique est un champ algorithmique assez vaste, intéressant mais complexe et technique ; nous n'allons pas l'aborder ici. Le *Dragon Book* (Aho et al., 2007) est un bon point d'entrée pour qui veut s'initier à cette question.

Dans les sections suivantes, on décrit la syntaxe abstraite d'un sous-ensemble de Java qu'on appelera *While*. En particulier, on va présenter une technique pour représenter et traiter ces arbres de syntaxe en Java. Ces sections peuvent être lues pour elles-mêmes, mais elles peuvent aussi servir de base à un exercice de type TD, où le but sera de compléter le code et en particulier d'implémenter la génération de code machine. Pour ce TD, le code effectuant l'analyse syntaxique sera fourni, ainsi que l'émulateur permettant d'exécuter le code XVM généré.

8.3 Un sous-ensemble de Java : *While*

On va s'intéresser essentiellement à la compilation des expressions et des fonctions. Pour cela, on travaille sur un petit langage de programmation correspondant essentiellement à ce qui est décrit dans les chapitres 1 à 4, c'est-à-dire le noyau impératif et les fonctions. Plus précisément, notre mini-langage, qu'on appelera *While* a les caractéristiques suivantes :

- On a deux types de base, les entiers et les booléens.
- On ne manipule que des objets de ces types de base, donc pas d'enregistrements, d'objets, de champs, méthodes ou constructeurs, d'héritage...
- On n'a donc pas non plus de méthodes ; les fonctions sont toutes statiques.
- Un programme est une séquence de fonctions.
- On renonce aux variables globales ; les variables sont locales aux fonctions.

8.3.1 Les expressions

Une expression *While* peut être :

- Une constante numérique : 2, 3, 837...
- Une constante booléenne : `true` ou `false`.
- Une variable : `x`, `y`...
- un opérateur binaire appliqué à deux arguments (qui sont des expressions) : `x+3`, `2*(x+y)`...
- Un opérateur unaire avec un argument : `-x`...
- Une fonction avec des arguments : `f(4)`, `g(x, y+2, 4)`...

8.3.2 Les instructions

Les instructions de notre mini-langage pourront être :

- Une affectation de variable `a = e;` où `e` est une expression.
- Une instruction de test `if (e) i1 else i2` où `i1` et `i2` sont des instructions.
- Une boucle `while (e) i` où `i` est une instruction.
- Un bloc, c'est-à-dire une séquence d'instructions.
- L'instruction `return(e);` où `e` est une expression.
- L'instruction `return;` pour terminer une fonction ne rendant pas d'argument.

Remarquons qu'on autorisera les blocs formés par la séquence vide d'instruction. Cela permet des tests sans branche `else`. Plus précisément la syntaxe concrète :

```
if (x == 3) x = 0;
```

sera traduite vers la même syntaxe abstraite que :

```
if (x == 3)
    x = 0;
else { }
```

8.3.3 Les définitions de fonctions

Pour simplifier la représentation des définitions de fonctions, on imposera une syntaxe et une organisation un peu plus rigide que le vrai langage Java. On exige que les variables locales soient définies au début du corps de la fonction. Aussi, une fonction `f` dont le type de retour est `R` sera définie par quelque chose comme :

```
R f(A1 a1, A2 a2, ... , An an) {
  T1 x1 = e1;
  T2 x2 = e2;
  ...
  Tk xk = ek;

  i
}
```

Où :

- `a1`, ... `an` sont les arguments de `f` et `A1`, ... `An` leurs types respectifs.
- `x1`, ... `xk` sont les variables locales, `T1`, ... `Tk` leurs types et `e1`, ... `ek` leurs valeurs initiales respectives. Les variables locales doivent donc toutes être définies au début de la fonction.
- `i` est une instruction (en pratique, généralement, un bloc d'instructions) qui est le corps de la fonction.

Rappelons que les types comme `R`, `T1`, `T2`, `A1`... sont tous égaux soit à `int` soit à `boolean`.

Enfin un programme complet est une séquence de définitions de fonctions.

8.4 Une implémentation possible de la syntaxe abstraite

On voit donc que la syntaxe du programme, après analyse syntaxique, est représentée sous forme d'arbre. Il est pratique ici d'utiliser une représentation des arbres plus sophistiquée que celle décrite dans le chapitre 5 ; celle que nous allons présenter ici utilise l'héritage de manière cruciale. Remarquons que la technique décrite dans cette partie peut tout à fait être utile pour autre chose que la représentation de syntaxe abstraite.

On va représenter un programme source comme une instance d'une classe `Instruction` et chaque expression comme instance d'une autre classe `Expression`. Le premier point-clé est que ces deux classes sont des *classes abstraites*.

On devra déclarer dans ces classes les méthodes qu'on utilisera, en particulier pour la compilation, c'est-à-dire pour générer le code machine correspondant. Mais on verra plus bas comment typer ces méthodes.

Pour l'instant, regardons comment on crée les arbres de syntaxe.

8.4.1 Représentation des expressions

On a donc vu qu'une expression peut être une constante entière, une variable, l'appel à un opérateur ou à une fonction. Chacun de ces cas va correspondre à une sous-classe particulière de `Expression` :

- Une constante entière (comme `1515`) sera une instance de la classe concrète `IntExpression` qui est une sous-classe de `Expression`.

- Une expression formée par une variable sera une instance de la classe `VarExpression`.
- Une expression formée à partir d'un opérateur unaire (comme `-x`) sera une instance de la classe `UniOpExpression`.
- Une expression de formée avec un opérateur binaire (comme `x+3`) sera une instance de `BinOpExpression`.
- Une expression formée à partir d'un appel de fonction (comme `f(x, y)`) sera une instance de `FunExpression`.

Toutes ces classes sont donc des sous-classes concrètes de la classe abstraite `Expression`. Elles forment en fait les différentes manières de construire une `Expression`.

Par exemple le cas des constantes numériques et des variables :

```
abstract class Expression;

class IntExpression extends Expression {
  int value;

  IntExpression(int value) {
    this.value = value;
  }
}

class VarExpression extends Expression {
  String name;

  VarExpression(String name) {
    this.name = name;
  }
}
```

On donne en figure 8.2 les classes correspondant aux différents cas. On utilise des types énumérés pour les opérateurs unaires et binaires. Pour le dernier cas, celui des expressions construites à partir d'une fonction, on utilise un champ de classe `Vector<Expression>` pour la liste des arguments.

De fait, on utilise plusieurs fois dans ce chapitre la classe générique `Vector` qui est une implémentation de tableaux redimensionnables fournie par la bibliothèque Java et munie en particulier des méthodes suivantes :

```
boolean add(E e) // qui ajoute un élément à la fin du vecteur

int     size() // qui rend le nombre courant d'éléments du vecteur

E       elementAt(int i) // qui rend l'élément en position
                         // i  du vecteur

Vector<E>()   // constructeur qui fabrique un vecteur vide
```

On remarque que la classe `FunExpression` ainsi que les classes correspondants aux cas des opérateurs (binaires ou unaires) possèdent des champs de classes `Expression` qui correspondent aux sous-expressions. C'est ce qui fait que la classe `Expression` peut representer des arbres de profondeur quelconque (ou, autrement dit, qu'il s'agit bien d'une structure de donnée récursive).

Exercice 8.4.1 Donnez une expression Java qui construit un objet de la classe `Expression` qui corresponde à `2 + 3 * x`. ◇

Solution page 191.

```
abstract class Expression;

class IntExpression extends Expression {
  int value;

  IntExpression(int value) { this.value = value; }
}

class VarExpression extends Expression {
  String name;

  VarExpression(String name) { this.name = name; }
}

enum UniOp {
  MINUS
}

class UniOpExpression extends Expression {
  public UniOp uniop;
  public Expression term;

  UniOpExpression(UniOp uniop, Expression term) {
    this.uniop = uniop;
    this.term  = term;
  }
}

enum BinOp {
  PLUS, MINUS, TIMES, DIV
}

class BinOpExpression extends Expression {
  public BinOp binop;
  public Expression left;
  public Expression right;

  BinOpExpression(BinOp binop, Expression left, Expression right) {
    this.binop = binop;
    this.left  = left;
    this.right = right;
  }
}

class FunExpression extends Expression {
  public String fun;
  public Vector<Expression> args;

  ... // on ne détaille pas les constructeurs de FunExpression
}
```

FIGURE 8.2 – L'ossature de la classe représentant les expressions, sans les méthodes.

Si les définitions de classes ci-dessus donnent le principe de l'implémentation, on ne voudra pas les utiliser telles quelles. En effet, elles doivent être enrichies de méthodes qui correspondent aux fonctions fabriquant des données à partir des expressions.

Exercice 8.4.2 (recommandé) Equipez les éléments de la classe `Expression` d'une méthode `toString()` qui affiche l'expression de manière raisonnable. Vous pouvez afficher un peu trop de parenthèses. Remarquez que vous avez intérêt à équiper en fait les sous-classes concrètes de la classe abstraite `Expression`. ◇

Solution page 191.

L'exercice précédent est important car il montre comment opérer sur une structure arborescente lorsqu'elle est définie comme `Expression`, c'est-à-dire par différentes sous-classes. Le code de cette méthode est alors réparti sur ces sous-classes. On verra plus loin, que la technique dite des "visiteurs" permet de regrouper ce code.

8.4.2 Représentation des instructions

On utilise la même technique pour la classe représentant les instructions de programmes. Plus précisément, une classe `Instruction` représentera à la fois les instructions et les séquences ou bloc d'instructions. Cette classe sera une classe abstraite, comme la classe `Expression` ci-dessus. Les différentes instructions correspondront à différentes sous-classes concrètes de `Instruction`. Une instruction pourra être :

- Une affectation de variable `a = e;` qui sera représentée par une instance de la classe `IAssign`.
- Une instruction de test `if (e) i1 else i2` où `i1` et `i2` sont des instructions. On utilisera la classe `IIf` pour la représenter.
- Une boucle `while (e) i` où `i` est une instruction. Ceci sera représenté par une instance de la classe `IWhile`.
- Un bloc, c'est-à-dire une séquence d'instructions. On représentera ceci par un élément de la classe `IBlock`.
- l'instruction `return(e);` où `e` est une expression, est représentée par un élément de la classe `IReturn`.

Les classes `IAssign`, `IIf`, `IWhile` et `IBlock` sont donc toutes des classes concrètes et des sous-classes de `Instruction`.

Deux points techniques :

- On autorisera l'expression `e` dans une instance de `IReturn` à valoir `null`, ce qui permettra de représenter l'instruction `return;` (pour les fonctions qui de rendent pas de résultat).
- On autorisera également la variable, dans une instruction d'affectation à valoir `null`. Dans ce cas, l'instruction correspond à l'évaluation simple d'une expression.

8.4.3 Représentation des fonctions

On peut remarquer qu'on se passe ici de variables statiques (c'est-à-dire globales) mais il ne serait pas difficile d'en ajouter. De même, les fonctions sont en fait toutes statiques (puisqu'on ne traite pas d'objets) ; on omet donc le mot clé `static` dans la syntaxe concrète.

8.4.4 Représentation du code machine

Une instruction machine sera une instance de la classe abstraite `AsmInstruction` (pour *assembler instruction*) :

```
public abstract class AsmInstruction {}
```

Cette classe sera habitée par les instances de diverses sous-classes concrètes correspondant aux instructions de la xvm. Par exemple le cas d'une instruction sans argument comme ADD :

```
public final class ADD extends AsmInstruction {
}
```

ou une instruction comme WFR(i) qui contient comme argument l'*offset* i :

```
public final class WFR extends AsmInstruction {
  private int offset;

  public WFR(int offset) {
    this.offset = offset;
  }

  public int getOffset() {
    return this.offset;
  }
```

En pratique, on va vouloir afficher le code machine, et pour cela équiper ces classes avec une méthode `toString()` intelligible.

Si un programme source est une séquence de définitions de fonctions, un programme compilé est, en première approximation, une séquence d'instructions machines. Pour cela, on va pouvoir utiliser la classe `Vector<AsmInstruction>`. Il faudra également noter dans le code les emplacements, ou adresses, où se trouvent les différentes fonctions qui composent le programme. On va pour cela utiliser une table associant une adresse à une chaîne de caractères :

```
  labels Map<String, Integer>
```

L'interface `Map` comporte en particulier les tables de hachage `HashMap` de la bibliothèque (voir en 13.4 ou plus généralement le chapitre 13).

Dans cette table, les chaînes de caractères sont des étiquettes ou *labels*. On choisit dans Dans les instructions de saut, comme GTO, l'argument sera donné sous forme de chaîne de caractères, correspondant à une étiquette (*label*) :

```
public final class GTO extends AsmInstruction {
  private String address;

  public GTO(String address) {
    this.address = address;
  }

  public String getAddress() {
    return this.address;
  }

}
```

Remarque 8.4.1 Avoir choisi un champ `String` (et pas `int`) dans la classe `GTO` (et on fait pareil pour GTZ ou GSB) a pour conséquences :

1. qu'il faut consulter la table `labels` pour retrouver l'adresse physique vers laquelle on veut effectuer le saut,
2. que le code représenté correspond plutôt à ce qu'on appelle de *l'assembleur* que du vrai code machine. On obtiendrait le code machine proprement dit en remplaçant ces étiquettes (les `String`) par les valeurs numériques des adresses.

La raison de ce choix est que cela facilite à la fois l'écriture de la génération de code et la lecture code généré. ◇

8.5 Typage

Nous avons dit que nous considérons deux types : `int` et `boolean`. Après l'analyse syntaxique on devrait donc procéder à une vérification de typage. Nous ne décrivons pas du tout cette phase ici. De fait, elle n'est pas nécessaire : on peut compiler la syntaxe abstraite en supposant qu'elle corresponde à un programme bien typé.

Il nous faut pour cela toutefois renoncer à la surcharge : on interdira qu'il y ait deux fonctions portant le même nom. De même, lorsque, par exemple, on aura un appel de fonction comme `f(x, y)` dans le programme, on supposera que la fonction `f` est bien définie avec le bon nombre d'arguments (ici deux). Si on précise cela ici, c'est simplement parce que ceci est habituellement vérifié statiquement par le typage.

8.6 Une implémentation de la génération de code

Le point approfondi dans ce chapitre, et le travail demandé dans le TD, est la génération de code machine.

8.6.1 Une classe pour le code en train d'être généré

Pour construire le code machine à partir du code source, on a intérêt à regrouper le vecteur d'instructions et la table dans une classe, qu'on équipera de plus des méthodes permettant d'ajouter des instructions au fur et à mesure.

On se donne donc une classe `Code` dont les instances contiendront, en particulier :

- une séquence d'instructions machines (champ de classe `Vector<AsmInstruction>`),
- une méthode permettant d'ajouter une instruction à la fin de ce code :

  ```
  void pushInstr (ASMinstruction i)
  ```

- une *table* permettant de retrouver des adresses dans le code (champ `labels` de classe `Map<String, Integer>`),
- une méthode permettant d'ajouter dans la table une étiquette vers la fin courante du code :

  ```
  void pushLabel(String s)
  ```

- enfin une autre table indiquant, pour chaque variable, sa position dans la pile :

  ```
  Map<String , Integer> offsets
  ```

```
class Code {
    public Vector<AsmInstruction> code;
    public HashMap<String, Integer> labels;
    public HashMap<String, Integer> offsets;

    Code() {
        code = new Vector<AsmInstruction>();
        labels = new HashMap<String, Integer>();
        offsets = new HashMap<String, Integer>();
    }

    public void assInstr(ASMInstruction i) {
        code.add(i);
    }

    public void pushLabel(String s) {
        labels.put(s, code.size());
    }
}
```

FIGURE 8.3 – Une implémentation possible (et assez minimaliste) de la classe `Code`

Lorsqu'on appelle `pushLabel(s)`, l'étiquette pointe vers la prochaine instruction qui sera ajoutée au code. Par exemple, si on veut que l'objet `c` de classe `Code` corresponde à :

$$\begin{array}{ll} & \mathsf{PUSH}(0) \\ (\alpha) & \mathsf{PUSH}(1) \\ & \mathsf{ADD} \\ & \mathsf{GTO}(\alpha) \end{array}$$

On pourra utiliser le code Java suivant :

```
c = new Code();                 // créé un Code vide
c.pushInstr(new PUSH(0));
c.pushLabel("alpha");           // ajoute une étiquette vers
                                //  l'instruction suivante
c.pushInstr(new PUSH(1));
c.pushInstr(new ADD());
c.add(new GTO("alpha"));
```

On pourra vouloir ajouter quelques méthodes supplémentaires à `Code`, par exemple une méthode générant des noms "frais" de *labels* pour les sauts. Mais cela apparaitra, le cas échéant, au cours de l'écriture du générateur de code dont on parle maintenant.

8.6.2 Une méthode pour la génération de code

On veut maintenant générer du code machine à partir d'expressions et d'instructions. C'est à dire implémenter les fonctions $[\![\texttt{e}]\!]$ et $[\![\texttt{p}]\!]$ définies en particulier dans les chapitres 2 et 4. On va implémenter ces fonctions par des méthodes `codeGen` ajoutées aux classes `Expression` et `Instruction`. Ces méthodes vont prendre en argument l'objet de classe `Code` auquel le code correspondant à l'expression ou l'instruction sera ajouté.

Autrement dit, on veut que `e.codeGen(c)` (respectivement `p.codeGen(c)`) ajoute la suite d'instructions $[\![\texttt{e}]\!]$ (respectivement $[\![\texttt{p}]\!]$) au code `c`.

On commence par indiquer dans les classes abstraites la présence de ces méthodes :

```
abstract class Expression {
  public void codeGen(Code c) {
      throw new Error("codeGen not yet implemented");
    }
}

abstract class Instruction {
  public void codeGen(Code c) {
      throw new Error("codeGen not yet implemented");
    }
}
```

On aurait pu également laisser les méthodes abstraites. Ce qui est important, c'est de redéfinir ces méthodes dans chaque sous-classe concrète. Ces définitions sont l'objet principal de l'exercice de programmation correspondant à ce chapitre, nous n'allons donc pas les détailler ici. Voici néanmoins trois cas simples pour donner l'idée.

8.6.3 Un cas simple : expression constante

Le code compilé d'une expression constante entière comme `2001` est simplement l'instruction PUSH correspondante, ici PUSH(2001).

On implémente ce schéma de compilation par la méthode `codeGen` suivante pour la classe correspondante :

```
class IntExpression extends Expression {
  int value;
...
  @Override
  public void codeGen(Code c) {
    c.pushInstr(new PUSH(this.value));
  }
}
```

8.6.4 Retour de fonction

On l'a dit, les instructions `return(e);` et `return;` correspondent à des instances de la classe `IReturn` et dans le second cas, le champ de l'objet de retour vaut `null`.

Traitons ce cas de l'instruction `return;`. On se rappelle (voir 4.5) que cette instruction Java est simplement compilée vers l'instruction machine RET. On peut donc proposer le code suivant :

```
class IReturn extends Instruction {
  Expression resultat;
  IReturn(Expression e) { resultat = e; }

  @Override
  void codeGen(Code c) {
    if (this.resultat == null)  c.pushInstr(new RET());
    else
     {
          ...  // ici on traite le cas d'un return rendant un résultat
     }
   }
 }
```

Le code manquant correspondra donc au schéma de compilation décrit en 4.6 qui permet à la fonction de rendre un résultat, c'est-à-dire compiler l'instruction `return(e);` par :

⟦e⟧
PXR
RET

8.6.5 Variables

Dans le langage considéré, une variable x fait soit référence à un argument de fonction, soit à une variable locale. Dans les deux cas, la valeur de la variable est rangée dans la pile. On va donc considérer qu'au moment de la compilation, on connait la position de la variable par rapport au *frame pointer* (l'*offset* de la variable). Ce qui donne un schéma de compilation très simple :

```
class VarExpression extends Expression {
  String name;
  public String toString() { return(name); }
  void codeGen(Code c) {
    c.pushInstr(new RFR(c.offsets.get(this.name)));
  }
}
```

Il faudra évidemment faire attention à mettre à jour la table `offsets` de `c` dans deux cas :

- lors de l'appel d'une fonction avec arguments,
- au début du corps d'une fonction, pour les variables locales.

8.7 Génération de code par visiteurs

Dans ce qui précède, la génération de code machine est effectué par la méthode `codeGen` et on voit que le code (Java) de cette méthode est dispersé dans les différentes sous-classes de `Expression` et `Instruction`. On peut vouloir chercher à regrouper toute la génération de code dans une seule classe, sans changer le reste du compilateur, c'est-à-dire en particulier :

- L'architecture des classes `Expression` et `Instruction`,
- la représentation du code machine et en particulier la classe `Code`.

Une possibilité pour cela est d'utiliser le *patron de conception* ou *design pattern* des visiteurs. Cette technique s'applique typiquement pour implémenter des fonctions agissant sur des structures comme `Expression` et `Instruction`.

Ce schéma ou patron des visiteurs a été présenté en (Gamma et al., 1994). On peut en trouver une description en ligne en (McDonald, 2007) (ainsi que d'autres *design patterns*). Le principe est donc de regrouper le code d'une fonction comme `codeGen` dans une classe de visiteurs qui peut traiter les structures en question. Prenons le cas de la structure `Expression`.

On commence par indiquer au niveau de la classe abstraite que celle-ci peut être visitée :

```
abstract class Expression {
...
  abstract void accept(ExprVisitor v);
}
```

L'interface `ExprVisitor` définit les composantes d'un visiteur pour `Expression` :

```
interface ExprVisitor {
  public void visit(IntExpression e);
  public void visit(UniOpExpression e);
  public void visit(BinOpExpression e);
  public void visit(VarExpression e);
  public void visit(FunExpression e);
}
```

Il faut ensuite, décliner comment s'effectue la visite dans chaque sous-classe. On fait cela en étendant les définitions des sous-classes de manière très uniforme :

```
class IntExpression extends Expression {
  int value;
...
  void accept(ExprVisitor v) {
    v.visit(this);
  }
}

class UniOpExpression extends Expression {
  int value;
...
  void accept(ExprVisitor v) {
    v.visit(this);
  }
}

class BinOpExpression extends Expression {
  int value;
...
  void accept(ExprVisitor v) {
    v.visit(this);
  }
}
...
```

Il faut remarquer qu'on utilise ici la surcharge et pas l'héritage. C'est statiquement que le compilateur détermine que dans le cas de `IntExpression`, `v.visit` désigne la première composante du visiteur `v` car **`this`** est de classe `IntExpression`.

Jusqu'ici, tout ce qu'on a fait est complètement générique pour la classe `Expression` et ne traite pas encore de la génération de code. On peut maintenant définir la génération de code comme un visiteur particulier. On ne donne ci-dessous que les parties qui correspondent à ce qu'on a déjà précisément décrit dans la section précédente.

```
class ExprCodeGen implements ExprVisitor {
  Code c;
  ExprCodeGen(Code c) {
    this.c = c;
  }
  visit(IntExpression e) {
    c.pushInstr(new PUSH(e.value));
  }
  visit(VarExpression e) {
    c.pushInstr(new RFR(c.offset.get(e.name)));
  }
...
}
```

Pour compiler vers un objet `c` de classe `Code` on commencera donc par créer un générateur :

```
ExprCodeGen g = new CodeGen(c);
```

puis, si l'on veut ajouter à `c` le code compilé d'une expression `e` il suffira de faire :

```
g.visit(e);
```

On procèdera de même pour les instructions, en créant une interface `InstrVisitor` de visiteurs dédiés à `Instruction`, en ajoutant aux sous-classes de `Instruction` les méthodes `accept` génériques, puis en construisant une classe implémentant `InstrVisitor` par la génération de code.

8.8 Optimisations

On a remarqué depuis longtemps que, une fois le schéma général de compilation établi, il était extrêmement utile d'effectuer un certain nombres d'optimisations sur le code produit. Il s'agit en général de repérer des situations où un certain code peut être remplacé par un code équivalent mais plus efficace.

De manière générale, trouver de nouvelles possibilités d'optimisation reste un domaine actif de recherche. Même si, prises une à une, les optimisations semblent relativement anodines, l'ensemble des optimisations effectuées par un compilateur modernes peuvent permettre une accélération par un facteur important.

On a déjà donné deux exemples d'optimisations possibles avec les boucles `do ... while;` (exercice 2.4.1) et les *appels terminaux* décrits en 4.10. Voici quelques exemples d'optimisations simples possibles, données sous forme d'exercices. Remarquez bien que nous n'essayons pas du tout ici d'être exhaustifs, mais simplement de donner une idée de ce qui peut être fait.

Solution page 191. **Exercice 8.8.1** On considère une optimisation sur le code machine. Dans le code suivant, on peut remplacer l'instruction GTO(α) par GTO(β).

```
        ...
        GTO(α)
        ...
(α)     GTO(β)
        ...
(β)     ...
```

Pourquoi a-t-on intérêt à faire cette optimisation ?

Donnez un exemple simple de code source qui produirait un tel code machine. Ce code source pourra, par exemple, utiliser des constructions `if (e) p1 else p2 .` ◇

Certaines optimisations doivent être détectées avant la génération du code machine. Par exemple :

Solution page 192. **Exercice 8.8.2** Une classe Java comporte la ligne suivante :

```
static final int c = 7125;
```

Comment sera compilée l'expression `c` dans le reste de la classe ? Proposez une optimisation de ce code. ◇

Solution page 192. **Exercice 8.8.3** On veut optimiser la compilation de fonctions de la forme :

```
f(int x) {
  if (e1) return e2;
  p;
}
```

On vous propose l'optimisation suivante :

- sortir le test `if (e1) return e2` du corps de la fonction,
- compiler les appels `f(e)` par un code machine qui va effectuer le test et n'exécuter GSB que lorsque qu'il est nécessaire d'exécuter `p`.

On prend l'exemple de la définition habituelle de la factorielle :

```
static int fact(int n) {
  if (n == 0) return 1;
  return(n * fact(n - 1));
}
```

Comment pourrait être compilée l'expression `fact(e)` avec une telle optimisation ? ◇

Un certain nombre d'optimisations importantes exploitent l'architecture du processeur, et ne peuvent être décrites dans le cadre d'une machine aussi simple que la XVM. Par exemple on peut compiler certaines fonctions de manière à ce que les arguments (et/ou le résultat) ne soient pas passés au code de la fonction par la pile (et donc en utilisant la mémoire) mais juste par des registres du processeur. D'autres optimisations très importantes vont exploiter le mécanisme de cache du processeur (mentionné en 2.6.5) encore une fois pour limiter le temps consacré au accès mémoire. Une famille particulièrement importantes d'optimisations exploite les possibilités de *pipeline* de la machine, c'est-à-dire que lorsque des opérations sont indépendantes l'une de l'autre, il est possible de commencer la seconde, voire la troisième ou plus, avant que la première ne soit achevée. Cela conduit à reconnaitre de tels ensembles d'instructions indépendantes, et éventuellement de les faire apparaître en ré-ordonnant les instructions du code machine.

De manière générale, les optimisations comme celles portant sur le cache ou le pipelining sont particulièrement efficaces. Elles sont aussi plus faciles à effectuer pour un programme que pour un programmeur humain. C'est la raison pour laquelle on ne programme quasiment plus directement en langage machine : le code produit par un bon compilateur sera plus rapide que celui écrit "à la main" en assembleur.

Chapitre 9

Mécanismes pour la modularité

On décrit dans ce bref chapitre deux familles de mécanismes qui contribuent à la modularité des programmes Java. Ces deux mécanismes sont liés à une caractéristique importante, la *compilation séparée*, c'est-à-dire la possibilité de compiler séparément deux classes, et néanmoins d'utiliser l'une dans l'autre. C'est-à-dire que les deux fichiers compilés (suffixe `.class`) sont assemblés, ou liés *(linked)*, avant l'exécution.

Nous ne cherchons pas à rendre compte de ces mécanismes de compilation séparée et de liage dans notre schéma de compilation et de machine abstraite. C'est pourquoi on va décrire de manière assez classique les deux mécanismes en question : les paquetages ou *packages* d'une part, et les mécanismes pour paramétrer du code d'autre part.

9.1 Les paquetages

9.1.1 Principes

La question principale à laquelle répond le mécanisme de *packages* est la gestion de l'espace des noms de classes. C'est-à-dire, essentiellement, lorsque dans une classe `A` on fait référence à une classe `B`, où le compilateur trouvera-t-il cette classe `B`. Il est nécessaire d'organiser l'espace des classes, ne serait-ce que pour éviter des collisions de noms. On peut tout à fait imaginer que deux efforts de développements de bibliothèques logicielles soient amenés à créer chacun une classe `Node` par exemple. Ces deux classes n'ont rien en commun sauf le nom ; mais si on ne pouvait les désigner séparément, on ne pourrait pas utiliser conjointement les deux bibliothèques.

Le mécanisme des paquetages répond à ce besoin d'organisation. Un paquetage contient une ou plusieurs classes ou interfaces. De plus, ces paquetages sont eux-mêmes organisés suivant une structure d'arbre ; cet arbre reflète l'arborescence des répertoires et fichiers où le compilateur ira chercher les classes.

A la racine de la hierarchie, on trouve un nœud `java`, et sous lui, plusieurs nœuds dont `java.lang`, `java.io` ou `java.util`. L'ensemble de la bibliothèque fournie avec le langage est dans cette hiérarchie. Par exemple les classes `Object` ou `Exception` dont nous avons parlé (resp. chapitres 6 et 7) sont en fait placées dans `java.lang`.

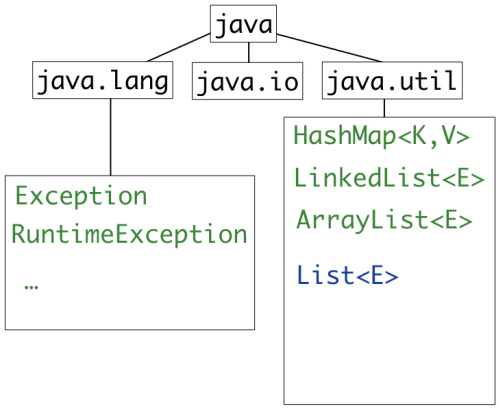


FIGURE 9.1 – Une toute petite partie de la hiérarchie des *packages* de la bibliothèque Java.

9.1.2 Noms qualifiés

On peut désigner une classe de la hiérarchie par son *nom qualifié* qui comprend la position du paquetage dans la hiérarchie. Par exemple la nom qualifié de `Object` est `java.lang.Object`.

On peut composer le nom qualifié avec la notation pointée pour accéder aux composantes d'une classe ou d'un objet. Par exemple la classe `java.lang.Math` contient de nombreuses fonctions (statiques) mathématiques, comme le cosinus. On peut y accéder par le nom qualifié `java.lang.Math.cos(a)` (où `a` est un objet de classe `double`). On comprend que les deux premiers points correspondent au chemin du paquetage et le troisième est l'accès à l'élément de la classe.

9.1.3 Noms abrégés

Les noms qualifiés sont précis mais longs. On peut les abréger en commençant par importer un paquetage. Par exemple, la classe `java.lang.Math` sera visible depuis une classe débutant par la ligne :

```
import java.lang.Math;
```

On peut alors accéder à la fonction cosinus par : `Math.cos(a)`.

On peut également importer toute une partie de la hiérarchie en utilisant `*`. Par exemple `import java.lang.*;` va importer tous les éléments de `java.lang` (dont `java.lang.Math` par exemple).

9.1.4 Déclaration d'un paquetage

On déclare l'appartenance à un *package* appelé `nomPack` par la ligne `package nomPack;` en début de fichier, avant les importations et la ou les déclaration(s) de classe(s). En général, les environnements comme Eclipse proposent une fonctionnalité pour créer un paquetage et regrouper les classes y appartenant.

Si on ne déclare pas d'appartenance à un paquetage dans un fichier `.java`, la classe appartient par défaut à un *package* anonyme. La manière dont cela est toléré peut varier légèrement entre les versions récentes de Java.

Un fichier `.java` débute donc typiquement par :

1. La déclaration de *package*,
2. un certain nombre de `import`,
3. la définition de la classe elle-même (`class NomClasse { ... }`).

9.1.5 Visibilité dans les paquetages

Lorsqu'on construit une bibliothèque logicielle, on retombe sur des questions de visibilité et d'encapsulation. On peut vouloir mettre à disposition une classe `A`. Pour implémenter cette classe `A` on pourra vouloir construire une classe `B`, mais cette dernière correspondra à un choix d'implémentation particulier, et on ne voudra pas que les utilisateurs de `A` aient accès à `B`. Pour cela, les classes composantes d'un paquetage peuvent elles aussi être publiques ou privées. Cette visibilité est déclarée lorsque l'on nomme la classe :

```
package nomPack;

public class A { ... }
```

Si, dans un paquetage, on ne déclare pas une classe publique, elle ne sera visible que depuis les classes appartenant au même paquetage ; ce sera le cas de la classe `B` suivante :

```
package nomPack;

class B { ... }
```

Enfin une classe peut-être déclarée comme explicitement privée, mais ce sera le cas pour une classe définie à l'intérieur d'une autre. Ici, la classe `B` ne sera visible qu'à l'intérieur de la classe `A` :

```
package nomPack;

public class A {
   private class B { ... }
 ... }
```

Par ailleurs, on peut restreindre la visibilité des éléments d'une classe (champs et méthodes) au paquetage de la classe. C'est pour cela que Java dispose de plus de deux niveaux de visibilité :

- Dans une classe publique, un élément déclaré explicitement par `public` est réellement public et sera visible à l'extérieur du paquetage.
- Un élément déclaré `protected` sera visible dans toutes les classes appartenant au même paquetage et dans toute sous-classe de la classe déclarante (même si la sous-classe est dans un autre paquetage).
- Un élément déclaré sans mot-clé sera visible exactement dans les classes du même paquetage.
- Un élément déclaré avec `private` ne sera visible que dans la classe elle-même.

9.2 Paramétrage de code par des classes

9.2.1 Fonction paramétrée par une classe

On a déjà vu qu'on pouvait paramétrer une classe par une autre ; c'est le mécanisme de classes génériques (en 5.11). On peut également vouloir paramétrer une fonction par une classe. Par exemple voici une fonction qui prend deux tableaux en arguments et compare leur taille :

```
static <E> boolean comp(E[] t, E[] u) {
    return (t.length < u.length);
}
```

Remarquez que :

- On utilise une syntaxe similaire aux classes génériques avec `<E>`, mais cette fois c'est juste une fonction qui est paramétrée et utilisable avec toute classe `E`.
- On n'a pas de problème de tableau générique ici (contrairement aux classes génériques), cette définition est utilisable.
- On peut utiliser cette fonction avec des arguments de type `Integer[]`, `String[]`, *etc...* En revanche on ne peut pas l'utiliser avec un tableau `int[]` puisque `int` n'est pas une classe.
- On ne peut pas utiliser cette fonction avec deux tableaux contenant des éléments de classes différentes. Pour cela, il faudrait la déclarer avec un type plus général :

```
 static <E, F> boolean comp(E[] t, F[] u) {
return (t.length < u.length);
}
```

9.2.2 Wildcards

Prenons un autre exemple de fonction paramétrée. Celle-ci va afficher les éléments d'une liste, et est paramétrée par la classe des éléments de la liste :

```
static <E> void printList(List<E> l) {
   for (Object e : l) System.out.println(e);  }
```

On peut remarquer qu'ici, la classe `E` n'est jamais utilisée dans la définition de la fonction. On peut du coup reformuler celle-ci en utilisant une *wildcard* :

```
static  void printList(List<?> l) {
    for (Object e : l) System.out.println(e);  }
```

Le mot anglais *wildcard* désigne une carte pouvant prendre n'importe quelle valeur (en français on utilise souvent un autre mot anglais, *joker*). On comprend que c'est le rôle du point d'interrogation dans la syntaxe ci-dessus.

9.2.3 Paramétrage borné

Lorsqu'une fonction ou une classe est paramétrée par une classe, on peut vouloir ajouter des contraintes sur ce paramètre. Supposons qu'une interface spécifie qu'une classe est munie d'une méthode qui permette de projeter les éléments vers `int` :

```
interface ToInt { int toInt(); }
```

On peut alors définir une fonction qui va additionner les valeurs obtenues par cette méthode pour tous les éléments d'une liste, à condition que leur classe soit une implémentation de `ToInt`. Une possibilité est de le faire en utilisant la syntaxe des wildcards :

```
void sumList(List<? extends ToInt> l) {
     int r = 0;
     for (Object e : l) r += e.toInt();
     return(r);
   }
```

Remarquez qu'ici on utilise le mot-clé `extends` bien que `ToInt` soit une interface. On pourrait faire de même si c'était une classe (abstraite ou concrète).

On peut de manière équivalent utiliser la syntaxe plus explicite :

```
<E extends ToInt> void sumList(List<E> l) {
    int r = 0;
    for (Object e : l) r += e.toInt();
    return(r);
  }
```

Cette possibilité de restreindre le paramétrage est importante dans certains cas. Supposons que nous ayons deux classes concrètes, dont l'une est une sous-classe de l'autre ; comme dans l'exemple classique :

```
class Point { double x; double y; ... }
class ColorPoint extends Point { String color; ... }
```

Un point important est qu'alors, `List<ColorPoint>` n'est pas une sous-classe de `List<Point>`, et ce même si on peut clairement voir une liste de `ColorPoint` également comme une liste de `Point`.

Aussi, si l'on a une fonction opérant sur une liste de `Point`, on aura intérêt à lui donner un type plus général :

```
static void afficheListe(List<? extends Point> l) {
  for (Point p : l) affiche(p);
}
```

On pourra ainsi appliquer la fonction `afficheListe` également à un argument de classe `List<ColorPoint>`.

Dans certains cas, on voudra effectuer borner les paramètres dans l'autre sens. Supposons qu'on dispose d'un `ColorPoint p` et que l'on veuille ajouter ce point à une liste. La manière la plus générale pour typer la fonction effectuant cette opération sera de dire qu'elle prend en argument une liste dont les éléments sont une sur-classe de `ColorPoint` :

```
static <E super Point> List<E> ajouteP(List<E> l) {
  return (l.add(p));
}
```

9.2.4 Classes génériques bornées

On peut également utiliser les quantifications bornées par `extends` ou `super` pour restreindre les paramètres d'une classe générique. Par exemple si on commence une définition par :

```
class <E extends ToInt> A {
```

on peut alors, dans la définition de `A` supposer que la classe `E` est bien une implémentation de `ToInt`, c'est-à-dire que les objets de classe `E` sont bien munis d'une méthode `toInt()`.

Deuxième partie

Compléments algorithmiques

Chapitre 10

Union-find

On présente ici une structure informatique et algorithmique classique, connue sous son nom anglais *union-find* ; on parle aussi de *disjoint set data structure*.

Les applications de cette structure sont nombreuses. De manière très synthétique, elle permet de déterminer si deux sommets d'un graphe appartiennent à la même composante connexe - mais sans proposer de chemin entre ces sommets ; ce deuxième problème étant l'objet d'algorithmes de parcours de graphes. Plus pratiquement, elle permet, par exemple, de construire les zones connexes de même couleur dans une image par exemple.

Par ailleurs, c'est un joli problème algorithmique et son implémentation ne nécessite que des tableaux et quelques fonctions.

10.1 Le problème

On considère un ensemble fini. On simplifie en se ramenant à l'ensemble des n premiers entiers naturels : $0, 1, \ldots, n-1$. Sur cet ensemble, on considère une relation d'équivalence qui va évoluer avec le temps.

On dispose de deux opérations :

- L'opération *union* consiste à relier deux objets, c'est-à-dire à unir les classes d'équivalences correspondantes. On écrira union(i, j) pour l'opération d'union entre les classes d'équivalence de i et j.
- L'opération *find* ne change pas la relation d'équivalence. Il s'agit d'une requête pour vérifier si deux objets appartiennent à la même classes d'équivalence ou non.

En général, et en particulier dans les approches décrites ici, on va identifier les classes d'équivalences avec un *représentant canonique*. On appelera donc find(i) le représentant canonique de i. Pour savoir si i et j appartiennent à la même classe d'équivalence, il suffit de vérifier si find(i) et find(j) sont égaux.

Exemple

On travaille sur l'ensemble $\{0; 1; \ldots; 9\}$. On part de la relation minimale (chaque élément n'est en relation qu'avec lui-même) et on effectue dans un premier temps les opérations suivantes : union$(2, 1)$, union$(1, 3)$, union$(7, 4)$ et union$(6, 9)$. Dans un deuxième temps union$(5, 6)$,

FIGURE 10.1 – Deux étapes de union-find

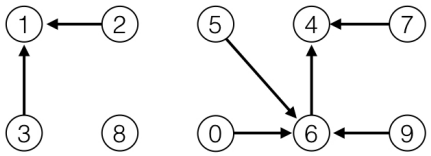


FIGURE 10.2 – Représentation dans l'approche paresseuse

union(0, 6) et union(6, 4). On a aux deux étapes les relations d'équivalences dessinées en pointillés dans la figure 10.1. Le choix précis du représentant canonique est laissé à l'implémentation. Tout juste sait-on qu'à la fin, par exemple, find(0) = find(9) ou find(3) ≠ find(4).

10.2 Implémentation naïve : quick-find

Dans toutes les implémentations que nous présenterons, la structure principale sera un tableau indicé par les éléments considérés, c'est-à-dire les entiers `0 ... n-1`. La première idée est extrêmement simple : dans le tableau, à l'indice i, on trouvera le représentant canonique de i. On peut appeler `rep` ce tableau.

On initialise le tableau par :

```
for (int i = 0; i++; i < rep.length) rep[i] = i;
```

La fonction find est donc on ne peut plus simple et rapide :

```
static int find(int i) { return(rep[i]) }
```

La fonction union(i, j) doit, elle, parcourir le tableau pour remplacer toutes les occurrences de find(i) par find(j) (l'inverse convenant aussi) :

```
static void union(int i, int j) {
   int ri = rep[i];
  for (int k = 0; k++; k < rep.length)
           if (rep[k] == ri) rep[k] = rep[j];
}
```

Dans les deux cas, le temps de calcul est facile à établir. Pour find(i) le temps est évidemment constant (donc asymptotiquement $O(1)$). Pour union(i, j) il faut parcourir le tableau, et donc le temps de calcul asymptotique est $O(n)$. C'est ce second temps qu'il s'agit d'améliorer.

10.3 L'approche paresseuse

On va cette fois utiliser le tableau pour faire pointer i pas forcément vers le représentant canonique, mais simplement un autre représentant de la classe. En fait, on va avoir une

représentation en arbre des classes d'équivalence, où chaque élément pointe vers son père. On choisit du coup d'appeler `pere` le tableau. Cela revient à orienter les arêtes des diagrammes, comme dans la figure 10.2 qui correspond au tableau suivant :

i	0	1	2	3	4	5	6	7	8	9
pere[i]	6	1	1	1	4	6	6	4	8	6

L'implémentation reste assez simple. L'initialisation est similaire à précédemment :

```
for (int i = 0; i++; i < pere.length) pere[i] = i;
```

On voit qu'ensuite que les représentants canoniques des classes d'équivalence sont les `i` tels que `pere[i]==i`.

On a donc l'implémentation suivante de `find` :

```
static int find(int i) {
  while (i != pere[i]) i = pere[i];
  return(i);
}
```

Pour `union` on fait pointer un des deux représentants canoniques vers l'autre. Dans un sens ou l'autre, on a le choix :

```
static void union(int i, int j) {
  i = find(i);
  pere[i] = find(j);
}
```

On voit qu'alors la complexité asymptotique des deux fonctions est la même : c'est la profondeur des arbres. En général, cette complexité est bornée par n. À ce stade on n'a donc pas amélioré la complexité, mais il nous suffit de limiter la profondeur des arbres pour cela.

10.4 Optimisation basée sur le poids

Le poids d'un arbre, donc d'une composante connexe, est simplement le nombre de ses éléments. On va jouer sur la liberté qui nous est donnée dans l'implementation précédente en faisant pointer systématiquement l'arbre de poids faible vers l'arbre de poids fort.

On utilise un tableau supplémentaire `poids` pour garder l'information de taille des arbres. Plus précisément, si `i` est le représentant canonique de la classe d'équivalence, alors `poids[i]` est le cardinal de la classe d'équivalence ; pour les autres indices, la valeur de `poids` n'est plus utilisée et est sans importance. Ce tableau est initialisé par :

```
for (int i = 0; i++; i < poids.length) poids[i] = 1;
```

La fonction `find` est inchangée. En revanche la fonction `union` devient :

```
static void union(int i, int j) {
  i = find(i);
  j = find(j);
  if (poids[i] < poids[j]) {
      pere[i] = j;
      poids[j] += poids[i];
   }
   else {
      pere[j] = i;
      poids[i] += poids[j];
```

```
    }
}
```

On peut alors montrer que la profondeur des arbres est majorée par $\ln_2(n)$. Plus précisément, l'astuce consiste à trouver le bon invariant. Etant donné un indice i, on appelle $|i|$ le cardinal de la classe d'équivalence de i, et $h(i)$ la hauteur de l'arbre auquel appartient i. On montre que :

Lemme 10.4.1 Après toute séquence d'opérations `union`, pour tout indice i, on a : $h(i) \leq \ln_2(|i|)$.$\diamond$

Démonstration Avant d'effectuer union(i, j), on a $h(i) \leq \ln_2(|i|)$ et $h(j) \leq \ln_2(|j|)$. Prenons le cas où $|i| \leq |j|$. Dans ce cas, on va faire pointer l'arbre de i vers l'arbre de j. la hauteur de l'arbre obtenu est donc soit $h(i) + 1$ soit $h(j)$. Dans le second cas, on a évidemment $h(j) \leq (ln_2(|i| + |j|)$. Dans le premier cas on a :

$$h(i) + 1 \leq ln_2(|i|) + 1 \leq ln_2(2|i|) \leq ln_2(|i| + |j|).$$

Le corollaire est qu'à chaque étape, le temps d'exécution d'une opération union ou find est borné par $O(\ln(n))$.

10.5 Optimisation basée sur la hauteur

On peut alternativement choisir de faire pointer l'arbre de plus faible hauteur vers le plus haut. La hauteur étant la longueur de la plus longue branche. On va à nouveau utiliser le tableau `poids`, mais cette fois il contient la hauteur de l'arbre.

Il suffit de changer le code de la fonction `union` par :

```
static void union(int i, int j) {
  i = find(i);
  j = find(j);
  if (poids[i] < poids[j]) {
      pere[i] = j;
   }
   else {
      pere[j] = i;
      if (poids[i] == poids[j])
          poids[i]++;
   }
}
```

On va à nouveau montrer le même lemme que précédemment.

Lors d'une opération union(i, j), le seul cas où l'arbre augmente de taille est lorsque $h(i) = h(j)$. Prenons le cas où $|i| \leq |j|$ (l'autre cas étant symétrique). La hauteur du nouvel arbre est alors $h(i) + 1$ et on a :

$$h(i) + 1 \leq \ln_2(|i|) + 1 \leq ln_2(2|i|) \leq ln_2(|i| + |j|)$$

ce qui est bien l'inégalité recherchée.

Bien que les complexités asymptotiques soit les mêmes, il semble qu'en pratique l'optimisation basée sur la hauteur donne de meilleurs résultats que l'optimisation basée sur le poids.

10.6 Compression de chemin

On présente enfin une autre amélioration, utilisable seule ou en combinaison avec l'une des deux optimisations précédentes. En pratique, cette nouvelle optimisation est la plus efficace ; en revanche le calcul de complexité est largement trop difficile pour être traité dans ce cours.

Il s'agit tout simplement de modifier la fonction `find`, non pas pour qu'elle rende un autre résultat, mais pour qu'en plus de calculer le représentant canonique recherché, elle aplatisse l'arbre "en passant".

Une première version très simple à implémenter consiste tout simplement à faire pointer chaque nœud visité vers son grand-père :

```
  static int find(int i) {
        while (pere[i] != i) {
            pere[i] = pere[pere[i]];
            i = pere[i];
        }
        return(i);
    }
```

Cette optimisation sera déjà très efficace.

Une autre version permet d'aplatir complètement le chemin, mais demande de parcourir ce dernier deux fois.

```
  static int find(int i) {
   int j = i;
   while (j != pere[j]) j = pere[j];
   int k;
   while (i != j) {
       k = pere[i];
       pere[i] = j;
       i = k;
   }
   return(j);
}
```

10.7 Exemples d'applications

Union-find est particulièrement pratique pour trouver des structures qu'on construit par transformations où ajouts successifs jusqu'à un certain seuil.

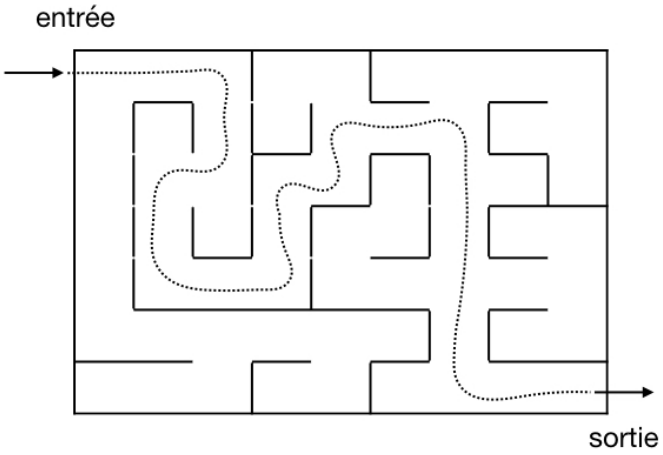
10.7.1 Percolation

Un joli exercice de programmation est l'utilisation de union-fond pour estimer le seuil de percolation dans un réseau carré. On décrit le problème en TD.

10.7.2 Labyrinthe

La construction d'un labyrinthe n'ayant qu'une solution est un problème proche de la percolation.

Solution page 192. **Exercice 10.7.1** On veut fabriquer de manière aléatoire des labyrinthes de la forme suivante, formé de n sur m cases carrées, séparées ou non par une cloison :



Il faut donc décider aléatoirement où placer des cloisons. On veut être sûr que :

- Il y ait un et un seul chemin de la case d'entrée vers la case de sortie (on s'interdit de faire demi-tour),
- il y a un minimum de cloisons, en particulier toutes les cases sont accessibles depuis l'entrée (et la sortie).

Expliquez comment on peut construire un tel labyrinthe, même très grand, avec union-find. ⋄

Chapitre 11

Les deux implémentations des piles

Ce petit chapitre présente deux manières d'implémenter les piles (*stacks*) qui sont une structure de données on ne peut plus classique et que nous avons indirectement déjà beaucoup utilisée dans ce livre, puisque la XVM repose justement sur une pile.

Mais les vrais buts de ce chapitre sont :

- De montrer, à travers un exemple simple, comment on peut utiliser le mécanisme de classes de Java pour construire deux implémentations différentes d'une même structure, de manière à passer facilement de l'une à l'autre. C'est une question de *modularité*.
- Sur un plan tout à fait différent, du point de vue algorithmique, on en profite pour présenter les techniques utilisant les tableaux redimensionnables, et plus généralement l'analyse de *complexité amortie*, ou complexité en moyenne.

11.1 Ce qui est attendu

Un pile est une structure de données qui, à tout moment, contient un nombre fini d'objets d'un même type ou d'une même classe. Dans ce qui suit, nous construirons des piles de chaînes de caractères, donc de classe `String`. On donnera à la fin du chapitre le code d'une version générique[1]. Les deux opérations principales sont celles permettant d'une part d'ajouter ou "pousser" (*push*), d'autre part d'enlever, un objet de la pile. Il faut également être capable de vérifier si une pile est vide.

Prenons une approche orientée-objet ; on veut donc une classe[2] `MyStack` munie d'un constructeur `MyStack()` qui crée une pile vide, et des méthodes suivantes :

- `boolean isEmpty()` qui indique si la pile est vide ;
- `void push(String x)` qui ajoute l'élément `x` sur la pile ;
- `String pop()` qui retire et renvoie l'élément en haut de la pile (et déclenche une erreur si la pile est vide).

Si on parle de pile, c'est qu'on est dans le cas d'une structure LIFO (*last in first out*), c'est-à-dire que l'élément retiré par `pop` est celui qui est *en haut* de la pile, donc le dernier à y avoir été poussé.

1. C'est-à-dire de piles dont on peut choisir la classe des éléments ; voir les sections 5.11 et 6.8.2.
2. On évite de l'appeler `Stack` car cet identificateur est déjà utilisé par la bibliothèque de Java.

11.2 Implémentation par liste chaînée

Conceptuellement, une pile est une liste. En particulier, le haut de la pile est le début de la liste, puisqu'il est facile d'ajouter ou de retirer un élément en tête de liste. Il est donc naturel de proposer une implémentation à base de listes chaînées. On propose l'implémentation habituelle des listes avec une classe récursive enveloppée (comme expliqué au chapitre 5).

```
class StackContent {
    String hd;
    StackContent tl;
    StackContent(String x, StackContent l) {
        hd = x;  tl = l;
  }
}

class MyStack {
    private StackContent content;

    MyStack() { content = null; }

    boolean isEmpty() { return(content == null); }

    void push(String x) {
        content = new StackContent(x, content);
    }

    String pop() {
    if (content == null) throw new Error("pop");
    String x = content.hd;
    content = content.tl;
    return(x);
    }
}
```

Cette implémentation fonctionne bien. On peut vérifier expérimentalement que les opérations push et pop sont en temps constant.

11.3 Implémentation par tableaux

On peut essayer de remplacer les listes chaînées par des tableaux pour gagner de l'espace mémoire ou gagner du temps. Dans les deux cas, on n'améliorera évidemment pas la complexité asymptotique, mais on peut espérer améliorer le facteur constant.

11.3.1 Tableau de taille fixe

Le plus simple est de choisir une taille fixe de tableau. Il faut alors déclencher une erreur si cette taille est dépassée. Remarquons que c'est la solution adoptée pour les piles des machines abstraites XVM et JVM.

```
class MyStack {
  private String[] content;
  private int index;
  MyStack(int n) {
    content = new String[n];
    index = 0;
```

```
   }
   public boolean isEmpty() {
     return(index == 0);
  }
  void push(String s) {
    if (index = content.length) throw new Error("push");
    content[index] = s;
    index++;
  }
  public String pop() {
   if (index == 0) throw new Error("pop");
   index--;
   return(content[index]);
  }
}
```

Si on essaye expérimentalement cette implémentation, on verra que les opérations `push` et pop sont toujours en temps constant, et probablement un peu plus rapides qu'avec des listes chaînées.

11.3.2 Tableau redimensionnable

Si un tableau de taille fixe peut être utilisé dans certains cas où l'on peut prévoir la taille maximale de la pile, ce n'est pas une solution au cas général. Pour le traiter, on choisit de *redimensionner* le tableau lorsqu'il est plein. C'est-à-dire qu'on crée un nouveau tableau plus grand dans lequel on recopie le contenu courant de la pile. La méthode qui redimensionne le tableau n'a pas à être visible de l'extérieur, et est donc privée. Voici une possibilité :

```
    private void reSize() {
        String[] nt = new String[content.length + 100];
        for (int i = 0; i < content.length; i++)
            nt[i] = content[i];
        content = nt;
    }
    void push(String s) {
        if (index == content.length) reSize();
        content[index] = s;
        index++;
    }
```

Attention à la complexité

Avec cette modification, on traite effectivement le problème général de la pile en utilisant des tableaux. En revanche, cette première version est trop naïve pour être efficace.

Analysons la complexité en temps d'une série de N opérations `push` successives.

- Chaque opération `reSize` prend un temps proportionnel à la taille courante de la pile. Disons que le temps est $a \cdot h$ où h est la hauteur de la pile.
- Si N est grand, on aura besoin de $N/100$ opérations `reSize`.
- La première opération `reSize` prendra un temps $a \cdot 100$, la seconde $a \cdot 200$, la troisième $a \cdot 300$ jusqu'à $a \cdot N/100$.

On voit que le temps total est $a \cdot N(N-1)/20000$ c'est-à-dire $O(N^2)$. Or on voudrait un temps constant pour chaque opération, c'est-à-dire un temps total en $O(N)$.

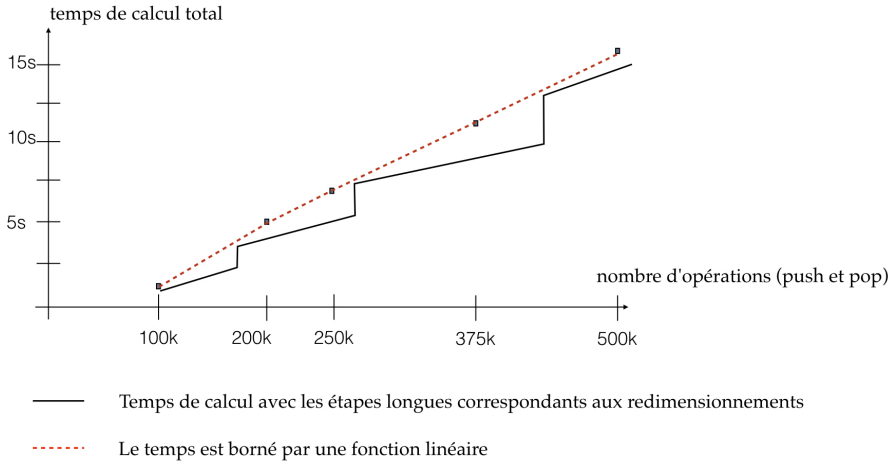


FIGURE 11.1 – Complexité amortie : temps nécessaires pour ajouter puis enlever n éléments dans une pile implémentée par tableaux redimensionnables.

Augmenter suffisamment la taille

Ce problème de complexité est du au fait que l'on passe trop de temps à recopier la pile parce que l'on effectue trop de redimensionnements. Pour remédier à cela, il faut plus augmenter la taille de la pile : au lieu de l'augmenter d'une taille fixe, on va la doubler à chaque redimensionnement. Il suffit pour cela de changer le code de la méthode correspondante :

```
private void reSize() {
    String[] nt = new String[content.length * 2];
    for (int i = 0; i < content.length; i++)
        nt[i] = content[i];
    content = nt;
}
```

11.3.3 Analyse de complexité en moyenne

On peut toujours considérer que le temps d'exécution de `reSize()` est $a \cdot h$

En revanche, en doublant la taille du tableau à chaque redimensionnement, on a un premier redimensionnement de taille 5, un second de taille 10, puis 20, 40 jusqu'à N, c'est-à-dire que l'on a au plus $\log_2(N)$ redimensionnements et un temps de calcul borné par :

$$a \cdot (N + \frac{N}{2} + \frac{N}{4} + \ldots + \frac{N}{2^i} + \ldots + 1)$$

Rappelons qu'il s'agit là du temps nécessaire à N opérations `push`. Le *temps moyen* d'une opération est donc borné par une constante.

En revanche, lorsqu'une opération `push` nécessite un redimensionnement, elle prendra un temps proportionnel à la taille courante de la pile.

La figure 11.1 présente la courbe de complexité de l'implémentation par tableaux redimensionnables en fonction du nombre d'opérations `push` successives.

À retenir

On a ici une situation typique de *complexité amortie* ou complexité en moyenne. La complexité en moyenne d'une opération est faible, mais dans certains cas, une opération peut coûter cher. Ces cas sont forcément rares, puisque la complexité en moyenne est bonne.

Remarquons bien que les techniques combinant tableaux redimensionnables et complexité en moyenne ne sont pas propres aux piles ; en revanche, on gardera le principe d'une progression géométrique de la taille pour avoir une bonne complexité amortie. Un exemple classique est celui des tables d'association par hachage décrites dans le chapitre 13.

11.3.4 Que choisir ?

On peut retrouver ce cas de figure dans d'autres cas, avec une alternative entre :

- soit une implémentation par tableaux redimensionnables, plus efficace au total mais avec certaines étapes qui peuvent prendre du temps,
- ou bien une implémentation par listes chaînées, moins efficace en moyenne mais avec une borne sur chaque étape élémentaire.

Le choix entre les deux dépendra de l'importance d'avoir ou non une borne sur le temps mis par chaque étape.

Exercice 11.3.1 Qu'en est-il de la comparaison entre ces deux implémentations pour ce qui est de la mémoire utilisée ? On se restreint pour l'instant à ce qui se passe lorsque l'on effectue N opérations `push`. ◇

Solution page 193.

11.3.5 Ne pas utiliser trop de mémoire

L'implémentation par tableaux redimensionnables telle que proposée ci-dessus n'est toujours pas tout-à-fait satisfaisante. En effet, supposons que l'on commence par ajouter beaucoup d'éléments dans la pile, puis qu'on en enlève (dépile) la plus grande partie. Dans ce cas, on commence par faire grossir le tableau mais on ne diminue jamais sa taille, même si on n'en utilise plus qu'une petite partie. Autrement dit, la quantité de mémoire utilisée ne diminue jamais (contrairement à ce qui se passe pour les listes chaînées).

On peut alors choisir de diminuer la taille du tableau lorsqu'il n'est que peu rempli. Dans un premier temps, on fait la même analyse que pour augmenter la taille du tableau :

- d'une part, comme pour tous les redimensionnements, il ne faut pas le faire trop souvent puisque diminuer la taille du tableau demande de recopier ses éléments,
- on va donc diviser par deux la taille du tableau lorsqu'on redimensionne.

Mais il y a une astuce, ou un piège, supplémentaire. En effet, si on divise par deux la taille du tableau dès qu'il n'est plus qu'à moitié rempli, on risque de faire énormément de redimensionnements si la taille de la pile oscille autour du seuil de redimensionnement. On ne va donc diminuer la taille du tableau que lorsqu'il ne sera plus rempli qu'au quart. On donne l'implémentation complète à la figure 11.2.

On peut montrer qu'avec une telle implémentation, pour toute séquence de N opérations `push` ou `pop` :

- Le temps de calcul est linéaire par rapport à N (en $O(N)$).
- La taille du tableau est au plus 4 fois le nombre d'éléments présents dans la pile.

```
class Redim {
    private String[] content;
    private int index;
    Redim() {
        content = new String[2];
        index = 0;
    }
    private void doubleSize() {
        String[] nt = new String[content.length * 2];
        for (int i = 0; i < content.length; i++)
            nt[i] = content[i];
        content = nt;
    }
    private void halfSize() {
        String[] nt = new String[content.length / 2];
        for (int i = 0; i < index; i++)
            nt[i] = content[i];
        content = nt;
    }
    public boolean isEmpty() {
     return(index == 0);
    }
    void push(String s) {
        if (index == content.length) doubleSize();
        content[index] = s;
        index++;
    }
    public String pop() {
        if (index < content.length / 4 && content.length > 10) halfSize();
        if (index == 0) throw new Error("pop");
        index--;
        return(content[index]);
    }
}
```

FIGURE 11.2 – L'implémentation complète à base de tableaux redimensionnables

11.4 Versions génériques

On s'est restreint, dans ce qui précède, à des piles de chaînes de caractères, pour pouvoir mettre l'accent sur les aspects algorithmiques. Mais le cas général est évidemment la construction de piles d'éléments de classes arbitraires, c'est-à-dire la construction de classes génériques.

On donne donc les versions génériques des deux implémentations, par listes chaînées et par tableaux redimensionnables.

Pour les listes chaînées, il suffit d'ajouter les annotations de classe au bons endroits.

```
class StackContent<E> {
    E hd;
    StackContent<E> tl;
    StackContent(E x, StackContent<E> l) {
        hd = x;  tl = l;
   }
}

class MyStack<E> {
    private StackContent<E> content;

    MyStack() { content = null; }

    boolean isEmpty() { return(content == null); }

    void push(E x) {
        content = new StackContent<E>(x, content);
    }

    E pop() {
      if (content == null) throw new Error("pop");
      E x = content.hd;
      content = content.tl;
      return(x);
    }
}
```

Pour les tableaux redimensionnables, on doit utiliser un tableau de `Object` à cause de la restriction de Java sur les tableaux génériques, comme décrit en 6.8.2. Comme ce tableau est privé, on sait qu'il n'est modifié que par les méthodes de la classe, donc qu'il ne contient que des éléments de la classe `E`; cela permet de ne pas vérifier la classe de l'objet dans la méthode `pop()`.

```
class Redim<E> {
   private Object[] content;
   private int index;
   Redim() {
     content = new Object[2];
     index = 0;
  }
  private void doubleSize() {
      Object[] nt = new Object[content.length * 2];
      for (int i = 0; i < content.length; i++)
      nt[i] = content[i];
      content = nt;
  }
  private void halfSize() {
```

```
      Object[] nt = new Object[content.length / 2];
      for (int i = 0; i < index; i++)
            nt[i] = content[i];
            content = nt;
  }
  public boolean isEmpty() {
     return(index == 0);
  }
  void push(E x) {
     if (index == content.length) doubleSize();
     content[index] = x;
     index++;
  }
  public E pop() {
     if (index < content.length / 4 && content.length > 10) halfSize();
     if (index == 0) throw new Error("pop");
     index--;
     return((E)content[index]);
  }
}
```

Chapitre 12

Files de priorité en Java

12.1 Files d'attentes simple

La file d'attente est souvent vue comme la structure duale de la pile : si cette dernière est *last in, first out* ou LIFO, la file est *first in, first out* ou FIFO ou encore "premier arrivé premier servi".

L'interface de la file d'attente est essentiellement identique à celle des piles avec trois méthodes pour respectivement :

- ajouter un nouvel élément dans la file,
- faire sortir le premier élément de la file,
- vérifier si la file est vide.

On peut regrouper cela dans l'interface générique suivante :

```
interface MyQueue<E> {
  boolean isEmpty();
  E pop();
  void push(E e);
}
```

On appelle ici cette interface `MyQueue` pour la distinguer de l'interface `Queue<E>` présente dans la bibliothèque Java ; implémenter cette dernière interface est possible mais serait trop long pour être pédagogique.

On peut implémenter les files en utilisant les mêmes chaînes que les listes ou les piles ; prenons :

```
class Node<E> {
  E content;
  Node<E> next;
  Node(E e, Node<E> l) {
     content = e; next = l;
  }
}
```

La différence avec les piles est que les opérations `pop` et `push` n'agissent pas au même endroit de la file. La première agit sur la tête de la file, la seconde à l'arrière de la file (alors que pour les piles, les deux opérations agissent sur le "haut" de la pile). Il faut donc garder en

mémoire *deux* pointeurs, correspondant respectivement à l'avant et l'arrière de la file. Voici une implémentation possible que nous appellons `PQueue` pour *Pointer Queue* :

```
class PQueue<E> implements MyQueue<E> {
    private Node<E> front;
    private Node<E> back;
    PQueue() {
        front = null; back = null;
    }
    public boolean isEmpty() {
        return(front == null) ;
    }
    public E pop() {
        if (front == null) throw new Error("pop");
        E r = front.content;
        front = front.next;
        if (front == null) back = null;
        return(r);
    }
    public void push(E e) {
        if (front == null) {
            Node<E> n = new Node<E>(e, null);
            front = n;
            back = n;
            return;
        }
        back.next = new Node<E>(e, null);
        back = back.next;
    }
}
```

On peut vérifier que chacune des trois méthodes s'effectue en temps constant.

Encapsulation et invariant

Ce qui est important c'est que le fait que les champs soient privés et ne peuvent être modifiés que par les deux méthodes `push` et `pop` suffit à garantir la cohérence de la représentation interne. Plus précisément, il y a un *invariant* qui est préservé par l'ensemble des méthodes ; à tout moment :

- soit les deux champs `front` et `back` sont égaux à **`null`**,
- soit `front` point vers le premier `Node` d'une liste et `back` pointe vers le dernier `Node` de cette même liste ; en particulier `back.next` vaut **`null`**.

Le fait que l'encapsulation permet de garantir la préservation d'invariants sur les structures de données est un point essentiel de la programmation orienté-objet.

12.2 La spécification des files de priorité

Une file de priorité correspond à encore une autre compréhension de l'interface `Queue`. Cette fois on dispose d'un ordre total sur les éléments de la classe `E`. Plus précisément on dispose d'une fonction de comparaison sur les éléments de `E` ; pour l'instant on supposera simplement que cette fonction est une fonction statique :

```
static boolean compare(E e1, E e2)
```

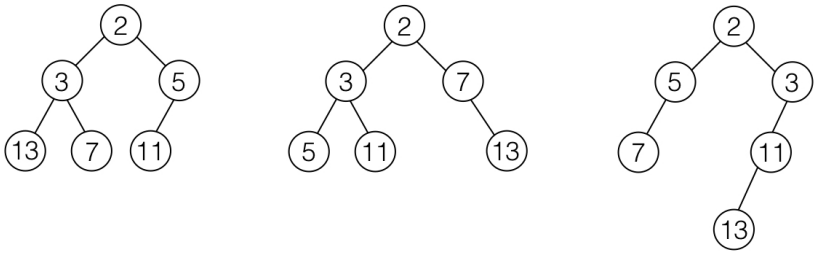


FIGURE 12.1 – Trois exemples de tas contenant les 6 premiers nombres premiers. Le premier est équilibré à gauche, le second seulement équilibré, le troisième n'est pas équilibré.

La spécification d'une file de priorité est que l'opération `pop` va cette fois renvoyer, et sortir de la file, l'élément *le plus petit* vis à vis de l'ordre défini par `compare`.

Par exemple, dans le cas d'une file de priorité de nombres entiers, si `compare` correspond à l'ordre habituel, si l'on fait les operations suivantes sur une file `f` initialement vide :

```
f.push(4);
f.push(1);
f.push(6);
```

des opérations `f.pop()` successives rendront 1, puis 4, puis 6.

Une illustration courante est de dire que si une file simple correspond à une file d'attente dans un magasin, la file de priorité correspond à la manière dont sont gérés les patients dans un service d'urgence dans un hôpital : le premier patient traité n'est pas forcément celui qui est arrivé en premier, mais celui dont l'état est jugé le plus urgent.

12.3 Principe algorithmique

Un peu comme dans le cas de union-find, l'idée est de ranger les éléments de la file de manière suffisamment ordonnée mais pas trop contrainte :

- Si on garde les éléments dans une liste non-triée, alors il faut in temps $O(n)$ pour trouver le plus petit élément (opération `pop()` inefficace).
- Si on garde les éléments dans une liste triée, alors il faut un temps $O(n)$ pour insérer un nouvel élément (opération `push(n)` inefficace).

La première solution n'est pas assez ordonnée, la seconde l'est trop. La structure qui offre le bon compromis entre ces deux solutions est d'utiliser un arbre binaire, donc chaque nœud porte un élément de la file.

Cet arbre est ordonné suivant le principe du *tas* : le père est toujours plus petit que ses fils. On donne trois exemples de tas en figure 12.1. Remarquez que cette notion de tas n'a rien à voir avec l'espace mémoire également appelé tas.

12.4 Représentation de l'arbre dans un tableau

On devine que pour avoir de bonne propriétés algorithmiques, on voudra garder l'arbre aussi équilibré que possible. Dans l'implémentation la plus utilisée des files de priorité, on va toujours avoir un arbre de la forme suivante :

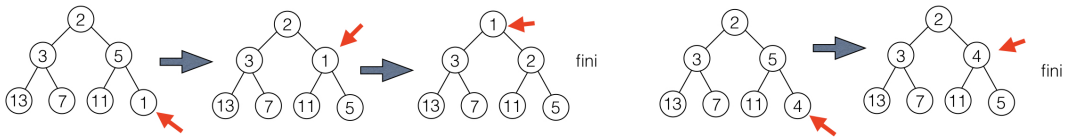
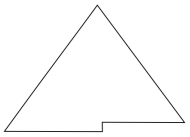
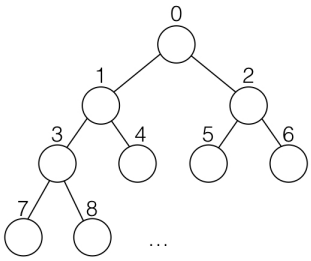


FIGURE 12.2 – Deux séquences d'exécution correspondant à la méthode up



C'est-à-dire que la différence de longueur entre les branches de l'arbre est au plus 1, et que les branches les plus longues sont à gauche.

De plus, plutôt que d'utiliser une structure d'arbre à base de pointeurs, on peut ranger un tel arbre dans un tableau, en utilisant la numérotation suivante :



On voit que le père d'un sommet d'indice i se trouve à la position $(i-1)/2$. Inversement, le fils gauche de i est en $2i+1$ et le fils droit en $2i+2$.

L'avantage de cette représentation est qu'un arbre équilibré à gauche de k nœuds utilise exactement les k premières cases du tableau. Aussi, si on ajoute un élément, il faudra utiliser la $k+1$-ième case et inversement si on ôte un élément, on devra libérer la k-ième case.

12.5 Ajout d'un élément

La méthode `void push(E e)` va donc ajouter un élément dans la file d'attente. On procède en deux étapes :

1. On commence par placer l'élément `e` à la $k+1$-ième case.

2. Ensuite on rétablit l'ordre (donc la propriété d'être un tas) en permutant cet objet avec son père, jusqu'à ce que soit on atteigne la racine de l'arbre, soit on atteint une position où il est moins prioritaire que son père.

La deuxième étape sera effectuée par une méthode privée qu'on peut appeler up. La figure 12.2 montre des étapes successives de cette opération.

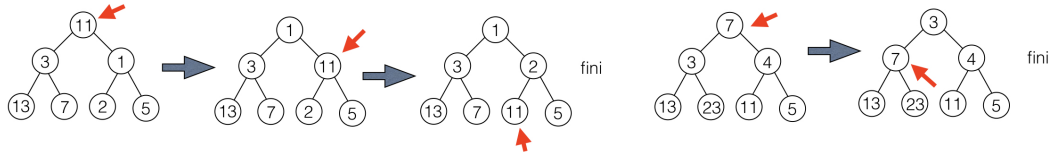


FIGURE 12.3 – Deux séquences d'exécution correspondant à la méthode `down`

12.6 Sortie d'un élément

La méthode `E pop()` va ôter l'élément le plus prioritaire de la file. Elle procède de manière assez similaire :

1. L'élément le plus prioritaire est celui en position 0. Il est ôté de l'arbre pour pouvoir être rendu comme résultat. À sa place on commence par recopier le "dernier" élément du tas, c'est-à-dire celui en position k.
2. Il faut ensuite, ici aussi réordonner le tas. Cette fois on permute l'élément en position 0 avec le plus prioritaire de ses fils, jusqu'à ce qu'il n'y ait plus de fils plus prioritaire que lui ; c'est illustré en figure 12.3.

12.7 Vers l'implémentation

Une file de priorité sera donc en particulier composée de :

– Un champ tableau `content` pour représenter le tas des éléments,

– un champ entier `index` contenant le nombre d'éléments de la file ; ce nombre correspond donc aussi à la première case "libre" du tableau `content`.

A cause du problème des *tableaux génériques* (voir 6.8.2) le tableau `content` doit être de type `Object[]` et pas `E[]`. On pourrait donc débuter la définition de la classe des files de priorité par :

```
class MyPriorityQueues<E> implements MyQueues<E> {
   private int index;
   private Object[] content;
   MyPriorityQueue() {
      index = 0;
      content = new Object[7]; // on choisit une taille initiale
                                          // ici une puissance de 2
                                          // moins 1
  }
  public void push(E e) {
     if (index == content.length) enlarge(); // redimensionnement
     content[index] = e;
     up(index);                  // appel de la fonction qui réordonne
     index++;
  }
  private swap(int i, int j) {
      Object ci = content[i];
      content[i] = content[j];
      content[j] = ci;
  }
```

```
...
}
```

On ne détaille pas, pour l'instant la méthode de redimensionnement `enlarge()` qui ressemble à ce que nous avons vu pour les piles ou les tables de hachage.

En revanche, la méthode de ré-ordonnancement `up(int i)` doit tenir compte de la relation de priorité. Si l'on travaillait sur des entiers avec l'ordre habituel on aurait le code suivant :

```
private void up(int i) {
     while (i != 0) {
         int pere = (i - 1) / 2;
         // si le père est prioritaire, c'est fini :
         if (content[pere] <= content[i]) return;
         swap(i, pere);          // sinon on échange
         i = pere;                 // et on itère
    }
}
```

Les méthodes `pop()` et `down(int i)` sont similaires. On voit que ce qu'il nous reste à faire c'est de comprendre comment passer à la file la fonction de comparaison.

12.8 Une interface pour définir l'ordre

Essentiellement, il faut paramétrer le code par une fonction. Java permet cela procédant en deux étapes :

- Encapsuler le code de la fonction dans un objet,
- puis passer cet objet en argument au code.

On commence donc par donner une interface [1] dont les instances contiendront exactement une fonction de comparaison :

```
interface Comp<E> {
    abstract boolean c(E e1, E e2);
}
```

Cela suffit pour terminer l'implémentation des files de priorités. Il suffit d'avoir un champ contenant la fonction de comparaison et de la passer en argument au constructeur :

```
class PriorityQueues<E> implements MyQueue<E> {
    private int index;
    private Object[] content;
    private Comp<E> comp;
    PriorityQueues(Comp<E> c) {
        index = 0;
        content = new Object[7];
        comp = c;
    }
...
```

On peut ensuite définir les méthodes `up` et `down` qui font appel au champ `comp`. Par exemple :

```
    private void up(int i) {
        while (i != 0) {
                int pere = (i - 1) / 2;
```

1. On peut tout à fait utiliser une classe abstraite au lieu d'une interface.

```
                if (comp.c((E)content[pere], (E)content[i])) return;
                swap(i, pere);
                i = pere;
        }
    }
```

On donne le code complet de la classe à la fin du chapitre (12.13).

12.9 Utilisation

Pour utiliser l'implémentation des files de priorités, il faut donc construire une instance de l'interface `Comp`. Faisons le pour l'ordre croissant sur les nombres entiers :

```
class CompP implements Comp<Integer> {
    public boolean c (Integer i, Integer j) {
        return(i <= j);
    }
    CompP() {}
}
```

On peut alors créer une file de priorité en construisant une instance de `CompP` puis en la passant au constructeur :

```
CompP comp = new CompP();
MyPriorityQueue<Integer> t = new MyPriorityQueue<Integer>(comp);
```

12.10 Classes anonymes

Une fois compris le mécanisme permettant de passer une fonction (ici la fonction de comparaison) en argument, on peut remarquer une certaine lourdeur à ce mécanisme. En particulier, chaque fois qu'on veut passer une nouvelle fonction en argument, il faut définir une nouvelle classe. Par exemple si on veut, toujours pour les nombres entiers, considérer que c'est le plus grand le plus prioritaire, il faut définir une nouvelle implémentation de l'interface `Comp` comme :

```
class CompN implements Comp<Integer> {
    public boolean c (Integer i, Integer j) {
        return(j <= i);
    }
    CompN() {}
}
```

Puis il faut créer une instance de cette classe.

On voit en particulier que les classes `CompN` ou `CompP` sont construites pour n'avoir chacune qu'une seul instance.

Cela sera en fait systématiquement pour des interfaces comme `Comp` et ses implémentations ; il s'agit en effet d'une interface composée d'une unique méthode. On parle de *single abstract method* (SAM). Lorsqu'on veut créer un objet dont la classe est une (implémentation d')une telle SAM, on peut, depuis la version 7 de Java, recourir au mécanisme de *classe anonyme*. Le résultat est le même mais le code est plus concis. Dans le cas présent cela peut donner :

```
Comp<Integer> comp = new Comp<Integer>() {
     public boolean c(Integer a, Integer b) {
```

```
        return(a <= b);
    }
};

MyPriorityQueue<Integer> t = new MyPriorityQueue<Integer>(comp);
```

On voit qu'on n'a plus eu besoin de nommer la classe de `comp` (ici de définir `CompP`. La syntaxe permet de définir l'objet et sa classe en donnant le code de son unique méthode.

12.11 Notation lambda

Le terme de notation *lambda* vient des langages fonctionnels comme Caml (Leroy et al., 2017) qui l'ont eux-mêmes repris du formalisme appelé λ-calcul (Barendregt, 1984). Dans le cas de Java, cette notation a été introduite avec la version 8 et permet de raccourcir encore un peu la notation des classes anonymes. La notation est celle qu'on utilise en mathématiques quand on définit une fonction, par exemple, par $x \mapsto 3x^2 - 7$.

Pour rester dans le cas de notre interface `Comp`, on peut remplacer la définition de l'objet `comp`, au choix, par :

```
Comp<Integer> comp = (Integer a, Integer b) -> { return(a <= b); };
Comp<Integer> comp = (a, b) -> { return(a <= b); };
Comp<Integer> comp = (a, b) -> (a <= b);
Comp<Integer> comp = (Integer a, Integer b) -> (a <= b);
```

Notons bien que ces notations portent toutes sur des interfaces ou classes abstraites avec *single abstract method*.

12.12 Files de priorité dans la bibliothèque Java

La bibliothèque Java propose, bien sûr, une implémentation de files de priorité nommée `PriorityQueue<E>` . Pour l'essentiel, cette implémentation suit les principes et l'architecture de ce que nous présentons dans ce chapitre. Les différences portent essentiellement sur des conventions avec des interfaces plus complexes que celles que nous avons utilisées :

- La classe `PriorityQueue` est une implémentation de l'interface `Queue`.
- Du coup les méthodes principales pour ajouter et retirer des éléments sont `add(E e)` et `poll()` (au lieu de `push` et `pop`).
- La classe des fonctions de comparaison s'appelle `Comparator<E>`; surtout la méthode unique de celle-ci ne rend pas un booléen mais un entier :

  ```
  int   compare(E e1, E e2)
  ```

  ce qui permet de distinguer le cas de deux éléments égaux.
- Il existe également un constructeur de `PriorityQueue<E>` qui ne prend pas de `Comparator` en argument. Dans ce cas, l'implémentation suppose que la classe `E` est munie d'une méthode `compareTo` qui définit un *ordre naturel* sur `E`.

12.13 Implémentation complète

On donne ci-dessous le code complet correspondant à ce qui est décrit dans ce chapitre.

```
class MyPriorityQueue<E> implements MyQueue<E> {
    private int index;
    private Object[] content;
    private Comp<E> comp;
    MyPriorityQueue(Comp<E> c) {
        index = 0;
        content = new Object[7];
        comp = c;
    }
    private void swap(int i, int j) {
        Object a = content[i];
        content[i] = content[j];
        content[j] = a;
    }
    private int gauche(int i) {
        return(2 * i + 1);
    }
    private int droite(int i) {
        return(2 * i + 2);
    }

    private void down() {
        int i = 0;
        while (gauche(i) < index) {
            if (droite(i) >= index) {
                if (comp.c((E)content[gauche(i)], (E)content[i]))
                    swap(i, gauche(i));
                return;
            }
            if (comp.c((E)content[i], (E)content[gauche(i)]) &&
                comp.c((E)content[i], (E)content[droite(i)]))
                return;
            int m = gauche(i);
            if (comp.c((E)content[droite(i)], (E)content[m]))
                m = droite(i);
            swap(i,m);
            i = m;
        }
    }
    private void up(int i) {
        while (i != 0) {
                int pere = (i - 1) / 2;
                if (comp.c((E)content[pere], (E)content[i])) return;
                swap(i, pere);
                i = pere;
        }
    }
    public E pop() {
        if (index == 0) throw new Error("pop");
        E r = (E)content[0];
        index --;
        content[0] = content[index];
        down();
        return(r);
    }
    public void push(E e) {
        if (index == content.length) enlarge();
        content[index] = e;
        up(index);
```

```
        index++;
    }
    public boolean isEmpty() {
        return(index == 0);
    }
    private void enlarge() {
        Object[] nt = new Object[content.length * 2];
        for (int i = 0; i < index; i++)
            nt[i] = content[i];
        content = nt;
    }

    public static void main(String[] args) {
        // exemple d'utilisation

        // syntaxe lambda
        Comp<Integer> comp = (a, b) -> { return(a > b); };

        MyPriorityQueue<Integer> t = new MyPriorityQueue<Integer>(comp);
        for (String i : args)
            t.push(Integer.parseInt(i));
        while (!t.isEmpty())
            System.out.println(t.pop());
      }
}
```

Chapitre 13

Tables de hachage

13.1 Objectif

Ce chapitre aborde un problème très fréquemment rencontré en informatique : la recherche d'informations dans un ensemble géré de façon dynamique. Les structures qui permettent cela et que nous allons décrire ici sont appelées des *tables d'association*.

Nous nous plaçons dans le cadre où une information complète se retrouve normalement à l'aide d'une clé qui l'identifie. Une illustration typique est l'annuaire téléphonique : connaître le nom et les prénoms d'un individu suffit normalement pour retrouver son numéro de téléphone. En cas d'homonymie absolue (tout de même rare), on arrive toujours à se débrouiller. Dans les bureaucraties modernes qui refusent le flou (et dans les ordinateurs) la clé doit identifier un individu unique, d'où, par exemple, l'idée du numéro de sécurité sociale. Nous voulons un ensemble dynamique d'informations, c'est-à-dire aussi pouvoir ajouter ou supprimer un élément d'information. On en vient naturellement à une interface caractérisant les opérations possibles sur ces données :

1. Trouver l'information associée à une clé donnée (par exemple quelle est l'adresse de "Jean Dupont").
2. Ajouter une nouvelle association entre une clé et une information (par exemple "Jean Dupont" habite au "57 rue des Cerises").

La seconde opération mérite d'être détaillée. Lors de l'ajout d'une paire clé-information, nous précisons :

- S'il existe déjà une information associée à la clé dans la table, alors la nouvelle information remplace l'ancienne.
- Sinon, une nouvelle association est ajoutée à la table.

On voit qu'il n'y a jamais dans la table deux informations distinctes associées à la même clé.

Notons qu'un cas particulier : lorsque l'information associée à la clé est triviale, on gère juste un ensemble de clé (la clé appartient à l'ensemble ou pas).

Exemple

On peut remarquer qu'un compilateur va souvent utiliser un tel mécanisme. Par exemple, lorsqu'il analyse du code, le compilateur va détecter les variables statiques et leur associer une

adresse, comme on l'a informellement décrit dans le chapitre 2. De même, lors de la compilation d'une classe, le compilateur associe un numéro à chaque champ d'une part, à chaque méthode d'autre part, comme décrit en 6.9. A chaque fois, ces associations pourront être mémorisées dans une table de hachage.

Dans ce qui suit, on prend un exemple "jouet" avec un carnet d'adresses, les clés seront des personnes décrites par leur nom et leur prénom :

```
class Personne {
  String nom;
  String prenom;
  Personne(String n, String p) {
      nom = n; prenom = p;
  }
}
```

Les adresses seront composées d'un numéro et d'un nom de rue :

```
class Adresse {
   int numero;
   String rue;
   Adresse(int n, String r) {
      numero = n;  rue = r;
  }
}
```

L'interface pour le carnet sera alors :

```
interface Carnet {
  Adresse get(Personne p);
  void add(Personne p, Adresse a);
}
```

Dans ce cas, on prend la convention que `get(p)` rend `null` si et seulement si il n'y a pas d'adresse associée à `p`.

Remarquons qu'on peut vouloir ajouter d'autres méthodes, par exemple une méthode `void remove(Personne p)` qui effacerait l'entrée `p` dans la table. Pour simplifier, on ne le fera pas dans l'exemple détaillé ici.

Notons aussi qu'il est clair que pour programmer une table d'associations, il est nécessaire de savoir vérifier l'égalité entre deux entrées. On peut choisir, pour cela, d'utiliser la méthode `equals` (voir 6.8.1). Cela veut dire que nous considérerons la définition de la classe `Personne` ci-dessus enrichie par :

```
@Override
boolean equals(Object o) {
   if (!(o instanceof Personne)) return(false);
   return(prenom.equals((Personne)o.prenom)
       && nom.equals((Personne)O.nom));
}
```

On remarque que pour comparer des `String` il faut utiliser `equals` et non pas `==`. La classe `String` est équipée d'origine d'une méthode `equals` vérifiant l'égalité structurelle.

13.2 Implémentation naïve

Pour commencer, on va définir une classe représentant les listes de paires formées par une personne et une adresse. Cela sera pratique dans une première implémentation naïve, mais

également pour les suivantes. L'idée est que une personne p est associée à l'adresse a si la paire (p, a) est présente dans la structure de donnée.

```
class Assoc {
    Personne personne;
    Adresse adresse;
    Assoc suite;

    Assoc(Personne p, Adresse a, Assoc s) {
     this.personne = p;
     this.adresse = a;
     this.suite = s;
  }
    Assoc findCell(Personne p) {
        if (p.equals(personne)) return(this);
        if (suite == null) return(null);
        return(suite.findCell(p));
    }
}
```

Une table d'association est donc fondamentalement un ensemble d'objets `Assoc`. Une manière naïve de procéder est d'utiliser une liste.

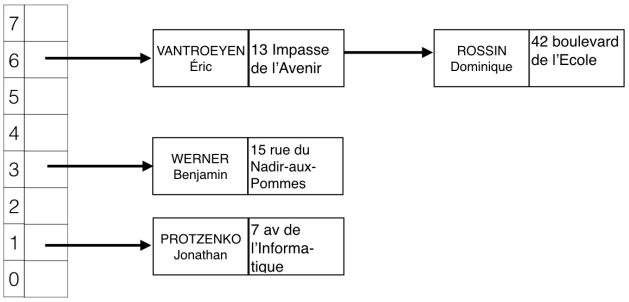
```
class L implements Carnet {
  Assoc contenu;
  L() { contenu = null; }

  public Adresse get(Personne p) {
      if (contenu == null) return(null);
      Assoc x = contenu.findCell(p);
      if (x == null) return(null);
      return(x.adresse);
  }

  public void add(Personne p, Adresse a) {
      if (contenu == null) {
          contenu = new Assoc(p, a, null);
          return;
      }
      Assoc x = contenu.findCell(p);
    if (x!=null) x.adresse = a;
    else contenu = new Assoc(p, a, contenu);
  }
}
```

Cette implémentation fonctionne, et pourrait facilement être généralisée à d'autres classes que `Personne` et `Adresse`. En revanche, elle n'est pas très efficace : on voit facilement que le temps d'exécution de `findCell` est proportionel à la longueur de la liste, c'est-à-dire au nombres d'entrée de la table. Cela veut dire que toute opération (recherche, modification) sera ralentie lorsque la taille de la table augmente, ce qui n'est pas satisfaisant.

13.3 Table de hachage

13.3.1 Principe

Si les clés sont des entiers compris entre 0 et $n - 1$, c'est facile : il suffit de stocker les valeurs dans un tableau de taille n. On appelle cela l'adressage direct ; ce serait le cas dans



Situation possible en mémoire lorsque la fonction de hachage vaut 1 pour la clé `"Protzenko" "Jonathan"`, 3 pour `"WERNER" "Benjamin"` et 6 pour `"VANTROEYEN" "Éric"` ainsi que pour `"ROSSIN" "Dominique"`.

FIGURE 13.1 – Résolution par chaînage

notre exemple si les personnes étaient identifiées par un numéro unique (comme le numéro de sécurité sociale). Une première difficulté apparaît lorsque ces numéros sont trop grands ; on peut alors se ramener à l'intervalle $0 \ldots n-1$ en prenant le reste de la division modulo n. Mais il y a alors la possibilité que deux personnes différentes se voient attribué le même indice dans le tableau ; c'est le phénomène de *collision* qui est crucial dans les tables de hachage. On voit un peu plus bas comment le résoudre.

L'idée de la table de hachage est de se ramener à ce cas. On se donne pour cela une fonction h qui à chaque clé va associer un entier compris entre 0 et n.

La méthode `hashCode()` fournie d'origine par Java est de bonne qualité sur les chaines de caractères `String`. On explique plus loin en 13.5 ce qu'on entend par la qualité d'une fonction de hachage.

On va donc stocker à l'emplacement $h(k)$ du tableau l'information associée à une clé k. Le problème est que la fonction h peut ne pas être injective, et qu'il est toujours possible que deux clés différentes se voient associées la même valeur de hachage, c'est-à-dire :

$$k \neq k' \ \wedge \ h(k) = h(k').$$

On appelle ce cas de figure une *collision*.

13.3.2 Résolution des collisions par chaînage

Il faut donc *résoudre les collisions*, c'est-à-dire séparer les informations associées à deux clés y compris lorsqu'elles ont la même valeur de hachage. Une solution simple est de mettre tous les éléments d'information dont les clés ont la même valeur de hachage dans une liste, comme celles utilisées en 13.2. Plus exactement, si la valeur de hachage est i, on va mettre cette liste en position i du tableau.

On donne dans la figure 13.1 le schéma d'une telle implémentation.

Voici une implémentation possible, qui réutilise la classe `L` définie en 13.2. On remarque de manière générale qu'on a également besoin du test d'égalité sur les clés, en particulier pour traiter les collisions lors de l'ajout d'une association à la table

```
class H implements Carnet {
    private L[] contenu;
    H() { contenu = new L[20]; }
```

```
    public void add(Personne p, Adresse a) {
        int i = Math.abs(p.hashCode() % contenu.length);
        if (contenu[i] == null) contenu[i] = new L();
        contenu[i].add(p, a));
    }

    public Adresse get(Personne p) {
        int i = Math.abs(p.hashCode() % contenu.length);
        if (contenu[i] == null) contenu[i] = new L();
        return(contenu[i].get(p));
    }
}
```

13.3.3 Complexité

Notons N le nombre d'éléments dans la table et supposons que le temps de calcul de la fonction de hachage est constant en $O(1)$. Supposons également que le hachage est uniforme, c'est-à-dire que la valeur de hachage d'une clé vaut $v \in [0 \dots n-1[$ avec une probabilité $1/n$.

Dans ce cas, la recherche d'un élément prend, en moyenne, un temps proportionnel à $1+\alpha$ où α est le *facteur de charge*, c'est-à-dire N/n. Le calcul précis n'est pas très difficile, peut être retrouvé dans la littérature (par exemple (Cormen et al., 2009)) et ne relève pas vraiment de ce cours. Mais on peut remarquer que si N devient nettement plus grand que n, on se retrouve dans une situation similaire à l'implémentation naïve, en divisant juste par n la taille de chaque liste.

13.3.4 Redimensionnement

Il faut donc borner le facteur de charge en augmentant la taille du tableau lorsque c'est nécessaire. On va typiquement se fixer une borne supérieure pour le facteur de charge et, comme c'est courant avec des tableaux redimensionnables (voir chapitre 11), utiliser une progression géométrique, en doublant la taille du tableau lorsque cette borne est atteinte. Dans une version avec redimensionnement on va donc ajouter une méthode pour agrandir le tableau :

```
class HR implements Carnet {
    private Assoc[] contenu;
     int count = 0;
    static double alpha = 4.0;              // le facteur de charge limite
    HR() { contenu = new Assoc[3]; }

    private void enlarge() {
        Assoc[] old = contenu;
        contenu = new Assoc[contenu.length * 2];
        for (int i = 0; i < old.length; i++)
            while (old[i] != null) {
                add(old[i].personne, old[i].adresse);
                old[i] = old[i].suite;
            }
    }

    public void add(Personne p, Adresse a) {
        if (count >= contenu.length * alpha) enlarge();
        int i = Math.abs(p.hashCode() % contenu.length);
        Assoc c = Assoc.findCell(p, contenu[i]);
```

```
        if (c != null) {
            c.adresse = a;
            return; }
        contenu[i] = new Assoc(p, a, contenu[i]);
        count++;
    }

    public Adresse get(Personne p) {
        int i = Math.abs(p.hashCode() % contenu.length);
        Assoc c = Assoc.findCell(p, contenu[i]);
        if (c == null) return(null);
        return(c.adresse);
    }
}
```

Avec une telle implémentation, le temps de chaque opération qui ne donne pas lieu à redimensionnement peut être considérée en temps constant. Le temps nécessaire à chaque redimensionnement est proportionel aux nombres d'associations dans le tableau. Le temps moyen des opérations est donc constant.

Remarquons qu'il serait possible d'ajouter une méthode permettant d'effacer une association de la table, même si nous ne la donnons pas ici. On peut aussi ajouter une fonctionnalité pour diminuer la taille du tableau lorsque la table est suffisamment peu remplie, sur le modèle de ce qui est décrit pour les piles dans le chapitre 11.

13.3.5 Adressage ouvert

Nous ne la détaillons pas ici, mais il faut mentionner une autre technique de résolution des collisions, à savoir le hachage ouvert.

Dans le hachage à adressage ouvert, les éléments d'informations sont stockés directement dans le tableau. Plus précisément, la table de hachage est un tableau de paires clé-information. Dans le cas de l'adressage ouvert, on a donc un facteur de charge α nécessairement inférieur à un. Étant donnée une clé k, on recherche l'information d'abord à la case d'indice $h(k)$, puis, si cette case est occupée par une information de clé différente de k, on continue la recherche en suivant une séquence d'indices prédéfinie, jusqu'à trouver soit la clé k, soit une case libre. Dans ce dernier cas, on sait qu'il n'y a pas d'information associée à k. La séquence la plus simple consiste à examiner successivement les cases $h(k), h(k) + 1, h(k) + 2$, etc (en procédant modulo la taille du tableau).

Pour ajouter une information (k, v) on procède de même, en regardant d'abord à l'indice $h(k)$ jusqu'à trouver soit une case vide, soit une case contenant la précédente association avec k.

On peut remarquer que l'adressage ouvert rend beaucoup plus compliqué la suppression d'une association dans la table.

13.4 Implémentation fournie par la bibliothèque Java

On trouve dans la package `java.util` une classe `HashMap<K,V>` paramétrée par la classe des clés `K` et la classe des informations `V`.

Voici le carnet d'adresses implémenté en utilisant `HashMap`. C'est essentiellement une paraphrase :

```
import java.util.*;
```

```
class HM implements Carnet {
    private HashMap<Personne, Adresse> contenu;
    HM() { contenu = new HashMap<Personne, Adresse> (); }

    public Adresse get(Personne p) { return(contenu.get(p)); }

    public void add(Personne p, Adresse a) { contenu.put(p, a); }
}
```

Remarquez qu'on peut préciser le facteur de charge de la table créée en utilisant des constructeurs particuliers fournis par cette classe.

> **Attention**
>
> Lorsqu'on utilise une bibliothèque comme `HashMap<K,V>` il est essentiel d'avoir bien (re-)défini les méthodes `hashCode` et `equals` de la classe `K`. Sinon, non seulement le programme n'aura pas le comportement attendu, mais en plus cela ne sera pas immédiatement visible, car il n'y aura pas forcément d'erreur à la compilation ou l'exécution.

13.5 Propriétés de la fonction de hachage

Décrire ce qui caractérise une bonne fonction de hachage est un problème algorithmique intéressant et suffisamment vaste pour déborder du sujet de ce cours. On en dit quelques mots néanmoins. Rappelons qu'une fonction de hachage est une fonction $h : U \to \{0; \ldots; m-1\}$ où U est l'ensemble des clés.

D'abord une fonction de hachage doit de rapprocher le plus possible d'une répartition uniforme.; ou, plus informellement, la fonction de hachage doit être "aussi injective que possible". En gros, cela veut dire que le nombre de clés k telles que $h(k) = j$ pour $0 \leq j < m$ doit être aussi proche que possible de $|U|/m$. Plus précisément, pour une probabilité P sur U on veut avoir pour tout j avec $0 \leq j < m$:

$$\sum_{\{k|h(k)=j\}} P(k) = \frac{1}{m}$$

En pratique on ne connait que rarement la probabilité P et on se limite à des heuristiques. En particulier, on veut que des clés voisines (ce qui arrive souvent en pratique comme Dupont et Dupond) donnent des valeurs de hachages très distinctes.

La fonction de hachage `hashCode()` définie par défaut par Java sur les `string` est de bonne qualité.

Chapitre 14

Graphes

Une partie des algorithmes sur les graphes ont déjà pu être vus plus tôt dans vos cursus. Ce chapitre vise à :

1. Motiver l'étude des graphes et donner les définitions mathématiques.
2. Présenter les questions algorithmiques et un certain nombres d'algorithmes spécifiques au graphes, en particulier les parcours.
3. Présenter les manières habituelles de les représenter, en particulier en utilisant les outils et les bibliothèques Java.

Un certains nombres de problèmes algorithmiques classiques sur les graphes ne sont pas traités ici. Par exemple l'algorithme Kruskal (arbre couvrant minimal), les algorithmes de Floyd-Warshall et de Bellman-Ford qui relèvent de la programmation dynamique, ou encore les questions de *flot maximal*, sont traités en deuxième année, cours INF421 (voir aussi (Cormen et al., 2009; Sedgewick and Wayne, 2011; Dasgupta et al., 2008; Kleinberg and Tardos, 2006)).

14.1 Principes et exemples

Les graphes sont des structures mathématiques qui sont particulièrement courantes en informatique. Aussi, de nombreux algorithmes importants et remarquables portent sur eux. Comme le nom l'indique, l'idée de ce qu'est un graphe se comprend bien avec un dessin ; c'est un ensemble de *sommets* qui peuvent être joints deux-à-deux par des *arêtes*. Lorsque ces arêtes ont un sens, généralement représenté par une flèche, on parle de *graphe orienté* ; ces arêtes orientées sont également appelées *arcs*.

Lorsque l'on abstrait les propriétés d'un système réel, on trouve très souvent un graphe. Les sommets comme les arêtes pouvant correspondre à des objets et liens soit matériels (villes, routes. . .) ou bien conceptuels (liens d'amitiés. . .). Dans tous les cas, un problème courant est de trouver un *chemin* dans un graphe, joignant un sommet à un autre. Voici quelques exemples réels importants ; ils sont grossièrement ordonnés du plus matériel au plus abstrait :

> **Réseau routier** Un réseau de transport (routier, ferroviaire, mixte Métro, bus et tram. . .) peut être vu comme un graphe dont les villes ou les stations sont les sommets et les routes ou voies ferrés les arêtes. Le graphe sera orienté ou non suivant que ces liaisons peuvent être à sens unique ou non. Trouver un chemin d'un point à un autre est le travail typique d'un logiciel de guidage.

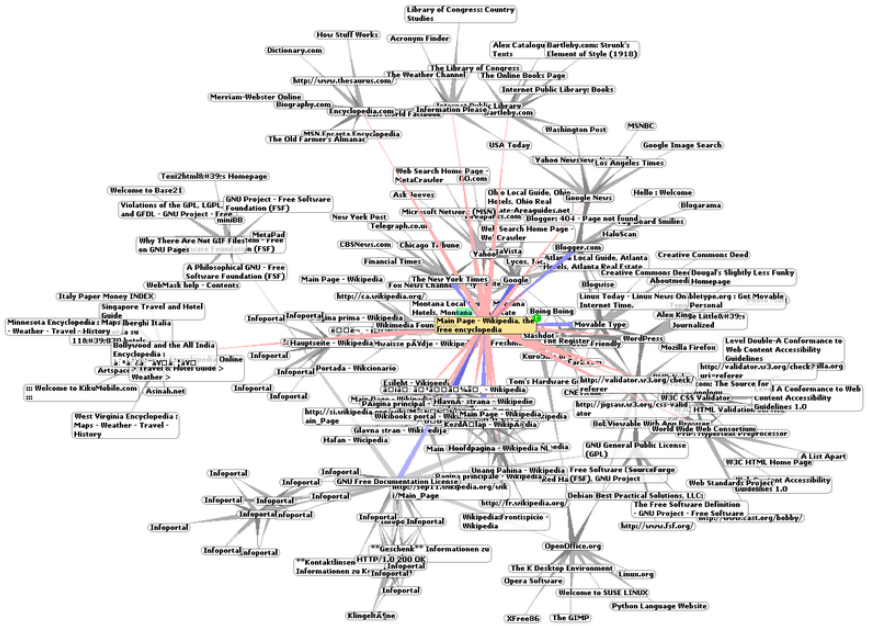


FIGURE 14.1 – Une partie du graphe du web (source wikipedia). Ici les sommets correspondent en fait à des ensembles de pages.

Les réseaux électriques ou hydrauliques sont d'autres exemples clairs. Ils illustrent bien les questions de *débit maximal* [1] qui apparaissent dans de nombreux algorithmes intéressants.

Réseaux de télécommunication Les noeuds des réseaux physiques de communication (téléphones fixes, téléphone mobiles, internet. . .) sont également les sommets d'un graphe dont les liaisons filaires ou radio sont les arêtes. La question de trouver un chemin vers une destination prend alors une tournure différente, la difficulté principale est que de très nombreux paquets ou messages doivent trouver une route vers leurs destination en parallèle. C'est la question du *routage*.

Réseau sociaux L'information, précieuse, dont dispose les opérateurs de réseaux sociaux, est un graphe dont les utilisateurs sont des sommets et les liens d'amitiés les arêtes. Cela permet d'ajouter ensuite de nouveaux types d'arêtes représentant, par exemple, les centres d'intérêts, porteurs d'enjeux économiques importants (publicités ciblées. . .). On cherche alors, par exemple, à identifier des groupes de membres plus fortement reliés.

Le web l'ensemble mondial des pages web est un exemple connu de graphe orienté. Les pages en sont des sommets, et les arcs indiquent qu'une page pointe vers une autre. La manière ont sont organisés ces pointeurs est cruciale pour permettre aux moteurs de recherche de déterminer quelles sont les pages intéressantes. La taille de ce graphe (on parle de mille milliards de sommets) rend les questions d'efficacité algorithmique cruciales.

Un jeu comme les échecs a une structure de graphe sous-jacente : les sommets sont les états du jeu, les arcs correspondent aux coups possibles, passant d'une position à une autre. Gagner correspond alors à trouver un chemin vers une position *mat*. Là encore,

1. Qui ne sont pas traitées dans ce cours, répétons-le. On peut, par exemple, se rapporter au chapitre 7 de (Kleinberg and Tardos, 2006) pour un traitement abordable et complet du sujet.

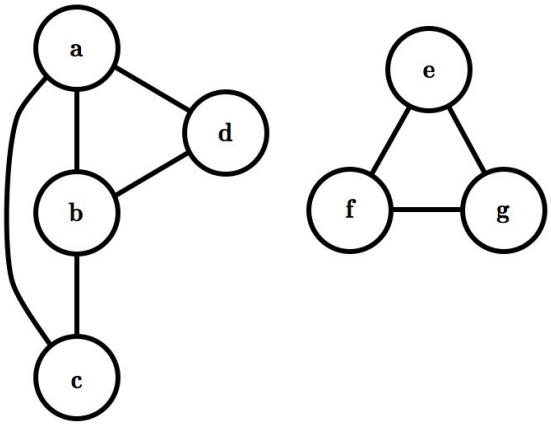


FIGURE 14.2 – Un graphe non-orienté avec deux composantes connexes : $\{a; b; c; d\}$ et $\{e; f; g\}$.

c'est le grand nombre de coups et de positions, c'est-à-dire la taille du graphe, qui rend ces questions d'exploration difficiles et intéressantes.

La planification de tâches Lorsque l'on est face à un large nombre de tâches à accomplir, dont certaines doivent être effectuées avant d'autres, on est en face d'un *graphe de dépendance*. Il y a un arc (orienté) de A vers B lorsqu'il est nécessaire d'avoir effectué A avant de commencer B. Il faut alors, par exemple, trouver un ordre dans lequel effectuer ces tâches. Ces questions d'ordonnancement sont présentes dans de nombreux secteurs d'activité. L'informatique elle-même en est une grande consommatrice ; un exemple sont *les logiciels complexes* composés de nombreux fichiers ou modules sont des graphes orientés : il y a un arc entre deux fichiers si le premier fait référence au second et doit donc être compilé après lui. Un autre exemple est la parallelisation partielle des calculs dans un processeur multi-cœurs.

En plus de la question de l'orientation ou non des arêtes, on précise en général, un certain nombre de points :

- Dans certains cas, le graphe considéré peut comporter des arcs (respectivement des arêtes) parallèles, c'est-à-dire s'il peut y avoir plus d'un arc (respectivement arêtes) joignant les mêmes sommets x à y. Lorsque c'est le cas, on parle parfois de *multigraphe*.
- Le graphe peut, ou pas comporter des *boucles*, c'est-à-dire des arêtes ou arc joignant un sommet à lui-même.
- Le nombre de sommets, comme celui d'arêtes ou d'arcs, peut être fini ou infini. On parle, pour simplifier, de *graphe fini* ou de *graphe infini*.

14.2 Définitions

Dans la plupart des cas et sauf mention contraire, nous considérerons des graphes finis, sans arcs ou arêtes parallèles. Cela nous permet d'utiliser les définitions simples ci-dessous.

Définition 14.2.1 (Graphe orienté, arc, successeur) Un *Graphe orienté* $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est donné par un ensemble $\mathcal{S}$ de *sommets* et un ensemble $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ d'arcs.

On dit que l'arc (x, y) joint le sommet x au sommet y, que x est *l'origine* de l'arc et y sa *destination*. On dit aussi que y est un *successeur* de x. ◇

Définition 14.2.2 (Graphe non-orienté, arête, voisin) Un *Graphe non-orienté* est donné par un ensemble $\mathcal{S}$ de *sommets* et un ensemble $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ d'arêtes, tel que

$$\forall (x, y), (x, y) \in \mathcal{A}, (y, x) \in \mathcal{A}.$$

En présence d'une arête (x, y), on dit que y est un *voisin* de x. ◇

Densité On note $|\mathcal{S}|$ le nombre de sommets et $|\mathcal{A}|$ le nombre d'arêtes. Les complexités algorithmiques dépendront typiquement de ces deux dimensions. Le nombre maximal d'arêtes est évidemment $|\mathcal{S}|^2$, et un graphe dont le nombre d'arêtes est relativement proche de ce maximum est dit *dense*. Un graphe peu dense est dit creux ; on utilise souvent l'adjectif anglais *sparse*.

Etant donné un graphe orienté, on passe au graphe non-orienté *sous-jacent* en transformant les arcs en arêtes.

Définition 14.2.3 (Boucle) Une boucle est un arc ou une arête joignant un sommet x à lui-même. ◇

14.2.1 Sous-graphes

Essentiellement, on passe d'un graphe à un *sous-graphe* en enlevant un certain nombre de sommets et/ou d'arcs/arêtes. Plus précisément.

Définition 14.2.4 (Sous-graphe) Soit un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$. Un graphe $\mathcal{G}' = (\mathcal{S}', \mathcal{A}')$ est un sous-graphe de $\mathcal{G}$ si et seulement si

$$\mathcal{S}' \subset \mathcal{S} \qquad \text{et} \qquad \mathcal{A}' \subset \mathcal{A}.$$

Définition 14.2.5 (Sous-graphe induit) Soit un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$. Etant donné $\mathcal{S}' \subset \mathcal{S}$, le sous-graphe de $\mathcal{G}$ induit par $\mathcal{S}'$ est définit comme $(\mathcal{S}', \mathcal{A}')$ où $\mathcal{A}'$ est l'ensembe des arcs (arêtes) de $\mathcal{A}$ qui joignent deux sommets de $\mathcal{S}'$. ◇

14.2.2 Chemins, accessibilité et connexité

Une question récurrente est, étant donné un graphe, de se rendre d'un sommet A à un sommet B en ne parcourant que de arcs/arêtes. C'est ce qu'on appelle trouver un *chemin* de A à B. C'est évidemment l'une des tâches premières d'un logiciel de navigation. En général, on cherchera un chemin qui optimise certaines mesures (le minimum d'arêtes, le moins de kms posibles...).

Définition 14.2.6 (Chemin, chemin simple) Un chemin de x à y est donné par une suite d'arcs (resp. d'arêtes) $a_0, \ldots, a_n$ et de sommets $z_0, \ldots, z_{n+1}$ tels que a_i joigne z_i à z_{i+1} et $z_0 = x$ et $z_{n+1} = y$.

Le chemin est *simple* s'il ne passe pas deux fois par le même sommet, c'est-à-dire que les z_i sont distincts deux-à-deux.

On remarque qu'il existe un chemin vide de tout sommet x à lui-même. ◇

Définition 14.2.7 (Accessible, connecté) S'il existe un chemin de x à y, on dit que y est accessible à partir de x.

Si le graphe est non-orienté, cette relation est symétrique et on dit aussi que x et y sont connectés. Il est clair que :

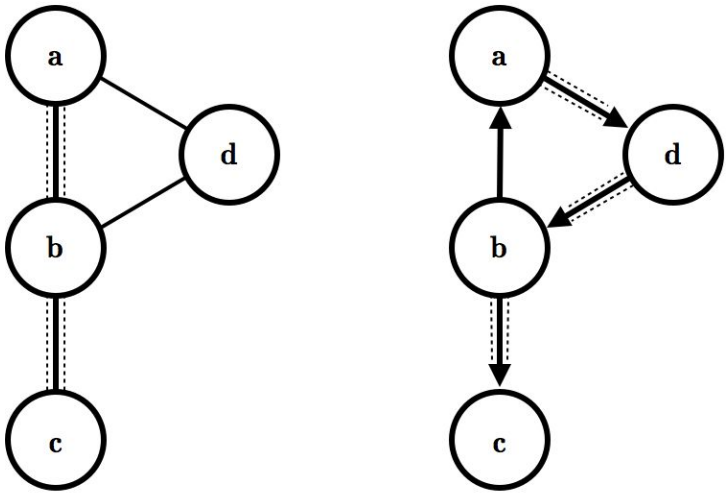


FIGURE 14.3 – Chemin de a à c dans un graphe non-orienté et dans un graphe orienté.

- Cette relation est transitive : s'il existe un chemin de x à y et un chemin de y à z, alors il existe un chemin de x à z.
- Dans le cas d'un graphe non-orienté, cette la relation de connection est donc une relation d'équivalence. ◇

Définition 14.2.8 (Connexe) Un graphe non-orienté est connexe si entre deux sommets quelconques il existe toujours un chemin. ◇

Définition 14.2.9 (Partie connexe) Etant donné un graphe non-orienté $\mathcal{G} = (\mathcal{S}, \mathcal{A})$. Un ensemble de sommets $\mathcal{S}' \subset \mathcal{S}$ est connexe si le sous-graphe induit par $\mathcal{S}'$ est connexe. ◇

Définition 14.2.10 (Composante connexe) Une composante connexe $\mathcal{S}'$ d'un graphe non-orienté est une classe d'équivalence de la relation de connexion. C'est donc aussi un partie connexe qui est maximale au sens de l'inclusion. C'est-à-dire :

Soit $x \in \mathcal{S}'$; alors quel que soit $y \in \mathcal{S}$, il existe un chemin de x à y si et seulement si $y \in \mathcal{S}'$.

Les composantes connexes forment donc une partition de $\mathcal{S}$. ◇

Pour les graphes orientés, l'existence d'un chemin de x à y n'implique pas forcément l'existence d'un chemin inverse de y à x. La notion intéressante est alors la *forte connexité*.

Définition 14.2.11 (Fortement connectés) Deux sommets x et y d'un même graphe sont fortement connectés s'il existe un chemin de x à y et un chemin de y à x. ◇

Définition 14.2.12 (Fortement connexe) Un graphe orienté est fortement connexe si quels que soient les sommets x et y, ils sont fortement connectés. ◇

Définition 14.2.13 (Partie fortement connexe) Etant donné un graphe orienté $\mathcal{G} = (\mathcal{S}, \mathcal{A})$. Un ensemble de sommets $\mathcal{S}' \subset \mathcal{S}$ est fortement connexe si le sous-graphe induit par $\mathcal{S}'$ est fortement connexe.

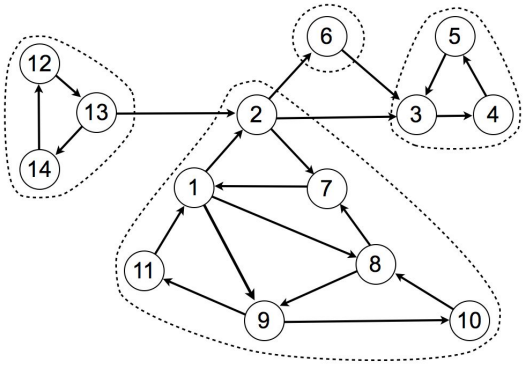


FIGURE 14.4 – Les composantes fortement connexes d'un graphe orienté.

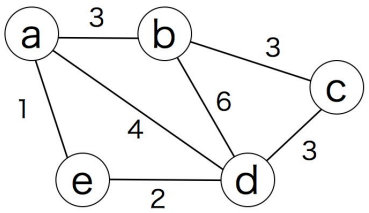


FIGURE 14.5 – Un graphe valué (non-orienté).

Une *composante fortement connexe* de $\mathcal{G}$ est une classe d'équivalence pour la relation de connexion forte. C'est-à-dire aussi un ensemble de sommets $\mathcal{S}'$ fortement connexe qui est maximale pour l'inclusion :

Soit $x \in \mathcal{S}'$; alors quel que soit $y \in \mathcal{S}$, $y \in \mathcal{S}'$ si et seulement y et x sont fortement connectés.$\diamond$

La figure 14.4 montre la décomposition d'un graphe orienté en composantes fortement connexes. Déterminer les composantes fortement connexes d'un graphe orienté est un problème particulièrement intéressant traité à la fin de ce chapitre.

14.2.3 Cycles

Définition 14.2.14 (Circuit, cycle) Un circuit est un chemin joignant un sommet x à lui-même. Un circuit simple est appelé un cycle. $\diamond$

On verra ci-dessous qu'un graphe non-orienté sans cycles peut être vu comme un ensemble d'arbres. La présence de cycles dans un graphe orienté est une question algorithmique courante.

14.2.4 Graphes Valués

Les algorithmes intéressants considèrent souvent des graphes munis d'informations supplémentaires. Un cas courant est celui ou les arêtes, ou les arcs, portent une *valeur numérique*. Cette valeur peut typiquement correspondre à la longueur de la route, au débit maximal du tuyau, à l'argent qu'il faudra dépenser pour la parcourir (essence + péage), etc. . . On cherchera alors à minimiser ou maximiser la valeur associée ; par exemple la longueur du chemin, le temps de parcours ou l'essence consommée.

La figure 14.5 représente un graphe valué. On traitera d'algorithmes sur des graphes valués dans les deux chapitres suivants.

14.3 Représentations informatiques

La représentation en mémoire d'un graphe est naturellement moins canonique que celle d'une liste par exemple. En effet, un graphe n'a pas d'élément privilégié qui correspondrait au premier élément d'une liste ou à la racine d'un arbre. On évidemment privilégier une représentation ou une autre suivant les opérations algorithmiques qu'on va effectuer. De plus, la géométrie peut varier beaucoup d'un graphe à l'autre : la rapport entre le nombre de sommets et le nombre d'arêtes, qui décrit la *densité* du graphe peut inciter à privilégier une représentation ou une autre.

14.3.1 Matrices d'adjacence

Principe

Cette représentation n'est pas la plus courante et on la mentionne bien qu'elle ne soit pas centrale dans ce cours. Elle est surtout utilisable pour les graphes denses. On représente les sommets par leur numéro de 0 à $n-1$. On représente ensuite l'information d'adjacence avec une matrice, c'est-à-dire aussi un tableau M à deux dimensions de taille $n \times n$. La valeur en $M_{i,j}$ indique s'il existe un arc entre les sommets i et j :

$$M_{i,j} = 1 \iff (i,j) \in \mathcal{A} \qquad M_{i,j} = 0 \iff (i,j) \notin \mathcal{A}.$$

On voit tout de suite que :

- Cette représentation n'autorise pas les arcs ou arêtes parallèles,
- cette représentation fonctionne pour les graphes orientés ; un graphes non-orienté représenté par une matrice symétrique,
- on peut étendre cette représentation aux graphes ayant des *arcs valués* ; $M_{i,j}$ prend alors la valeur de l'arc correspondant ; il faudra alors choisir une valeur correspondant à l'absence d'arc (suivant le contexte 0, l'infini, etc. . .).

La représentation matricielle est gourmande en mémoire et n'est pas utilisable pour les graphes comportant trop de sommets. Lorsque la densité est faible, on voit aussi qu'il faut parcourir une ligne de la matrice, c'est-à-dire $|\mathcal{S}|$ cases, pour énumérer tous les voisins d'un sommets. En revanche, lorsque l'on peut l'utiliser, elle permet de calculer la connexité, ou la distance entre toutes les paires de sommets de manière efficace et élégante par produit de matrices pour calculer la *clôture transitive du graphe*.

Chemins par produit matriciel

Théorème 14.3.1 Soit M^p la puissance p-ième de la matrice M. La valeur $M^p_{i,j}$ est le nombre de chemins de longueur p entre i et j. ◇

Démonstration Par récurrence sur p. C'est vrai pour $p = 1$ puisqu'un chemin de longueur 1 est exactement une arête. (On peut aussi dire que c'est vrai pour $p = 0$ si on considère q'un chemin de longueur 0 lie un sommet quelconque à lui même, et que M^0 est la matrice identité.)

Pour $p > 1$, un chemin de longueur p est un chemin de longueur $p-1$ entre i et un sommet k suivi d'une arête (k, j). Comme le nombre de chemins de longueur $p-1$ entre i et k est $M^{p-1}_{i,j}$, on peut déduire facilement que le nombre de chemins de longueur p entre i et j est bien $\Sigma^n_{k=0} M^p_{i,k}.M_{k,j}$. □

Corollaire 14.3.1 Soit $C = \Sigma^n_{k=0} M^p$. Alors $C_{i,j}$ est le nombre de chemins de longueur inférieure ou égale à n entre i et j. En particulier, $C_{i,j} \neq 0$ si et seulement si il existe un chemin entre i et j.◇

Démonstration La première affirmation est une conséquence immédiate du lemme précédent. La seconde découle du fait qu'un chemin, une fois qu'on a ôté les cycles, est de longueur au plus n. □

On sait donc calculer une matrice indiquant l'existence de chemins de longueur quelconque entre tous sommets i et j. En particulier, une fois cette matrice calculée, on sait déterminer en temps constant l'existence, ou non, d'un chemin entre i et j.

Cette approche ramène la question du calcul de chemins et de distance à celui du produit matriciel, qui est un problème algorithmique assez riche et étudié, par exemple en INF421.

14.3.2 Listes de voisins

Lorsque le graphe est moins dense, c'est-à-dire que $|\mathcal{A}|$ est fortement inférieur à $|\mathcal{S}|^2$, la représentation matricielle est rapidement trop gourmande en mémoire. On utilise alors des représentations où, à chaque sommet on sait associer la liste de ses voisins (ou de ses successeurs, si le graphe est orienté).

En Java, si les sommets d'un graphe sont représentés par des éléments d'une classe `Node` (les sommets d'un graphe sont appelés *the graph's nodes* en anglais), alors on pourra par exemple représenter un graphe par une table de classe `HashMap<Node, LinkedList<Node>>`.

Dans le pseudocode des algorithmes décrits ci-dessous, on utilisera la notation *s.voisins* pour la liste des voisins (ou successeurs) d'un sommet *s*.

En plus de limiter la mémoire nécessaire, une telle représentation permet d'accéder plus rapidement au voisins d'un sommet ; avec les matrices d'adjacence, il faut parcourir les indices correspondant à tous les sommets du graphe à chaque fois.

On supposera une telle représentation dans la suite de ce chapitre. Remarquons qu'elle est beaucoup plus proche de la géometrie du graphe que ne l'est la représentation matricielle.

14.4 Graphes acycliques non-orientés : Arbres

Un cas particulier important est celui de graphes qui ne comportent pas de cycles, ou de graphes *cycliques*. Cette propriété correspond à des réalités très différentes, suivant que l'on parle de graphes orientés ou non. C'est pourquoi nous séparons les deux cas.

La question de cycles dans un graphe orienté est traitée par un parcours en profondeur d'abord ; voir 14.6.3.

Un arbre peut être vu comme un graphe : les noeuds en sont les sommets, les liens père-fils les arêtes. Du coup, les arbres sont des graphes particuliers : ils sont connexes et ne possèdent pas de cycles. On peut caractériser plus précisément les graphes non-orientés qui sont des arbres :

Théorème 14.4.1 Soit $\mathcal{G}$ un graphe non-orienté de n sommets. Ces propriétés sont équivalentes et caractérisent le fait que $\mathcal{G}$ est un arbre :

1. $\mathcal{G}$ est connexe et sans cycles ;
2. $\mathcal{G}$ est connexe et a $n-1$ arêtes ;
3. $\mathcal{G}$ n'a pas de cycles et $n-1$ arêtes ;
4. étant donnés deux sommets de $\mathcal{G}$, ils sont reliés par une et une seule chaîne ;
5. $\mathcal{G}$ est connexe et cette connexité disparaît dès que l'on efface une arête quelconque ;
6. $\mathcal{G}$ n'a pas de cycles mais l'ajout d'une arête crée toujours un cycle ; ◇

On montre d'abord trois propriétés auxiliaires :

Lemme 14.4.1 Un graphe sans cycle est muni au moins d'un sommet ayant au plus un voisin.◇

Démonstration Si le graphe ne possède que un ou deux sommets, c'est évidemment le cas. Si le graphe possède un sommet s_1 ayant un voisin s_2 et que tous les sommets admettent au moins deux voisins, alors s_2 admet un voisin s_3 distinct de s_1. Puis s_3 un voisin s_4 distinct de s_2, etc. On construit ainsi une suite $(s_i)_{i\in\mathbb{N}}$ infinie. Comme le graphe n'est pas infini, cette suite doit comporter un circuit dont on peut extraire un cycle, ce qui contredit l'hypothèse initiale.□

Lemme 14.4.2 Tout graphe connexe de n sommets est muni d'au moins $n-1$ arêtes. ◇

Démonstration Si le graphe comporte un cycle, en retirant une arête de ce cycle on préserve la connexité et on diminue le nombre d'arêtes. On se ramène donc au cas ou le graphe est sans cycle.

On peut alors raisonner par récurrence. La propriété est vraie pour $n = 1$. Si $n > 1$, d'après le lemme précedent, il existe un sommet s qui admet une seule arête adjacente. Lorsque l'on supprime ce sommet et cette arête, on peut appliquer l'hypothèse de récurrence pour conclure. □

Lemme 14.4.3 Tout graphe de n sommets et au moins n arêtes comporte un cycle. ◇

Démonstration On peut montrer le lemme pour chaque composante connexe, et donc supposer le graphe connexe.

On raisonne alors par récurrence sur n. Le lemme est vrai pour $n = 1$. Si $n > 1$, on choisit une arête joignant deux sommets distincts. Comme il n'y a pas de cycle, en suppriment cette arête, on crée deux composantes connexes distinctes. Si ces deux composantes comportent respectivement n_1 et n_2 sommets, on a $n = n_1 + n_2$. Par hypothèse de récurrence, les deux composantes comportent au plus, respectivement, $n_1 - 1$ et $n_2 - 1$ arêtes. Le graphe de départ comportait donc bien au plus $n_1 + n_2 - 1 = n - 1$ arêtes. □

On peut maintenant prouver le théorème 14.4.1.

Démonstration On montre les implications successives.

(1) ⇒(2) Soit un graphe connexe et sans cycle de n sommets. D'après le lemme 14.4.2 il a au moins $n-1$ arêtes ; d'après le lemme 14.4.3 il en a au plus $n-1$.

(2) ⇒(3) Si le graphe admettait un cycle, on pourrait supprimer une arête en préservant la connexité. On aurait alors un graphe connexe de $n-2$ arêtes, ce qui contredirait le lemme 14.4.2.

(3)⇒(2) Si le graphe possède plusieurs composantes connexes, dont le nombre de sommets est respectivement $n_1, \ldots, n_k$, on a $n = n_1 + \ldots + n_k$. On voit facilement qu'au moins une composante connexe i admet strictement plus de $n_i - 1$ arêtes et comporte donc un cycle. Le graphe est donc connexe.

(2) ∧ (3) ⇒ (4) Le graphe est connexe, donc deux sommets sont joints par au moins une chaîne. S'ils étaient joints par deux chaînes distinctes, on pourrait exhiber un cycle.

(4) ⇒ (5) Toute paire de sommet est jointe par une chaîne, le graphe est donc connexe.

Si l'on supprime l'arête (s_1, s_2), alors il ne peut plus exister de chemin entre s_1 et s_2, car sinon ces deux sommets seraient joints par deux chaînes distinctes dans le graphe initial.

(5) ⇒ (6) S'il existait un cycle, comportant deux sommets consécutifs s_1 et s_2, on pourrait supprimer l'arête entre ces deux sommets en préservant la connexité. Il n'y a donc pas de cycle.

Par ailleurs, soient deux sommets distincts s_1 et s_2. Le graphe est connexe, ils sont donc joints par une chaîne. Aussi l'ajout d'une arête entre s_1 et s_2 crée un cycle.

(6) ⇒ (1) On montre que le graphe est connexe : soient deux sommets s_1 et s_2 qui ne sont pas voisins. L'ajout de l'arête (s_1, s_2) crée un cycle. Ce cycle, débarrassé de l'arête (s_1, s_2) donne un chemin de s_1 à s_2 dans le graphe originel. □

Arborescence Lorsqu'un arbre est considéré comme un graphe non-orienté, on ne distingue plus quel sommet est la racine. C'est pourquoi on appelle parfois *arborescence* un graphe possédant la propriété d'être un arbre et muni d'une racine distinguée.

Le lemme précédent dit essentiellement qu'un arbre est un graphe possédant juste assez d'arêtes pour préserver la connexité. Inversement, on peut obtenir un arbre en retirant des arêtes d'un graphe connexe :

Définition 14.4.1 (Arbre couvrant) Un arbre couvrant d'un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est un sous-graphe $(\mathcal{S}, \mathcal{A}')$ de $\mathcal{G}$, possédant la propriété d'être un arbre.

Si le graphe a plusieurs composantes connexe, on parlera de forêt couvrante. ◇

On verra ci-dessous qu'un arbre couvrant est obtenu par un parcours du graphe. Un problème algorithmique courant est de trouver un arbre couvrant de poids minimal, lorsque les arêtes du graphe sont valués. Cela correspond à trouver un ensemble minimal d'arêtes qui préserve la connexité (par exemple trouver une manière minimale de construire un réseau irriguant tous les sommets).

14.5 Vérifier l'existence d'un chemin

Vérifier si deux sommets sont reliés par un chemin dans un graphe non-orienté est traité facilement en utilisant une structure union-find (chapitre 10). Il suffit d'effectuer une opération union pour chaque arête.

En revanche, cela ne permet pas de déterminer *quel* est le chemin entre deux sommets. Pour cela, il faut utiliser des algorithmes de parcours de graphes.

14.6 Parcours de graphes

Une vaste famille de problèmes est résolue en *parcourant* un graphe, ou au moins une de ses composantes connexes : on part d'un sommet, passe à un de ses voisins.

14.6.1 Généralités

On verra que les parcours de graphes peuvent répondre à des buts très divers ; dans tous les cas, le principe d'un parcours est de *visiter* l'ensemble de la composante connexe. A chaque étape on peut donc distinguer états pour les sommets :

1. les sommets pas encore visités,
2. les sommets en traitement, ce sont par exemple des sommets déjà visités, dont certains voisins n'ont pas été visités,
3. les sommets déjà traités, c'est-à-dire que le travail de parcours est achevé pour ces sommets.

Initialement, tous les sommets sont à l'état (1), sauf le sommet de départ qui est à l'état (2). Une étape du parcours consiste alors à :

- choisir un sommet s en traitement (2),
- si tous les voisins de s ont été visités, c'est-à-dire qu'aucun n'est à l'état (1), alors s passe à l'état (3),
- sinon on choisi un de des voisins de s non visités, qui passe de l'état (1) à l'état (2).

Le parcours se termine lorsque tous les voisins (de la composante connexe) sont à l'état (3). On a alors visité tous les sommets, dans un ordre qui dépend de la manière dont sont choisis les sommets visités à chaque étape.

> Dans tous les cas, au cours d'un parcours, il faut se souvenir des sommets déjà parcourus. Très souvent, on munira les sommets d'une marque permettant d'indiquer leur état.

En conséquence, dans les algorithmes de parcours, on voit les graphes comme des structures partiellement mutables. En pratique, en Java, on pourra aussi souvent préférer utiliser une table d'association par hachage pour associer un état à un sommet.

On peut donner une description générique du parcours. Dans la suite, on se donnera trois valeurs NonVu, Frontiere et Vu pour désigner les trois états possibles des sommets. Voici à gauche une version "nue" du parcours générique et, à droite, le même code instrumenté pour numéroter les sommets. Il s'agit de l'ordre dans lequel les sommets sont vus la *première fois*, c'est-à-dire une numérotation *préfixe*.

Parcours(s_0)
  $s_0.etat \leftarrow$ Frontiere;
  tant que ($\exists\, s, s.etat =$ Frontiere) **faire**
    si ($\exists\, s' \in s.voisins, s'.etat =$ NonVu)
    alors $s'.etat \leftarrow$ Frontiere
    sinon $s.etat \leftarrow$ Vu

Parcours(s_0)
  $compteur \leftarrow 0$;
  $s_0.etat \leftarrow$ Frontiere;
  tant que ($\exists\, s, s.etat =$ Frontiere) **faire**
    si ($\exists\, s' \in s.voisins, s'.etat =$ NonVu)
    alors { $s'.etat \leftarrow$ Frontiere; $s.num \leftarrow compteur$; $compteur++$; }
    sinon $s.etat \leftarrow$ Vu

Le type de parcours est défini par la manière dont on choisi, à chaque étape, le sommet de la frontière traité. Aussi, à chaque type de parcours correspond une structure de données pour stocker la frontière. On distingue classiquement deux sortes de parcours types : le parcours en profondeur d'abord et le parcours en largeur d'abord.

Arbre du parcours

Dans tous les cas, un parcours construit, au moins implicitement, un *arbre couvrant* de la composante connexe du sommet de départ. Ou, pour être plus précis, une *arborescence* couvrante, dont la racine est le sommet de départ. On peut expliciter cette arborescence,par exemple, en faisant pointer chaque sommet vers son père ; c'est possible en munissant chaque sommet d'un champ pour cela. Le code du parcours générique peut alors également être instrumenté pour construire l'arborescence :

Parcours(s_0)
  $s_0.etat \leftarrow$ Frontiere;
  tant que ($\exists\, s, s.etat =$ Frontiere) **faire**
    si ($\exists\, s' \in s.voisins, s'.etat =$ NonVu)
    alors $\begin{cases} s'.etat \leftarrow \text{Frontiere} \\ s'.pere \leftarrow s \end{cases}$
    sinon $s.etat \leftarrow$ Vu

14.6.2 Le parcours en largeur : BFS

On va voir que la DFS commence par partir au plus loin du sommet de départ. Le *parcours en largeur d'abord* ou *breadth-first search* (BFS) commence lui par parcourir les sommets à distance 1 du sommet de départ, puis à distance 2, etc... Il permet donc de calculer les distances de chaque sommet par rapport au sommet de départ.

On obtient naturellement un parcours en largeur d'abord en utilisant une file d'attente (FIFO, *first-in-first-out*) pour représenter la frontière.

Voici une description de BFS, qui calcule de plus, pour chaque sommet accessible sa distance depuis le sommet de départ.

BFS(s)
$\begin{cases} f \leftarrow newFIFO(); \\ s.state \leftarrow \text{Vu}; \\ s.dist \leftarrow 0; \\ f.push(s); \\ \textbf{tant que non } f.isEmpty() \textbf{ faire} \\ \quad t \leftarrow f.pop(); \\ \quad \textbf{pour } u \in t.voisins \textbf{ faire} \\ \quad\quad \textbf{si } u.state = \text{NonVu} \\ \quad\quad \textbf{alors} \begin{cases} u.dist \leftarrow t.dist + 1; \\ f.push(u); \end{cases} \end{cases}$

Tout comme la DFS, la complexité du parcours BFS pour un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est de $O(|\mathcal{A}| + |\mathcal{S}|)$.

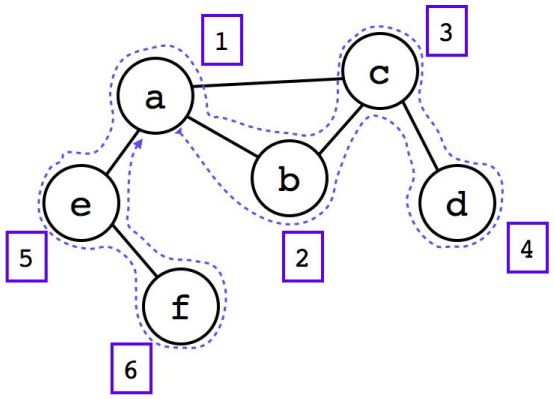


FIGURE 14.6 – Un parcours en profondeur d'abord (DFS)

14.6.3 Parcours en profondeur d'abord : DFS

L'algorithme

Le parcours en profondeur d'abord est souvent désigné par son acronyme anglais : DFS pour *depth first search*. Le principe du parcours en profondeur est simple :

- Le prochain sommet parcouru est un voisin (non encore visité) du dernier sommet visité.
- Si jamais tous les voisins du dernier sommet sont déjà visités, alors on cherche parmi les voisins de l'avant-dernier sommet visité, et ainsi de suite.

A noter que pour un graphe donné, et même en fixant le sommet de départ, il peut y avoir plusieurs parcours en profondeur possibles.

La figure 14.6 présente un parcours en profondeur possible, pour un graphe et un sommet de départ donnés.

Un parcours en profondeur d'abord est obtenu naturellement, lorsque l'on représente la frontière par un pile (LIFO, *last in first out*), puisqu'on poursuit le parcours à partir du dernier sommet visité.

Un premier intérêt du parcours en profondeur est qu'il est facile à programmer ; plus précisément à programmer récursivement. En effet, il n'est pas nécessaire de représenter explicitement la frontière. La *pile des appels récursifs* suffit. Le pseudo-code du parcours DFS :

DFS(*s*)
  s.state ← Frontiere
  pour *s'* ∈ *s.voisins* **faire**
    si *s'.state* = NonVu
    alors *DFS*(*s'*)
  s.state ← Vu

Dans le cas le plus simple, on programme donc DFS en n'utilisant que deux états pour les sommets : *Vu* et *NonVu* :

DFS(*s*)
s.state ← Vu
pour *s'* ∈ *s.voisins* **faire**
  si *s'.state* = NonVu
  alors *DFS*(*s'*)

Complexité Le parcours DFS traite donc chaque sommet et chaque arcs (ou arête). Suivant la densité du graphe, c'est le nombre d'arcs ou le nombre de sommets qui sera dominant. La complexité du parcours est en $O(|\mathcal{A}| + |\mathcal{S}|)$.

Etude de DFS

On voit que le parcours DFS a la compacité typique des fonctions récursives. Mais cette fonction récursive effectue un effet de bord : le marquage des sommets visités. Il faut donc tenir compte de cet effet de bord lorsque l'on énonce ce que fait cet algorithme.

Le lemme suivant décrit le comportement de DFS. Il peut paraître anodin à première vue, mais on verra qu'il permettra d'établir des résultats puissants.

Lemme 14.6.1 Le programme *DFS(s)* termine. A la fin de son exécution :

1. les sommets qui étaient marqués Vu au départ le sont encore,
2. les sommets qui au départ étaient accessibles depuis *s* par un chemin entièrement NonVu, sont marqués Vu,
3. les autres sommets ne sont pas marqués par l'exécution. ◇

Démonstration La première assertion est immédiatement vérifiée, puisque les sommets Vu ne changent plus d'état.

Il nous faut donc montrer que les seuls sommets marqués au cours de l'exécution sont ceux accessibles à partir de *s* par un chemin entièrement NonVu. Plus précisément on montre les deux assertions (2) et (3) par deux récurrences distinctes.

On montre (2) par récurrence sur la longueur du chemin. Si la longueur du chemin est 0 le seul sommet concerné est *s* lui-même, qui est effectivement marqué comme *Vu* par le programme à la fin de l'exécution.

Soit un sommet *s'* accessible depuis *s* par chemin de longueur $n + 1$ et entièrement NonVu. Ce chemin commence par un sommet *s''*, successeur de *s* et *NonVu*. Aussi, le programme exécute-t-il *DFS*(*s''*). Par hypothèse de récurrence, à l'issue de cette exécution, *s'* est marqué *Vu*.

On montre (3) par récurrence sur le nombre de sommets NonVu au début de l'exécution de *DSF*(*s*). Lors de l'appel de *DFS*(*s*), le sommet *s* est marqué Frontiere et ce nombre diminue. Ensuite, *DFS* est appelé uniquement pour des voisins *s'* de *s* qui sont NonVu. Par hypothèse de récurrence, chacun des ces appels ne marquera que des sommets accessibles depuis ces *s'* par des chemins non marqués. □

Corollaire 14.6.1 Si tous les sommets d'un graphe sont marqués NonVu avant l'exécution de *DFS(s)*, alors, après l'exécution, les sommets accessibles depuis *s* sont marqués Vu. Les autres sont NonVu.

Si le graphe est non-orienté, les sommets Vu à la fin de l'exécution sont exactement la composante connexe de *s*. ◇

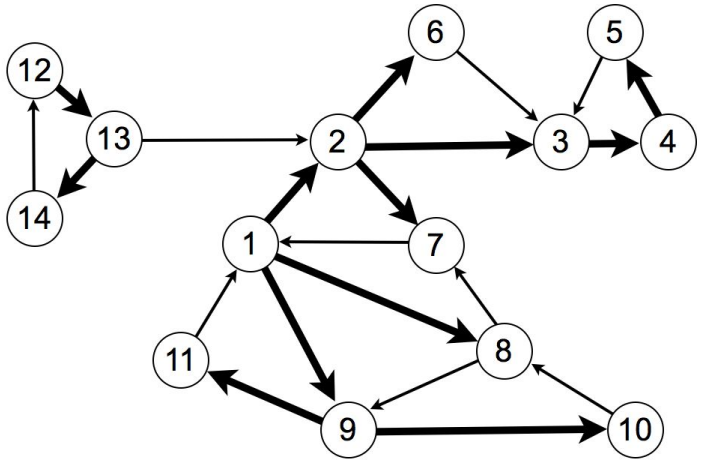


FIGURE 14.7 – Une numérotation DFS et les arcs de l'arborescence de Trémaux correspondante en gras.

Variantes de DFS

En pratique, on va utiliser des variantes de cette procédure pour diverses tâches. A chaque fois, le parcours reste le même, et donc le mode de raisonnement est similaire. Par exemple, voici comment on peut utiliser l'exploration DFS pour construire et numéroter les composantes connexes dans le cas où le graphe est non-orienté. A la fin, chaque sommet porte le numéro de sa composante.

```
DFSn(s, i)
  s.state ← Vu; s.comp ← i;
  pour s' ∈ s.voisins faire
    si s'.state ≠ Vu
  alors DFSn(s', i)

COMPOSANTE()
  i ← 0;
  tant que ∃ s . (s.state = NonVu) faire
    DFSn(s, i); i + +;
```

L'arborescence et la numérotation de Trémaux

L'arborescence construite au cours de la DFS est souvent appelée arborescence de Trémaux, du nom d'un ingénieur polytechnicien du XIX$^{\text{éme}}$ siècle qui a décrit un algorithme de sortie de labyrinthe pour un individu muni d'une simple craie. Cet algorithme étant essentiellement un DFS.

Cela permet de souligner que le parcours DFS correspond bien à un parcours que l'on peut effectuer dans un labyrinthe : on explore les sommets de proche en proche, alors que dans un parcours BFS, les sommets explorés successivement peuvent être très éloignés les uns des autres.

L'arborescence de Trémaux peut être construite en instrumentant le code DFS comme en 14.6.1. Mais surtout, l'étude de cette arborescence va nous être très utile pour divers algorithmes sur les graphes, particulièrement des graphes orientés. Pour cela, nous allons utiliser le *rang* des sommets, c'est-à-dire l'ordre de parcours [2]. On peut définir le rang en instrumentant le code du parcours :

```
DFSnum
  compteur ← 0;
  tant que ∃ s, s.state = NonVu faire
    DFSrec(s);

DFSrec(s)
  s.state ← Vu;
  s.num ← compteur;
  compteur + +;
  pour s' ∈ s.voisins faire
      si s'.state = NonVu
      alors DFSrec(s')
```

La figure 14.7 présente un graphe orienté avec les sommets numérotés dans un ordre DFS ; les arcs appartenant à l'arborescence correspondante étant en gras.

Lemme 14.6.2 Si un sommet s' est un descendant d'un sommet s pour l'arborescence de Trémaux, alors s' est accessible depuis s. ◇

La preuve est à peu près évidente, puisque l'arborescence suit un parcours du graphe.

Détection de cycles

Lorsque l'on utilise bien les trois états, Vu, NonVu et Frontiere, une très légère modification du parcours est suffisante pour détecter la présence de cycles dans un graphe orienté :

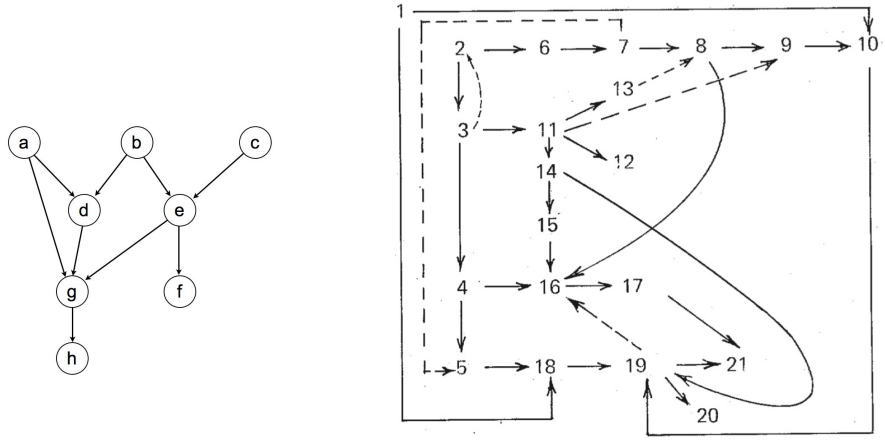
```
DFScycle
  tant que ∃ s.s.state = NonVu faire
    DFScycleRec(s);

DFScycleRec(s)
  s.state ← Frontiere;
  pour s' ∈ s.voisins faire
      si s'.state = NonVu
      alors DFScycleRec(s')
      sinon si s'.state = Frontiere
      alors émettre ("cycle");
```

La version précédente du parcours DFS permet facilement de détecter la présence de cycles dans un graphe. Pour cela, une propriété importante est la suivante :

Lemme 14.6.3 Supposons qu'on lance *DFScycleRec*(s) alors qu'aucun sommet n'est dans l'état Frontiere. Alors, à tout moment de l'exécution, l'ensemble des sommets marqués Frontiere forme un chemin simple d'origine s. De plus, au moment d'un appel récursif *DFScycleRec*(t), ce chemin joint s à t. ◇

Démonstration C'est vrai au début de l'exécution, qui commence par $s.state \leftarrow$ Frontiere. On voit qu'ensuite cet invariante est préservé. □

2. Pour être tout à fait précis, on peut remarquer que cette numérotation est dite *préfixe*, c'est-à-dire que le numéro est fixé dès que le sommet est visité. Dans une numérotation postfixe, le numéro est fixé lorsque tous les voisins ont été visités.



a Un graphe acyclique orienté, ou DAG. **b** Les dépendances entre les chapitres d'un livre.

FIGURE 14.8 – Deux graphes acycliques orientés - le second donne les ordres de lecture possibles du livre de Henk Barendregt (Barendregt, 1984)

Lemme 14.6.4 Le programme *DFScycle* envoie le message "cycle" si et seulement si le graphe comporte au moins un cycle. ◇

Démonstration Supposons que le programme émet le message "cycle". Soit t le sommet qui est en train d'être traité au moment où le message est émis. On a alors un arc de t vers un sommet u qui est dans l'état Frontiere. D'après le lemme précédent, t est accessible depuis u. Comme u est aussi accessible depuis t, on a bien un cycle.

Supposons maintenant l'existence d'un cycle. Soit t le premier sommet de ce cycle à passer dans l'état Vu. Soit u le sommet suivant t dans le cycle. Il existe donc un arc de t à u.

Lorsque *DFScycleRec*(t) est appelé, u ne peut donc pas déjà être dans l'état Vu ; il ne peut pas non plus être dans l'état NonVu, car sinon u serait un descendant de t dans l'arborescence de Trémaux, et serait marqué Vu avant t. Donc u est alors dans l'état Frontiere, et le programme détecte bien le cycle. □

14.6.4 Graphes Acycliques orientés : DAGs

Les graphes orientés acycliques sont souvent appelés DAGs, de l'anglais *Directed Acyclic Graphs*. On peut aussi caractériser les DAGs comme des graphes orientés où l'accessibilité est une relation d'ordre.

Un arbre est un cas particulier de DAG, mais un DAG a la possibilité de partager des sous-arbres (figure 14.8.a).

Les DAGs interviennent souvent dans les problèmes d'ordonnancement : organisation des différentes tâches sur un chantier ou les dépendances entres les différents modules d'un logiciel. On aura un arc (s, t) si s est nécessaire à t, c'est-à-dire que s doit être effectué avant t. Par exemple, le livre de Henk Barendregt (Barendregt, 1984) sur le λ-calcul, bien connu en informatique théorique, contient un tel DAG qui résume les dépendances entre ses chapitres (figure 14.8.b).

Tri Topologique

Il est possible d'ordonner totalement les sommets d'un DAG, d'une manière qui soit compatible avec l'ordre d'accessibilité. C'est ce qu'on appelle le *tri topologique*. Le tri topologique est effectué par une variante DFS. Il nous fournit un ordre possible pour lire les chapitres du livre de Barendregt, pour effectuer les tâches du chantier, etc. . .

```
TriTopo(G)
  F ← [];
  tant que ∃ s sommet de G . s.state = NonVu faire
    TriTopoRec(s);
  renvoyer (F);

TriTopoRec(s)
  s.state ← Frontiere;
  pour s' ∈ s.voisins faire
    si s'.state = NonVu
    alors TriTopoRec(s');
    sinon si s'.state = Frontiere
    alors émettre (Error("cycle"));
  s.state ← Vu;
  F ← s :: F;
```

On peut alors prouver la correction du tri topologique :

Théorème 14.6.1 Etant donné un graphe G, *TriTopo*(G) termine et :

- *TriTopo*(G) renvoie l'erreur "cycle" si et seulement si G contient un cycle,
- sinon, *TriTopo*(G) renvoie une liste F qui contient exactement tous les sommets de G telle que si s est accessible depuis s' alors s apparaît après s' dans F. ◇

Démonstration On voit que *TriTopoRec* est un parcours DFS et donc termine. On voit aussi que le programme renvoie une erreur si et seulement si il détecte un cycle (ou un arc arrière) ; il se comporte de ce point de vue exactement comme celui de la section 14.6.3.

On se place donc maintenant dans le cas où le graphe est acyclique.

Il est facile de voir qu'à la fin de l'exécution, chaque sommet de G apparaît une et une seule fois dans F. On voit également qu'au cours de l'exécution, les éléments de F sont exactement les sommets t qui sont Vu.

Au moment où *TriTopoRec*(s) est appelé, les seuls sommets à l'état Frontiere forment un chemin qui aboutit à s (lemme 14.6.3). Donc, s est accessible depuis ces sommets. Comme il n'y a pas de cycle, aucun sommet à l'état Frontiere n'est alors accessible depuis s.

Soit un sommet s' accessible depuis s par un chemin $s, s_1, \ldots, s'$. Lorsque *TriTopoRec*(s) est appelé, on a trois possibilités :

- Soit tout le chemin est NonVu. Dans ce cas, à la fin de l'exécution de *TriTopoRec*(s), s' aura été visité (lemme 14.6.1) et aura bien été rajouté à F.
- Soit un s_i est Vu ; et alors s_i appartient déjà à F. Cela veut dire que *TriTopoRec*(s_i) a déjà été exécuté ; donc s', qui est accessible depuis s_i a déjà été visité et rajouté à F.
- Soit un s_i est Frontiere. Mais on sait qu'alors s_i est accessible depuis s' et s' est accessible depuis s_i. Comme s' et s_i sont distincts, on a alors un cycle.

On a donc bien l'invariant suivant : à tout moment de l'exécution du programme, si x apparaît dans F, alors tous les sommets accessibles depuis x apparaissent dans F après x. □

14.6.5 Composantes fortement connexes (hors-programme)

On rappelle que la notion de composante fortement connexe (définition 14.2.13) n'est intéressante que pour les graphes orientés. Dans un DAG, les composantes fortement connexes sont réduites à des singletons. En présence de cycles en revanche, il devient important de déterminer quelles sont ces composantes fortement connexes.

Par la suite, on s'autorise l'abréviation anglaise SCC (*strongly connected component*) pour composante fortement connexe.

Le DAG des composantes fortement connexes

Une composante fortement connexe, lorsqu'elle n'est pas réduite à un seul sommet, contient forcément un cycle. En revanche, l'ensemble des composantes fortement connexes de tout graphe orienté constitue un DAG ; voir la figure 14.4.

Une fois déterminé l'ensemble des SCC, on peut donc, si nécessaire, trier topologiquement le DAG des SCCs.

Déterminer les SCCs

Il est facile de déterminer la SCC d'un sommet x de manière naïve. Une première exploration DFS à partir de s permet d'obtenir l'ensemble des sommets accessibles depuis s. Pour chacun de ces sommets s', une DFS permet de vérifier si s est également accessible depuis s', c'est-à-dire si $s' \in SCC(s)$. On doit alors effectuer jusqu'à $O(|S|)$ DFS, ce qui donne une complexité quadratique.

Il est possible de déterminer efficacement les SCCs, c'est-à-dire d'affecter à chaque sommet un numéro désignant de façon unique sa SCC, en effectuant une seule exploration DFS du graphe. L'algorithme correspondant est l'un des plus sophistiqués traités dans ce cours ; il a été proposé par l'informaticien américain Robert Tarjan [3] en 1972 (Tarjan, 1972). Cet algorithme n'est pas très long et n'utilise pas de structures de données complexes ; c'est sa preuve de correction qui est remarquablement subtile.

Lemme 14.6.5 Soit deux sommets s_1 et s_2 qui appartiennent à la même composante fortement connexe. Alors tout chemin de s_1 à s_2 est entièrement composé de sommets qui appartiennent également à cette composante connexe. ◇

Démonstration Elle est quasi immédiate. Si un s appartient à un chemin de s_1 à s_2, alors il est accessible à partir de s_1. Comme s_2 est accessible à partir de s et que s_1 est accessible à partir de s_2, il est clair que s_1 est aussi accessible à partir de s. Donc s et s_1 appartiennent à la même SCC. □

Un point clé de l'algorithme est d'identifier les SCC par leur **racine**, c'est-à-dire le sommet de plus faible rang de la SCC.

Lemme 14.6.6 Tous les sommets d'une composante connexe sont descendants de la racine dans l'arborescence de Trémaux. ◇

Démonstration Soit r la racine de la SCC, et s un sommet de la même SCC. Il existe un chemin de r à s, et ce chemin est entièrement inclus dans la SCC. En conséquence, lorsque r est numéroté et marqué Frontiere au cours de la DFS, tout le chemin de r à s est NonVu. Tout ce chemin, s compris, fait donc partie de la descendance de r (lemme 14.6.1). □

3. Tarjan est l'auteur de plusieurs algorithmes remarquables sur les graphes ; en particulier, également, un algorithme de test de planarité en temps linéaire, beaucoup plus compliqué encore que l'algorithme des SCCs.

```
Tarjan
  i ← 0;  // compteur pour les sommets
  c ← 0;  // compteur pour les SCCs
  pour s, s.state ← NonVu;
  F ← []  // Pile pour construire les SCCs
  tant que ∃ s, s.state = NonVu faire
    TarjanRec(s);

TarjanRec(s)
  s.state ← Frontiere;
  s.num ← i
  s.up ← i
  F ← i :: F
  i + +
  F ← s :: F
  pour s' ∈ s.voisins faire
    { si s'.state = NonVu
      alors { TarjanRec(s');
              s.up ← min(s.up, s'.up);
      sinon si s' ∈ F
      alors s.up ← min(s.up, s'.num);
  s.state ← Vu;
  si s.num = s.up  // s est racine d'une SCC
    alors { c + +;
            répéter
              t ← F.head;
              F ← F.tail;
              t.scc ← c;
            jusqu'à t = s;
```

FIGURE 14.9 – L'algorithme de Tarjan détermine les composantes fortement connexes en un seul parcours DFS.

Corollaire 14.6.2 Si s est racine d'une SCC pour une arborescence de Trémaux donnée, alors $s' \in SCC(s)$ si et seulement si s' est un descendant de s et il existe un chemin de s' vers s. ◇

Explication de l'algorithme

L'algorithme de Tarjan est donné en figure 14.9. On voit que :

- Il suit un déroulement de DFS classique et numérote chaque sommet s par son rang dans le champ $s.num$.
- Il calcule également, pour chaque sommet, un indice supplémentaire $s.up$; on verra que $s.up$ correspondra à ce qu'on appellera le *rang d'attache* de s.
- Le point crucial de l'algorithme est que s est la racine d'une SCC si et seulement si le rang de s est égal à son rang d'attache ; c'est-à-dire si $s.num = s.up$ à l'issue de $TarjanRec(s)$.

C'est ce dernier point dont nous allons ébaucher la démonstration. On peut déjà remarquer que, à la fin de l'exécution de $TarjanRec(s)$ on a :

- $s.up \leq s.num$,
- le sommet de rang $s.up$ est accessible depuis s.

Le premier point est évident, puisque $s.up$ est initialisé à la même valeur que $s.num$ et ne peut ensuite que baisser. Le second est une récurrence simple : les valeurs que peut prendre $s.up$ sont des $s'.up$ avec s' descendant de s, donc accessible depuis s.

On peut ensuite définir de manière non-algorithmique l'indice calculé par $s.up$; plus précisément :

Définition 14.6.1 Le rang d'attache d'un sommet s est défini comme le plus petit rang d'un sommet s' tel que :

- s' est accessible depuis s par un chemin ne comportant que des descendants de s dans l'arborescence de Trémaux [4],
- s' est lui-même soit égal à s, soit un ancêtre de s dans l'arborescence de Trémaux, soit un sommet depuis lequel un ancêtre de s est accessible. ◇

On verra plus bas que la valeur finale de $s.up$ est effectivement le rang d'attache de s. Pour l'instant, on remarque tout de suite que le rang d'attache de s est inférieur ou égal au rang de s. On voit aussi que :

Lemme 14.6.7 Le sommet dont le rang est le rang d'attache de s appartient à la même SCC que s. ◇

Démonstration Soit s' le sommet dont le rang est le rang d'attache de s. Par définition, on voit que s' est accessible depuis s, et vice-versa. □

Corollaire 14.6.3 Un sommet s est racine de $SCC(s)$ si et seulement si le rang de s est égal à son rang d'attache. ◇

On peut alors monter :

Théorème 14.6.2 A la fin de $TarjanRec(s)$, la valeur de $s.up$ est le rang d'attache de s. ◇

4. Du point de vue des arcs, cela signifie que tous les arcs de ce chemin, sauf le dernier, sont des arcs de l'arborescence de Trémaux.

Démonstration On raisonne par récurrence sur le nombre de sommets marqués Vu, c'est-à-dire pour lesquels le parcours est terminé. A un instant donné, l'hypothèse de récurrence signifie alors que les racines de SCC marquées Vu ont été détectées correctement. En conséquence, si un sommet est marqué Vu, mais encore élément de F, on sait que la racine de sa SCC est, à cet instant, marquée Frontiere. Donc, depuis tout sommet marqué Vu et élément de F, il existe un chemin vers un sommet marqué Frontiere. Cela permet de déduire que le calcul de $s.up$ correspond bien à celui du rang d'attache. □

Annexe A

la XVM

XVM							
instruction	pile avant	pile après	PC	SP	FP	R	condition ou effet
Opérations							
ADD	$x\ ;\ y\ ;\ z\ ;\ldots$	$x+y\ ;\ z\ ;\ldots$	$+1$	-1			
SUB	$x\ ;\ y\ ;\ z\ ;\ldots$	$y-x\ ;\ z\ ;\ldots$	$+1$	-1			
MULT	$x\ ;\ y\ ;\ z\ ;\ldots$	$x\times y\ ;\ z\ ;\ldots$	$+1$	-1			
DIV	$x\ ;\ y\ ;\ z\ ;\ldots$	$y/x\ ;\ z\ ;\ldots$	$+1$	-1			
EQ	$x\ ;\ x\ ;\ y\ ;\ldots$	$1\ ;\ y\ ;\ \ldots$	$+1$	-1			
EQ	$x\ ;\ y\ ;\ z\ ;\ldots$	$0\ ;\ z\ ;\ \ldots$	$+1$	-1			si $x\neq y$
LEQ	$x\ ;\ y\ ;\ z\ ;\ldots$	$1\ ;\ z\ ;\ \ldots$	$+1$	-1			si $y\leq x$
LEQ	$x\ ;\ y\ ;\ z\ ;\ldots$	$0\ ;\ z\ ;\ \ldots$	$+1$	-1			si $x<y$
NOT	$0\ ;\ y\ ;\ z\ ;\ \ldots$	$1\ ;\ y\ ;\ z\ ;\ \ldots$	$+1$				
NOT	$x\ ;\ y\ ;\ z\ ;\ \ldots$	$0\ ;\ y\ ;\ z\ ;\ \ldots$	$+1$				si $x\neq 0$
Instructions de saut							
GTO(m)	$x\ ;\ y\ ;\ z\ ;\ldots$	$x\ ;\ y\ ;\ z\ ;\ldots$	m				
GTO	$x\ ;\ y\ ;\ z\ ;\ldots$	$y\ ;\ z\ ;\ldots$	x	-1			
GSB(m)	$x\ ;\ y\ ;\ z\ ;\ldots$	PC $+1$; FP ; $x\ ;\ y\ ;\ \ldots$	m	$+2$	SP		
GSB	$x\ ;\ y\ ;\ z\ ;\ldots$	PC $+1$; FP ; $\ y\ ;\ \ldots$	x	$+1$	SP -1		
RET	$x\ ;\ y\ ;\ z\ ;\ldots$	$z\ ;\ \ldots$	$st(f+1)$	f	$st(f)$		où f est l'ancienne valeur de FP
GTZ(m)	$0\ ;\ y\ ;\ z\ ;\ \ldots$	$y\ ;\ z\ ;\ \ldots$	m	-1			
GTZ(m)	$x\ ;\ y\ ;\ z\ ;\ldots$	$y\ ;\ z\ ;\ \ldots$	$+1$	-1			si $x\neq 0$
Manipulations de la pile							
PUSH(a)	$x\ ;\ y\ ;\ z\ ;\ldots$	$a\ ;\ x\ ;\ y\ ;\ z\ ;\ \ldots$	$+1$	$+1$			
POP	$x\ ;\ y\ ;\ z\ ;\ldots$	$y\ ;\ z\ ;\ \ldots$	$+1$	-1			
PXR	$x\ ;\ y\ ;\ z\ ;\ldots$	$y\ ;\ z\ ;\ \ldots$	$+1$	-1		x	
PRX	$x\ ;\ y\ ;\ z\ ;\ldots$	$R\ ;\ x\ ;\ y\ ;\ \ldots$	$+1$	$+1$		R	
FETCH(i)	$x_0\ ;\ x_1\ ;\ldots;\ x_j\ ;\ldots$	$x_i\ ;\ x_0\ ;\ x_1\ ;\ldots;\ x_j\ldots$	$+1$	$+1$			
WFR(i)	$x;y;\ldots;x_i;\ldots$	$y;\ldots;x;\ldots$	$+1$	-1			où x_i est la i-éme position au-dessus de FP
WFR($-i$)	$x;y;\ldots;x_i;\ldots$	$y;\ldots;x;\ldots$	$+1$	-1			où x_i est la i-éme position en-dessous de FP
RFR(i)	$x;y;\ldots;x_i;\ldots$	$x_i;x;y;\ldots;x_i;\ldots$	$+1$	$+1$			où x_i est la i-éme position au-dessus de FP
RFR($-i$)	$x;y;\ldots;x_i;\ldots$	$x_i;x;y;\ldots;x;\ldots$	$+1$	$+1$			où x_i est la i-éme position en-dessous de FP
Accès mémoire							
READ	$x\ ;\ y\ ;\ z\ ;\ldots$	mem(x) ; $y\ ;\ z\ ;\ \ldots$	$+1$				
WRITE	$x\ ;\ y\ ;\ z\ ;\ldots$	$z\ ;\ \ldots$	$+1$	-2			mem(x) $\leftarrow y$
CREAD	$x\ ;\ y\ ;\ z\ ;\ldots$	mem($x+y$) ; $z;\ldots$	$+1$	-1			
CWRITE	$x\ ;\ y\ ;\ z\ ;\ t\ ;\ldots$	$t\ ;\ldots$	$+1$	-3			mem($x+y$) $\leftarrow z$
Utilisation du tas							
ALLOC	$x\ ;\ y\ ;\ z\ ;\ldots$	$a\ ;\ y\ ;\ z\ ;\ldots$	$+1$				a : nouvelle adresse avec x mots réservés

On résume les instructions de la XVM et leur comportement avec, pour chacune, leurs effets respectifs sur la pile, sur le compteur de programme PC, le pointeur de pile SP, le pointeur de page FP et le registre R.

Dans les colonnes de ces registres (PC, SP, FP et R) on laisse la case vide lorsque l'instruction ne modifie pas la valeur du-dit registre. Dans les colonnes PC et SP, +1 signifie que la valeur du registre est augmentée de un, −2 qu'elle est décrémentée de deux, etc. Aussi, en général, on trouve +1 dans la colonne PC ce qui signifie qu'on passe ensuite à l'instruction suivante ; les seules exceptions étant les instructions de saut (GTO, GTZ et GSB).

Dans la colonne SP, +1 signifie aussi que le nombre d'éléments sur la pile a augmenté de 1, +2 qu'il a augmenté de 2 etc.

Remarques

Sauts

On présente ci-dessus deux versions des instructions de saut GTO et GSB. Rappelons (6.9.2) qu'il est possible de voir GTO(n) comme une abréviation de PUSH(n); GTO, et de même GSB(n) comme une abréviation de PUSH(n); GSB. C'est-à-dire que les versions GTO(n) et GSB(n) sont redondantes et pourraient être supprimées. Nous choisissons toutefois de les faire apparaitre car elles sont utilisées au début de la présentation de la machine, en particulier dans les chapitres 2 et 3.

GC

On rappelle également que le GC (*garbage collector*, présenté en 5.7) est supposé fonctionner automatiquement, donc n'a pas à être appelé par l'utilisateur. Ce dernier point est spécifique à une machine virtuelle ; ce serait évidemment différent pour un processeur physique où le GC doit être programmé et appelé.

Annexe B

Solutions aux exercices

Solution de l'exercice 1.1.1 Il va afficher `world`. En effet, seule l'instruction immédiatement après `if (false)` fait partie de la conditionnelle. Formellement, ce programme est donc la séquence formée par :

- l'instruction de test `if (false) System.out.println("Hello ");`
- l'instruction `System.out.println("world");` ◇

On peut remarquer que le programme est mal indenté, mais cela n'est pas pris en compte par le compilateur.

Si on avait voulu faire porter le test sur les deux instructions d'affichage, on aurait du écrire :

```
if (false) {
  System.out.println("Hello ");
  System.out.println("world");
}
```

Solution de l'exercice 2.1.1 Par exemple :

PUSH(5); PUSH(3); SUB; PUSH(9); MUL; PUSH(2); SUB. ◇

Solution de l'exercice 2.1.2 Il faut penser à dupliquer la valeur en haut de la pile pour faire le test. Une possibilité est :

$$(\alpha)\mathsf{FETCH}(0); \mathsf{GTZ}(\beta); \mathsf{PUSH}(1); \mathsf{SUB}; \mathsf{GTO}(\alpha); (\beta)\mathsf{STOP}.$$

Si on suppose que la valeur en haut de la pile est strictement positive au début, on peut aussi écrire :

$$(\alpha)\mathsf{PUSH}(1); \mathsf{SUB}; \mathsf{FETCH}(0); \mathsf{NOT}; \mathsf{GTZ}(\alpha); \mathsf{STOP}.$$

Solution de l'exercice 2.3.1 On utilise le fait que `e1 < e2` est identique à `!(e2 <= e1)`. On peut donc prendre la séquence : ⟦`e2`⟧ ⟦`e1`⟧ LEQ NOT.

Remarquons que être si on veut être absolument précis, cette solution n'est pas tout à fait correcte, car elle revient à évaluer `e2` avant `e1`, alors que Java précise qu'il faudrait le faire dans l'ordre inverse. On détaille plus les questions d'ordre d'évaluation en 3.3.

Si on veut respecter l'ordre d'évaluation il faut soit disposer d'une instruction vérifiant l'ordre strict, soit faire quelque chose de plus compliqué, par exemple disposer d'un emplacement mémoire pour ranger la valeur de `e1` pendant qu'un calcule `e2`. Quelque chose comme :

```
⟦e1⟧
PUSH(0)                                  ; une adresse libre
WRITE                                    ; on y met la valeur de e1
⟦e2⟧
PUSH(0) ; on récupère la valeur de e1
READ                                     ; et on la met en haut de la pile
LEQ
NOT
```

◇

Solution de l'exercice 2.3.2 Tout simplement le même code que pour `0 - e` :

PUSH(0) ⟦e⟧ SUB.

Solution de l'exercice 2.4.1

(α)⟦p⟧ ⟦e⟧ NOT GTZ(α).

On peut remarquer que dans chaque itération de la boucle, on ne passe que par un saut GTZ, et non pas un GTZ puis un GTO comme dans le cas de la boucle while habituelle. Cela permet dans certaines machines une exécution plus efficace [1]. C'est pourquoi certains compilateurs préfèrent compiler l'instruction `while (e) p;` comme `if (e) do p while (e);` en utilisant la construction ci-dessus. C'est un exemple classique d'optimisation de programme. ◇

Solution de l'exercice 2.4.2 Par exemple :

```
      ⟦e1⟧        On calcule la valeur de e1
      GTZ(α)      si elle vaut false
      PUSH(1)     on place true sur la pile
      GTO(β)      on a fini
(α)   ⟦e2⟧        On place la valeur de e2 sur la pile
(β)   ...         suite du programme
```

◇

Solution de l'exercice 2.5.1 La commande ADD va commencer par calculer la somme des valeurs se trouvant aux adresses SP − 1 et SP − 2 et ranger le résultat à l'adresse SP − 2, et enfin décrémenter SP de 1. Les autres opérations comme MIN, MULT ou DIV fonctionnant de manière similaire. ◇

Solution de l'exercice 3.2.1 La variable `n` dans le corps de la fonction `reset` ne fait référence à rien. Ce programme sera donc refusé par le compilateur.

Plus exactement, il sera refusé dès la phase de typage, car le type de `n` dans la définition de `reset` ne peut être déterminé. ◇

Solution de l'exercice 3.3.1 Cela ne change rien, puisque les arguments `x` et `y` sont calculés avant d'exécuter la fonction et donc de calculer `x + y` ou `y + x`. ◇

Solution de l'exercice 3.4.1 Dans `C1` la variable globale est initialisée à 3 puis sa valeur modifiée à chaque appel de `f()` ; il affiche donc `12`. Dans `C2` chaque appel à `f()` aboutit à mettre la valeur `6` à `n`. Le programme affiche donc `6`. ◇

Solution de l'exercice 3.5.1 Dans `f(f(m))` le `f` interne fait référence à la fonction prenant un `int` en argument ; appliqué à `4` cette fonction rend en résultat le nombre `5.0` de type `float` (elle effectue au passage une coercion vers `float`). Le `f` externe fait référence à la fonction prenant un `float` en argument. Le programme va donc afficher `10.0`. ◇

1. En particulier un meilleur *pipe-lining*, c'est-à-dire d'exécuter certaines instructions en avance ; ce point est plus important pour des processeurs matériels, et donc plus pour des compilateurs natifs, que pour des machines virtuelles comme Java.

Solution de l'exercice 3.5.2 Ce programme est bien compilé, et le nom de fonction `main` est surchargé. Lorsqu'on appelle le programme, c'est la fonction `main(String[] a)` qui est appelée. Le programme affiche donc `"bonne nuit"`. ◇

Solution de l'exercice 3.6.1 Une solutions générale, lorsqu'on utilise une expression `e` comme une instruction, est :

– d'utiliser le code ⟦`e`⟧ qui exécute l'expression et place sa valeur en haut de la pile,

– puis d'ajouter l'instruction POP qui va effacer cette valeur.

Toutefois, lorsque l'expression est un appel de fonction, comme `f(3)` on peut compiler cela de manière un peu plus efficace en gardant ⟦`e`⟧ et en omettant simplement l'instruction PRX qui recopie la valeur du résultat de la fonction en haut de la pile. On compile alors `f(3);` par : PUSH(3) ; GSB(adr(`f`)) ;POP. ◇

Solution de l'exercice 3.6.2 On peut compiler `x++` comme :

PUSH(adr(x))	
READ	on lit la valeur de `x`
FETCH(0)	on la duplique
PUSH(1)	
ADD	on ajoute 1
PUSH(adr(x))	
WRITE	on écrit la nouvelle valeur de `x`

A la fin on a bien l'ancienne valeur de `x` en haut de la pile.

Pour `++x` on peut faire :

PUSH(adr(x))	
READ	on lit la valeur de `x`
PUSH(1)	
ADD	on ajoute 1
FETCH(0)	on duplique la nouvelle valeur de `x`
PUSH(adr(x))	
WRITE	on écrit la nouvelle valeur de `x`

Il reste alors une copie de la nouvelle valeur de `x` en haut de la pile.

On peut remarquer que dans les deux cas, on limite l'utilisation d'instructions READ et WRITE, en utilisant simplement une instruction FETCH(0). ◇

Solution de l'exercice 4.4.1 C'est parce que Java demande d'évaluer les arguments de gauche à droite, donc du premier au dernier. Il est donc naturel d'empiler la valeur du premier argument en premier, donc en bas, et celle du dernier argument à la fin, donc en haut. ◇

Solution de l'exercice 4.7.1 Dans les deux cas, à chaque appel de la fonction, est empilé l'argument, le frame pointeur et l'adresse de retour. Dans les deux cas, il y a une série d'appels récursifs avec les arguments 4, 3, 2 puis 1. Dans la version `fact1`, dans le corps de la fonction, l'appel récursif `fact(i - 1)` est effectué avant d'empiler `i` pour la multiplication. Dans la version `fact2` l'argument `i` est empilé avant l'appel récursif.

La hauteur de pile est dans les deux cas atteinte lors du test `i<2` dans le dernier appel récursif, avec l'argument 1. On a donc une pile moins haute pour `fact1` que pour `fact2`. Dans le premier cas :

2	constante 2
1	copie de l'argument `i`
adr(`fact1`)	adresse de retour de l'appel `fact1(1)`
FP_4	frame pointer
1	argument de `fact1(1)`
adr(`fact1`)	adresse de retour de l'appel `fact1(2)`
FP_3	frame pointer
2	argument de `fact1(2)`
adr(`fact1`)	adresse de retour de l'appel `fact1(3)`
FP_2	frame pointer
3	argument de `fact1(3)`
adr(c)	adresse de retour de l'appel `fact1(4)`
FP_1	frame pointer
4	argument de `fact1(4)`

et dans le second :

2	constante 2
1	copie de l'argument `i`
adr(`fact2`)	adresse de retour de l'appel `fact1(1)`
FP_4	frame pointer
1	argument de `fact1(1)`
2	argument de `fact1(2)` empilé pour la multiplication
adr(`fact2`)	adresse de retour de l'appel `fact1(2)`
FP_3	frame pointer
2	argument de `fact1(2)`
3	argument de `fact1(3)` empilé pour la multiplication
adr(`fact2`)	adresse de retour de l'appel `fact1(3)`
FP_2	frame pointer
3	argument de `fact1(3)`
4	argument de `fact1(4)` empilé pour la multiplication
adr(c)	adresse de retour de l'appel `fact1(4)`
FP_1	frame pointer
4	argument de `fact1(4)`

◇

Solution de l'exercice 5.9.1 Une version itérative :

```
static int somme(Liste l) {
  int r = 0;
  while (l != null) {
      r += l.head;    // ou r = r + l.head;
      l = l.tail;
  }
  return(r);
}
```

Une version récursive :

```
static int somme(Liste l) {
  if (l == null) return(0);
  return( l.head + somme(l.tail));
}
```

◇

Solution de l'exercice 5.9.2 Si la liste ne contient pas de cycle, le champ `tail` de son dernier élément est `null` et le programme termine lorsque le lièvre atteint cet élément.

Si la liste contient un cycle, il y a deux phases :

- Tant que la tortue n'a pas atteint le cycle, elle se rapproche du cycle d'une case à chaque tour.
- Une fois que la tortue a atteint le cycle, le lièvre est également dans le cycle. A chaque tour il se rapproche de la tortue d'une case. Il finit donc par la rattrapper.

L'algorithme termine donc toujours en un temps proportionnel au nombre d'éléments de la liste.

Par ailleurs, le programme rend bien `false` si et seulement si la liste contient `null`, c'est-à-dire qu'il n'y a pas de cycle. Comme il termine, il rend `true` lorsqu'il y a un cycle. ◇

Solution de l'exercice 5.9.3 Une solution possible :

```
boolean cycleP(Liste l) {
  if (l == null) return(false);
  Liste turtle = l;
  Liste hare = l.tail;
  while (turtle != hare) {
    if (hare == null || hare.tail == null)
    return(false);
    turtle = turtle.tail;
    hare = hare.tail.tail;
  }
  return(true);
}
```

Solution de l'exercice 5.10.1 Par exemple :

```
new ArbreBin(1,
                   new ArbreBin(3, new ArbreBin(9, null, null), null),
                   new ArbreBin(5, null, null))
```

Solution de l'exercice 5.10.2 Par exemple :

```
static boolean isHeap(ArbreBin t) {
  if (t == null) return true;
  if (t.left != null && t.value > t.left.value) return false;
  if (t.right != null && t.value > t.right.value) return false;
  return (isHeap(t.left) && isHeap(t.right));
 }
```

Solution de l'exercice 5.11.1 On utilise une définition semblable aux classes `Liste<E>` ou `ArbreBin`, mais chaque nœud contient la *liste* de ses sous-arbres :

```
class Tree<E> {
  E value;
  Liste<Tree<E>> subTrees;
  Tree(int i, Liste<Tree<E>> l) {
        value = i;  subTrees = l;
  }
}
```

Solution de l'exercice 5.12.1 C'est l'utilisation de `null` pour fabriquer une liste vide qui est erronée. Ce `null` sera considéré comme de type `Liste<String>` et l'appel récursif de `length()` sur cet objet causera une `NullPointerException`. Pour construire la liste vide correctement, il faut utiliser le constructeur `Liste<String>()`. Le code suivant fonctionne :

```
public static void main(String[] a) {
    Liste<String> l =
           new Liste<String>("a",
                             new Liste<String>("b",
                                               new Liste<String>()));
    System.out.println(l.length());
    }
}
```

Solution de l'exercice 5.13.1 En fait c'est presque plus simple. On garde une copie de l'adresse dans la pile et on la recopie avant chaque écriture.

```
PUSH(3)
ALLOC      ; on alloue un bloc de 3 mots
PUSH(7)    ; la première valeur qu'on va écrire
FETCH(1)   ; on recopie l'adresse
PUSH(0)    ; à l'offset 0
CWRITE
PUSH(8)    ; on recommence pour la deuxième valeur
FETCH(1)   ; à la même adresse
PUSH(1)    ; cette fois à l'offset 1
CWRITE
PUSH(9)    ; la troisième valeur qu'on va écrire
FETCH(1)
PUSH(2)    ; à l'offset 2
CWRITE
           ; rien de plus à faire, il reste une copie de l'adresse en haut
```

◇

Solution de l'exercice 5.13.2 On a omis de vérifier que l'indice est bien dans les bornes du tableau (positif et inférieur à la taille du tableau). Remarquez que cela n'est pas fondamentalement difficile, mais serait fastidieux à écrire ici. ◇

Solution de l'exercice 5.13.3 Tout simplement : ⟦t⟧; PUSH(0); CREAD. Voire en utilisant l'instruction READ : ⟦t⟧; READ. ◇

Solution de l'exercice 5.13.4 Ici aussi on va omettre la vérification que l'indice `e1` est bien dans les bornes.

On va placer sur la pile, dans l'ordre la valeur qu'on écrit, l'adresse du tableau, puis l'indice (qu'il faudra incrémenter de un pour avoir l'offset) :

```
⟦e2⟧
⟦t⟧
⟦e1⟧
PUSH(1)
ADD
CWRITE
```

Remarquez que ce code laisse bien la pile dans l'état initial. ◇

Solution de l'exercice 6.8.1 Dans tous les cas, la méthode `finalize()` doit être redéfinie pour Arthur. Il y a deux possibilités.

Soit on redéfinit `finalize()` pour tous les élèves, mais de telle manière à ce qu'il ne se passe quelque chose que pour Arthur. Par exemple :

```
class Eleve  {
   String nom;
   int age;
   boolean turbulent;
   String messageAdieu;
   Eleve(String n, int a) { nom = n; age = a; turbulent = false;
             messageAdieu =""; }
   Eleve(String n, int a, String message)
    { nom = n; age = a; turbulent = true;
      messageAdieu = message;}
  @Override
  public void finalize(){
    if (turbulent){System.out.println(messageAdieu);}
    return;
  }
}
```

Dans ce cas, il faudra construire Arthur, et uniquement lui, avec le second constructeur de la classe `Eleve`.

Une autre possibilité est de définir une classe spéciale pour Arthur, ou plutôt pour tous les élèves turbulents :

```
class EleveTurbulent extends Eleve {
     String messageAdieu;
     Eleve(String n, int a, String message) {
        super(n, a);
        messageAdieu = message;
       }
  #Override
  public void finalize(){
    System.out.println(messageAdieu);
    return;
  }
}
```

Le défaut de la première solution est de demander de changer le code de la classe `Eleve` même pour les élèves sages. Le défaut de la seconde solution est que l'on ne peut plus définir de sous-classe de élève (ou en tout cas qu'Arthur ne peut plus appartenir à une autre sous-classe).◇

Solution de l'exercice 6.8.2 Il faut se souvenir que la définition de la méthode `equals` dans la classe prédéfinie `Object` est `public boolean equals (Object o)`. Lorsque l'on souhaite *redéfinir* (*override*) cette méthode dans une sous-classe, il faut donc définir une méthode qui a *exactement* la même signature. On peut changer le nom des arguments si on le souhaite, *mais pas leur type*. L'argument de la méthode `equals` doit donc toujours avoir le type `Object`.

Le code proposé dans l'énoncé de l'exercice présente un piège dans lequel on tombe malheureusement facilement. Ce code est accepté par le compilateur Java, qui ne comprend pas que nous souhaitons redéfinir la méthode `equals(Object)`. Le compilateur croit que nous souhaitons définir une nouvelle méthode `equals(Point)`. N'oublions pas que Java autorise la *surcharge* (*overloading*) et que deux méthodes qui ont le même nom, mais des arguments de types différents, sont considérées comme deux méthodes différentes. Ainsi donc, ce code est accepté, mais ne se comporte pas de façon correcte. Si les variables `o1` et `o2` sont de type `Object`, et si on appelle `o1.equals(o2)`, alors la méthode effectivement appelée sera celle de la classe `Object`, qui effectue un test d'égalité physique, et ce même si les objets `o1` et `o2` sont en fait des instances de la classe `Point`.

Le code correct, proposé avant l'énoncé de l'exercice, utilise l'annotation `@Override` pour annoncer notre intention de redéfinir une méthode existante. Si on prend soin de toujours utiliser cette annotation, alors on ne pourra plus tomber dans le piège ci-dessus. Le compilateur nous indiquera en effet qu'il n'existe pas de méthode `equals(Point)` dans la classe `Object`. Nous comprendrons alors (espérons-le) que l'argument `o` de la méthode `equals` doit avoir le type `Object`.

Ceci soulève un problème. Si `o` a le type `Object`, comment peut-on comparer l'objet **`this`**, qui a le type `Point`, avec l'objet `o` ? On ne peut pas écrire **`this`**`.x == o.x &&` **`this`**`.y == o.y`, car un objet de type `Object` n'a pas (a priori) de champs `x` et `y`. Nous devons donc d'abord déterminer si `o` est une instance de la classe `Point`. Si oui, alors nous pourrons accéder aux champs `x` et `y` de `o`, et effectuer un test d'égalité structurelle entre points. Si non, alors nous considérerons que **`this`** et `o` sont différents, puisqu'ils ne n'ont pas la même classe principale.

Pour cela, nous utilisons un *downcast*. L'instruction `Point that = (Point) o;` vérifie si l'objet `o` est une instance de la classe `Point`. Si oui, cette instruction réussit, et nous avons alors accès à ce même objet sous un nouveau nom, `that`. La variable `that` admet le type `Point`, ce qui va nous permettre d'accéder aux champs `that.x` et `that.y`. Si non, cette instruction échoue, et lance une exception de classe `ClassCastException`.

Ici, nous rattrapons cette exception et renvoyons **`false`**, pour indiquer que **`this`** et `o` sont, dans ce cas, différents. Nous pourrions également choisir de ne pas rattraper cette exception, donc de la laisser s'échapper. Cela reviendrait à considérer cette situation comme le résultat d'une erreur de programmation. En effet, dans certaines applications, on n'a aucune raison de vouloir comparer deux objets de classes différentes, et mieux vaut dans ce cas signaler une erreur si jamais cette situation se présentait. ◇

Solution de l'exercice 6.8.3 Si l'on dote la classe `Point` d'une notion d'égalité structurelle, qui considère deux objets comme égaux lorsqu'ils ont le même contenu, alors mieux vaut que ce contenu ne soit pas modifiable. Dans le cas contraire, on risquerait de rencontrer une situation où `p.equals(q)` renvoie **`true`** à un certain moment, puis plus tard, après une modification du contenu de `p` ou `q`, ce même appel renvoie **`false`**. Ce serait perturbant. Il semble plus raisonnable de rendre les champs `x` et `y` immuables, de sorte que l'appel `p.equals(q)` renvoie toujours le même résultat.

En règle générale, on préferera donc doter les objets *modifiables* d'une notion d'égalité *physique*, et doter les objets *immuables* d'une notion d'égalité *structurelle*. Ce n'est pas une règle absolue, mais c'est la pratique la plus naturelle et la plus courante. ◇

Solution de l'exercice 6.9.1 On commence par empiler 33. Puis on recopie l'argument `this` en haut de la pile puis on appelle la méthode `d()` de la super-classe, dont on connait statiquement l'adresse.

```
PUSH(33)
RFR(-1)              ; recopie de this
GSB(adr(Point,d))    ; appel de super.d()
POP                  ; on efface la copie de this
PRX                  ; on pousse le résultat de super.d() sur la pile
ADD                  ; on calcule le résultat
PXR                  ; qui est poussé dans le registre R
RET                  ; c'est fini
```

◇

Solution de l'exercice 7.5.1 Le programme est correctement compilé. Les deux premières exceptions sont rattrapées et affichent donc `A` puis `B`. En revanche `E1` n'est pas une sous-classe de `E2` et donc la troisième exception n'est rattrapée que par le niveau supérieur de **`catch`**. Au final le programme affiche donc :

```
A
B
D
```

◇

Solution de l'exercice 7.9.1 Ce programme va afficher 6. En effet, chaque appel récursif de `f` va créer (en fait empiler) un traitement de `E`. A chaque fois, la valeur associée à l'exception est incrémentée avant de renvoyer l'exception au niveau supérieur. ◇

Solution de l'exercice 8.4.1 En tenant compte des précédences syntaxiques, cette expression correspond à `2 + (3 * x)`. On peut la construire, par exemple, par

```
new BinOp(PLUS, new IntExpression(2),
                new BinOp(MULT, new IntExpression(2),
                                VarExpression("x")))
```

◇

Solution de l'exercice 8.4.2 Rappelons que la méthode `toString()` existe pour toute classe et tout objet. On doit donc la redéfinir. On omet ci-dessous les constructeurs qui sont déjà donné dans la figure 8.2.

```
class IntExpression extends Expression {
  int value;
  public String toString() { return(value); }
  ...
}
class VarExpression extends Expression {
  String name;
  public String toString() { return(name); }
  ...
}
class UniOpExpression extends Expression {
  public UniOp      uniop;
  public Expression term;
  public String toString() {
    if (uniop == MINUS)
       return ("(- " + term.toString() + ")" );
    else
    return ("(? " + term.toString() + ")" );
  }
  ...
}
class BinOpExpression extends Expression {
  public BinOp      binop;
  public Expression left;
  public Expression right;

  public String toString() {
    return( "(" + left.toString() + binop.printString() +
    right.toString() + ")" );
  }
  ...
}
```

Solution de l'exercice 8.8.1 On a intérêt à effectuer cette optimisation, car le code optimisé donnera exactement le même résultat, en effectuant un saut GTO de moins ; il sera donc plus rapide.

Un exemple de code source pour lequel cette optimisation pourra être effectuée est :

```
if (e1)
  if (e2) p1 else p2;
  else p3;
```

Solution de l'exercice 8.8.2 L'expression est compilée par la suite de deux instructions : PUSH(a); READ où a est l'adresse utilisée pour stocker la valeur de `c`. Comme cette valeur ne change pas au cours de l'exécution, on peut remplacer la séquence par PUSH(7125), ce qui économise une instruction et un accès à la mémoire. On voit que c'est uniquement parce que la variable est `final` qu'on peut effectuer cette optimisation. ◇

Solution de l'exercice 8.8.3 Une possibilité :

```
     ⟦e⟧
     FETCH(0)        ; on recopie la valeur de e pour faire le test d'égalité
     PUSH0
     EQ              ; ; on fait le test
     GTZ(α)          ; si false on effectue l'appel
     POP             ; si true on met le résultat 1 en haut de la pile...
     PUSH1
     GTO(β)          ; ... et on va à la suite du code
(α)  GSB(adr(fact))
     POP
     PRX
(β)  ...
```

Remarquez qu'il faut également traiter ainsi l'appel récursif de la définition de la factorielle. Son code devient alors :

```
adr(fact)  RFR(−1)         ; on pousse n sur la pile
           RFR(−1)         ; on pousse n - 1 sur la pile
           PUSH(1)
           SUB
           FETCH(0)
           PUSH(0)
           EQ
           GTZ(α)
           POP
           PUSH(1)
           GTO(β)
(α)        GSB(adr(fact))
           POP
           PRX
(β)        MUL
           PXR
           RET
```

Solution de l'exercice 10.7.1 Il y a plusieurs formulations possibles ; en voici une :

1. On débute avec des cloisons partout, c'est-à-dire que chaque case n'est en relation qu'avec elle-même.
2. On considère les cloisons dans un ordre aléatoire (on peut pour cela commencer par choisir une permutation aléatoire ; voir (Fisher and Yates, 2017)).
3. Si les cases de part et d'autre de la cloison ne sont pas reliées, on enlève la cloison, sinon on la laisse.

4. On itère l'étape précédente jusqu'à ce que toutes les cases soient dans la même relation d'équivalence.

On remarque que chaque fois qu'on enlève une cloison, on diminue le nombre de classes d'équivalence de un. On sait donc qu'on doit enlever $n \times m - 1$ cloisons.

Le fait qu'on enlève uniquement des cloisons entre cases non-reliées garantit qu'il n'y a jamais de cycle, donc qu'il n'y a toujours qu'un seul chemin entre deux cases de la même relation d'équivalence.

Il s'agit d'une variante de l'algorithme de Kruskal pour trouver un arbre couvrant maximal dans un graphe (voir par exemple (Cormen et al., 2009) ou (Sedgewick and Wayne, 2011)). ◇

Solution de l'exercice 11.3.1 Là encore, les comportements asymptotiques des deux implémentations sont similaires et la quantité de mémoire utilisée est en $O(N)$ (à condition bien sûr qu'on ait une borne sur la taille mémoire des chaînes de caractères empilées). A noter qu'après chaque redimensionnement de tableau, le tableau précédent peut être effacé par le GC.

En revanche, comme pour la vitesses, il y a un facteur constant favorable à l'implémentation par tableaux. En effet avec un tableau, il faut allouer n mots dans le tas pour pointer vers n chaînes de caractères. Dans le cas des listes chaînées, pour chaque élément il faut créer un objet de classe `StackContent` contenant le pointeur vers la chaîne mais aussi (au moins) un pointeur vers l'élément suivant. ◇

Bibliographie

Aho, A., Lam, M., Sethi, R., and Ullman, J. (2007). *Compilateurs : principes, techniques et outils : Avec plus de 200 exercices*. Pearson. https://books.google.fr/books?id=iN9TNwAACAAJ.

Arnold, D. N. (2000). The patriot missile failure. http://www-users.math.umn.edu/~arnold/disasters/patriot.html.

Barendregt, H. (1984). *The lambda calculus : its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland.

Bournez, O. (2017). Fondements de l'informatique : Logique, modèles, calculs. INF412.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms (Third Edition)*. MIT Press.

Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. (2008). *Algorithms*. McGraw-Hill.

Dea, C., Heckler, M., Grunwald, G., Pereda, J., and Phillips, S. (2014). *JavaFX 8 : Introduction by Example*. Apress, Berkely, CA, USA, 2nd edition.

Dijkstra, E. (1979). Classics in software engineering. chapter Go to Statement Considered Harmful, pages 27–33. Yourdon Press, Upper Saddle River, NJ, USA.

Dowek, G. (2008). *Les principes des langages de programmation*. Informatique (Palaiseau). les Éd. de l'École polytechnique. https://books.google.fr/books?id=8m3UMIUVM1IC.

Dowek, G. (2010). *Les démonstrations et les algorithmes*. Editions de l'Ecole polytechnique.

Dowek, G. (2015). *La Logique*. Le Pommier.

Doxiadis, A., Papadimitriou, C., Papadatos, A., and Di Donna, A. (2009). *Logicomix : An Epic Search for Truth*. Bloomsbury USA.

Eco, U. (1994). *La recherche de la langue parfaite dans la culture européenne*. Faire l'Europe. Éditions du Seuil. https://books.google.fr/books?id=lbwcAQAAIAAJ.

Fisher and Yates (2017). The Fisher-Yates shuffle. http://en.wikipedia.org/wiki/Fisher-Yates_shuffle.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education.

Hofstadter, D. R. (1979). *Godel, Escher, Bach : An Eternal Golden Braid*. Basic Books, Inc., New York, NY, USA.

Hunt, J. (1997). *Smalltalk and Object Orientation : An Introduction*. Springer.

Kernighan, B. W. and Ritchie, D. (1988). *The C Programming Language, Second Edition*. Prentice-Hall.

Kleinberg, J. M. and Tardos, É. (2006). *Algorithm design*. Addison-Wesley.

Leroy, X. (2010). Comment faire confiance à un compilateur ? *La Recherche*, 440. http://gallium.inria.fr/~xleroy/publi/cahiers-inria-2010.pdf.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. (2017). *The OCaml system, release 4.06, Documentation and user's manual*. INRIA. http://caml.inria.fr/pub/docs/manual-ocaml/.

Levy, S. (2010). *Hackers : Heroes of the Computer Revolution - 25th Anniversary Edition*. O'Reilly Media. https://books.google.fr/books?id=mShXzzKtpmEC.

Mathiassen, L., Munk-Madsen, A., Nielsen, P. A., and Stage, J. (2000). *Object Oriented Analysis and Design*. Marko Publishing.

McDonald, J. (2007). Design patterns. https://dzone.com/refcardz/design-patterns.

Mimram, S. (2020). *PROGRAM = PROOF*. Independently published.

Muller, J.-M., Brisebarre, N., de Dinechin, F., Jeannerod, C.-P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., and Torres, S. (2010). *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston. ACM G.1.0 ; G.1.2 ; G.4 ; B.2.0 ; B.2.4 ; F.2.1., ISBN 978-0-8176-4704-9.

Naftalin, M. and Wadler, P. (2006). *Java generics and collections*. O'Reilly. http://shop.oreilly.com/product/9780596527754.do.

Oracle (2012a). Creating a GUI with JFC/Swing. http://docs.oracle.com/javase/tutorial/uiswing/.

Oracle (2012b). Java SE 7 API specification. http://docs.oracle.com/javase/7/docs/api/index.html.

Pottier, F. and Werner, B. (2014). *Algorithmique & Programmation, INF431*. Ecole Polytechnique. http://www.enseignement.polytechnique.fr/informatique/INF431/X12-2013-2014/.

Sedgewick, R. and Wayne, K. (2011). *Algorithms (fourth edition)*. Addison-Wesley.

Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley Professional, 4th edition.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160.

Index