Programmation Avancée
(INF441)

François Pottier

8 mars 2016

Table des matières

Introduction

Ce cours s'adresse aux élèves de deuxième année de l'École Polytechnique, c'est-à-dire à des étudiants ayant acquis une certaine familiarité d'une part avec la programmation (en particulier en Java) et d'autre part avec quelques algorithmes et structures de données élémentaires.

L'objectif de ce cours est d'approfondir la maîtrise de la **programmation**. Son propos n'est donc pas de présenter de nouveaux algorithmes ou structures de données. Il ne s'agit pas non plus de passer en revue les détails de tel ou tel langage de programmation, ni de donner un catalogue de « trucs et astuces » connus des vieux programmeurs. L'objectif est de mettre l'accent sur un petit nombre de **concepts fondamentaux**, communs à la plupart des langages de programmation. Parmi ces concepts, l'**abstraction** joue un rôle central, car elle seule permet de construire des logiciels complexes à partir de composants simples et réutilisables. Nous mettons en lumière les deux principales significations de ce mot, qui rappellent respectivement les quantifications existentielle et universelle familières aux mathématiciens. Nous étudions également les différentes formes concrètes que peut prendre l'abstraction dans un langage de programmation.

Afin de mieux souligner la façon dont un même concept existe sous des formes légèrement différentes dans plusieurs langages de programmation, nous utilisons Java et OCaml. Nous établissons, à chaque fois que cela est possible, des liens entre ces deux langages. Nous nous efforçons de n'employer qu'un fragment relativement réduit, essentiel, de ces langages. Nous supposons connues les variables (modifiables en Java, immuables en OCaml), les structures de contrôle élémentaires (`if`, `switch`, `while` en Java ; `if`, `match`, `while` en OCaml), et les méthodes ou fonctions récursives. Le Chapitre 1 propose des remarques et exercices qui servent de rappel ou d'introduction à ces notions, surtout en ce qui concerne OCaml.

S'agit-il d'un cours de programmation « avancée », comme le suggère immodestement son titre ? Le lecteur ou l'élève en sera juge. Ce cours est constitué de huit séances seulement, ce qui en limite bien sûr la portée. Disons simplement qu'il ne s'agit plus d'apprendre à écrire du code pour résoudre un problème fixé, mais d'apprendre à organiser son code pour le rendre évolutif, réutilisable dans de nombreuses situations, applicable donc à de nombreux problèmes.

Aperçu

Le Chapitre 1 rappelle comment les **données** et les **comportements** sont représentés, en Java et en OCaml, à l'aide de concepts tels que les structures de données algébriques, les objets, et les fonctions. Il souligne le fait que **l'organisation de la mémoire** (la pile, le tas) est la même pour ces deux langages. Il rappelle quel est le **coût** asymptotique (en temps et en espace) des opérations élémentaires, comme la création d'un objet ou l'appel d'une méthode. La séance 1 correspond à ce chapitre.

Le Chapitre 2 met en avant la notion d'**abstraction**, comprise comme une forme d'isolation volontaire entre composants logiciels. L'abstraction favorise la construction de logiciels de grande envergure à partir de composants réutilisables et qui peuvent évoluer indépendamment les uns des autres. Les séances 2 et 3 correspondent à ce chapitre.

Le Chapitre 3 met l'accent sur le fait que « abstraire » peut aussi signifier **paramétrer**. Un composant peut être paramétré par une valeur, par un type, voire par une collection de

types et de valeurs, c'est-à-dire par un autre composant. En ce sens, l'abstraction **multiplie les possibilités d'assemblage** entre composants, et facilite donc à nouveau la construction de logiciels modulaires. Les séances 4 et 5 correspondent à ce chapitre.

Le Chapitre 4 aborde le problème de l'**itération**. Comment écrire du code réutilisable pour **produire** ou pour **consommer** une suite d'éléments ? Cette question très concrète et en apparence très simple soulève en réalité la question fondamentale de la répartition du **contrôle** entre producteur et consommateur. Elle admet plusieurs familles de solutions, parmi lesquelles nous étudions les **réducteurs**, également connus sous le nom de « *fold* », fréquents en OCaml ; les **itérateurs modifiables**, également appelés « *iterators* », fréquents en Java ; et les **itérateurs immuables**, également connus sous le nom de **flots** ou « *streams* ». Les séances 6, 7 et 8 correspondent à ce chapitre.

Principales références bibliographiques

Ce polycopié a vocation à rester relativement court et proche du contenu des huit séances. Pour aller plus loin, je recommande chaudement la lecture de tout ou partie des deux ouvrages suivants :

1. « *Program development in Java : abstraction, specification, and object-oriented design* ».
   Barbara Liskov et John Guttag (2001).
   Ce livre est consacré à la construction modulaire de programmes. La notion d'abstraction y joue, comme ici, un rôle central. Les exemples sont écrits en Java, mais les concepts présentés sont indépendants de ce langage. Ce livre est beaucoup plus complet que nous ne pouvons l'être ici.

2. « *Apprendre à programmer avec OCaml : algorithmes et structures de données* ».
   Sylvain Conchon et Jean-Christophe Filliâtre (2014).
   Une première partie est consacrée à l'apprentissage d'OCaml à l'aide d'exemples variés. Les deux parties qui suivent présentent différentes structures de données et algorithmes. Le code y est toujours écrit dans un style abstrait, modulaire et concis, ce qui en fait une excellente école de l'abstraction.

À propos de ce document

Afin de clarifier le contenu de ce polycopié, des « remarques » isolées expliquent certaines idées qu'il n'est pas essentiel de connaître, mais qui peuvent néanmoins être intéressantes. De façon symétrique, des « détails » isolés décrivent certains aspects pratiques qui ne sont pas toujours dignes d'intérêt d'un point de vue fondamental, mais qui peuvent néanmoins être utiles !

Ce polycopié contient des exercices corrigés relativement nombreux. **Les exercices font partie du cours.** C'est souvent grâce à un exercice que l'on s'aperçoit que l'on n'avait pas vraiment compris... et que l'on progresse vers une meilleure compréhension du cours. La plupart sont des exercices de programmation. Aussi, il faut les traiter non pas sur papier, mais avec l'aide de la machine. **Vérifiez que votre code est accepté** par un compilateur Java ou OCaml, et **testez-le**.

Ce cours a eu lieu pour la première fois en 2014–2015. Vous avez donc entre les mains la seconde itération de ce polycopié. Elle contient certainement encore de nombreuses imperfections. Je vous remercie par avance de bien vouloir me signaler les erreurs que vous remarquerez et me faire part de vos remarques et suggestions d'amélioration.

François Pottier

Chapitre 1

Données et comportement

Un langage de programmation est dit « généraliste » ou « *general-purpose* » s'il a vocation à être utilisé dans de nombreux domaines d'application, et non pas seulement dans un domaine précis, pour lequel il serait spécialisé, et auquel il serait limité.

Les langages généralistes sont Turing-complets : d'une façon ou d'une autre, ils permettent d'exprimer tous les algorithmes concevables. De plus, ces langages s'efforcent d'offrir des facilités pour que ces algorithmes s'écrivent sous une forme concise, claire, et générale. Parmi les facilités communes à presque tous ces langages, on trouve (à quelques détails près) une même organisation des **données** en mémoire, et une même façon de décrire des **comportements**, via des méthodes ou des fonctions.

L'objet de ce premier chapitre est de **souligner cette uniformité conceptuelle**. Bien que des langages comme Java, OCaml, Scala, C#, Scheme, Python, ou JavaScript puissent sembler assez différents en apparence, ils s'appuient en réalité sur les mêmes concepts fondamentaux. Il est important de comprendre cela avant de pousser plus loin notre étude.

Dans ce chapitre, nous rappelons d'abord comment les **données** sont structurées dans le tas (§1.1). Ensuite, nous présentons un certain nombre d'entités (fonctions, objets, suspensions) qui **marient données et comportement** et offrent à l'utilisateur un **service** abstrait (§1.2). Enfin, nous décrivons brièvement le **modèle de coût** commun à Java et OCaml, c'est-à-dire le coût en temps et en espace de chaque opération élémentaire (§1.3).

Parmi les langages de programmation, on distingue ceux qui sont **typés statiquement**, comme Java, OCaml, Scala, C#, où le compilateur impose le respect d'une certaine discipline, et garantit ainsi l'absence de certaines erreurs ; et ceux qui sont **typés dynamiquement**, comme Scheme, Python, JavaScript, où cette discipline n'est pas imposée pendant la compilation, mais seulement pendant l'exécution du programme. (D'autres encore, comme C, ne sont pas typés au sens où j'emploie ce mot. Même si C impose une certaine discipline de types, il existe des erreurs que C ne détecte ni à la compilation, ni à l'exécution.) Dans les langages typés statiquement, les **types** offrent un vocabulaire pour décrire les données et les comportements. Aussi, dans ce premier chapitre, nous utilisons les types de Java et d'OCaml pour décrire les objets que nous construisons. Dans les langages typés dynamiquement, ce vocabulaire descriptif n'existe pas officiellement. Néanmoins, données, fonctions, objets sont organisés de la même manière. Les concepts fondamentaux que nous présentons dans ce chapitre valent donc pour tous les langages typés cités plus haut, qu'ils soient typés statiquement ou dynamiquement.

1.1 Données

Les langages de programmation qui nous intéressent proposent un mécanisme d'**allocation dynamique de mémoire**. Cela signifie qu'il existe une zone de la mémoire, nommée **tas** ou « *heap* », dans laquelle il est possible à tout moment (dans la limite de l'espace disponible) de

créer de nouveaux **blocs de mémoire**, ou simplement « blocs ». Un bloc de mémoire est parfois également appelé « objet », mais nous éviterons cette terminologie, car le mot « objet » a un sens plus riche dans le cadre de la programmation orientée objets (§1.2).

La structure d'un bloc est simple : un bloc est en général composé d'une **étiquette** et d'un certain nombre de **champs**. L'étiquette indique quelle est la nature de ce bloc, et détermine le nombre et la nature des champs. Chaque champ contient une **valeur**, c'est-à-dire soit une **valeur primitive** (un nombre entier, un nombre à virgule flottante, un caractère, etc.), soit un **pointeur** vers un autre bloc.

1.1.1 Blocs, étiquettes et champs en Java

À titre d'exemple, voici un fragment de code Java extrait du polycopié INF411 (Filliâtre, 2014). Pour les besoins de l'algorithme de Huffman, on souhaite construire un arbre binaire. Plus précisément, on souhaite allouer (dans le tas) des blocs, et relier ces blocs les uns aux autres par des pointeurs, de façon à former un arbre binaire. La forme de ces blocs peut être décrite ainsi en Java :

```
abstract class HuffmanTree {
  int freq;
  // constructeur et méthodes omis
}
class Leaf extends HuffmanTree {
  final char c;
  // idem
}
class Node extends HuffmanTree {
  HuffmanTree left, right;
  // idem
}
```

Cette manière de représenter les arbres en Java est parfois appelée **composite pattern**.

Pour Java, l'étiquette d'un bloc alloué dans le tas est simplement sa classe. On peut donc créer des blocs étiquetés `Leaf`, à l'aide de l'expression `new Leaf (...)`, ou bien des blocs étiquetés `Node`, à l'aide de l'expression `new Node (...)`. La classe `HuffmanTree` est abstraite : il est interdit d'écrire `new HuffmanTree (...)`. Un bloc `x` de type `HuffmanTree` a donc pour étiquette **soit** `Leaf`, **soit** `Node`.

Notons que, lorsqu'on écrit « le bloc `x` », on veut dire en réalité « le bloc situé dans le tas à l'adresse `x` », voire, de façon encore plus explicite, « le bloc situé dans le tas à l'adresse qui est contenue dans la variable `x` ».

Un bloc étiqueté `Leaf` a deux champs, nommés respectivement `freq` et `c`. Tous deux contiennent des valeurs primitives, à savoir respectivement un entier et un caractère. Un bloc étiqueté `Node` a trois champs, nommés respectivement `freq`, `left` et `right`. Le premier contient un entier, tandis que les deux suivants contiennent des pointeurs vers d'autres blocs, censés représenter les sous-arbres gauche et droit. On voit que **le nombre et la nature des champs d'un bloc dépendent de son étiquette**.

Lorsqu'on dispose d'un bloc `x` de type `HuffmanTree`, on ne sait pas a priori si c'est une feuille ou un nœud, donc s'il a un champ `c` ou bien des champs `left` et `right`. (On sait toutefois, dans les deux cas, qu'il a un champ `freq`.) Il faut donc **analyser l'étiquette avant de pouvoir accéder aux champs**. En Java, cette analyse se fait typiquement via un appel de méthode. Supposons qu'une méthode, par exemple `traverse`, est déclarée dans la classe abstraite `HuffmanTree` et définie dans les deux sous-classes `Leaf` et `Node`. Alors, un appel de la forme `x.traverse(...)` consulte l'étiquette du bloc `x` et choisit, parmi les deux définitions de la méthode `traverse`, celle qui est appropriée. (Ce mécanisme est connu sous le nom de « *dynamic dispatch* ».) La définition de la méthode `traverse` située dans la classe `Leaf` a accès au champ `c`, tandis que

celle située dans la classe `Node` a accès aux champs `left` et `right`. Ainsi, répétons-le, **l'analyse de l'étiquette permet l'accès aux champs**.

Remarque 1.1 (Cast et `instanceof`) L'analyse de l'étiquette d'un objet `x` peut se faire non seulement via un appel de méthode, comme nous l'avons illustré ci-dessus, mais aussi via un test de la forme `x instanceof Node` ou bien via un « *cast* » de la forme `(Node) x`.

L'expression `x instanceof Node` teste si l'étiquette du bloc `x` est `Node`. Si oui, elle renvoie `true`. Si non, elle renvoie `false`.

L'expression `(Node) x` teste elle aussi si l'étiquette du bloc `x` est `Node`. Si oui, elle renvoie `x`. Si non, elle lance une exception `ClassCastException`. Du point de vue du compilateur, si `x` a le type `HuffmanTree`, alors `(Node) x` a le type `Node`. Ce « cast » explicite, qui peut échouer, permet donc de convertir le type de `x` de `HuffmanTree` vers `Node`. On l'appelle parfois « *downcast* » ou « conversion vers le bas », parce que `Node` est sous-classe de `HuffmanTree`. Rappelons, à titre de comparaison, que « *l'upcast* » ou « conversion vers le haut », par exemple de `Node` vers `HuffmanTree`, est toujours permise : tout objet de type `Node` peut être considéré comme un objet de type `HuffmanTree`. Cette conversion est implicite et n'échoue jamais. ◇

1.1.2 Blocs, étiquettes et champs en OCaml

Si l'on transcrit cet exemple en OCaml, alors il s'écrit de façon différente en apparence, mais la structure des blocs alloués dans le tas est **exactement la même**. Voici la définition du type des arbres :

```
type tree =
| Leaf of int * char
| Node of int * tree * tree
```

Cette définition indique qu'un bloc de type `tree` porte soit l'étiquette `Leaf`, soit l'étiquette `Node`. Ces étiquettes sont appelées **constructeurs de données**, en anglais « *data constructors* », ou bien « constructeurs » tout court. Cette définition indique de plus que si l'étiquette est `Leaf`, alors le bloc a deux champs, dont les types sont respectivement `int` et `char` ; tandis que si l'étiquette est `Node`, alors le bloc a trois champs, dont les types sont respectivement `int`, `tree` et `tree`. La structure des blocs est donc bien la même que dans le cas de Java.

Pour créer un nouveau bloc, en Java, on écrit par exemple `new Leaf (f, c)`. En OCaml, l'expression correspondante est `Leaf (f, c)`. Cela explique pourquoi `Leaf` est appelé un constructeur : on l'utilise pour construire (c'est-à-dire allouer et initialiser) un nouveau bloc.

Comme en Java, en OCaml, il faut analyser l'étiquette avant de pouvoir accéder aux champs. En OCaml, cette analyse se fait obligatoirement via une expression `match ... with ...`. On peut la considérer comme une forme généralisée de l'instruction `switch` que l'on trouve en C et Java. Si l'on dispose d'un bloc `x` de type `tree`, elle permet d'examiner l'étiquette de `x` et d'adopter un comportement différent suivant que cette étiquette est `Leaf` ou `Node`. Par exemple, une fonction `traverse` peut être définie ainsi :

```
let rec traverse (path : string) (x : tree) : unit =
  match x with
  | Leaf (_, c) ->
      Hashtbl.add dictionary c path
  | Node (_, x0, x1) ->
      traverse (path ^ "0") x0;
      traverse (path ^ "1") x1
```

Cette expression `match x with ...` comporte deux branches, qui correspondent aux deux valeurs possibles de l'étiquette portée par le bloc `x`. Dans la première branche apparaît une nouvelle variable locale `c`, dans laquelle on va trouver la valeur du second champ du bloc `x`. De même, les variables `x0` et `x1` sont locales à la seconde branche, et on y trouve les valeurs des second et troisième champs du bloc `x`. Le premier champ du bloc `x` n'est pas utilisé ici ;

c'est pourquoi on a préféré écrire _ plutôt que d'introduire une variable locale `freq` que l'on n'utilisera pas. On retrouve, en OCaml, le fait que l'analyse de l'étiquette permet l'accès aux champs : en dehors de l'instruction **`match`**, il n'existe aucun moyen d'accéder aux champs d'un bloc `x` de type `tree`.

1.1.3 Sommes, produits et récursivité

Dans les deux langages, Java et OCaml, les blocs qui constituent un arbre sont décrits par un type que l'on peut qualifier de **somme de produits**. En effet, l'étiquette d'un bloc est **soit** `Leaf`, **soit** `Node`. Or, on parle traditionnellement de **somme** lorsque l'on a un **choix** entre plusieurs alternatives. Ensuite, dans la branche `Leaf`, on a un champ `freq` **et** un champ `c` ; dans la branche `Node`, on a un champ `freq` **et** un champ `left` **et** un champ `right`. Or, on parle traditionnellement de **produit** lorsque l'on a **simultanément** plusieurs composantes.

Dans les deux langages, Java et OCaml, ces blocs sont décrits par un **type récursif**. Ce phénomène est clairement visible en OCaml, où la définition du type `tree` fait référence au type `tree` lui-même : les deuxième et troisième composantes d'un arbre étiqueté `Node` sont elles-mêmes des arbres. Il est moins visible en Java, parce qu'on n'écrit pas explicitement le fait que « un bloc de type `HuffmanTree` est soit un bloc de classe `Leaf`, soit un bloc de classe `Node` ». Si on l'écrivait, on verrait que la définition de `HuffmanTree` fait référence à `Leaf` et `Node`, tandis que les définitions de `Leaf` et `Node` font référence à `HuffmanTree`.

Les trois concepts que nous venons de mettre en évidence – **type somme**, **type produit**, **type récursif** – sont fondamentaux. Ils permettent de définir tous les types de données usuels. Un mathématicien ou un informaticien théoricien, qui aime employer des notations plus concises que Java ou OCaml, définirait ainsi notre type des arbres :

$$T = (Int \times Char) + (Int \times T \times T)$$

Il s'agit bien d'une définition récursive, dont le membre droit est une somme de produits. On parle de **type algébrique** pour désigner un type construit ainsi à l'aide de sommes, de produits, et de récursivité.

Remarque 1.2 (GC) En Java et en OCaml, il n'existe aucun moyen pour le programmeur de demander la désallocation d'un bloc. Chaque bloc est libéré automatiquement, dès qu'il devient **inaccessible**, par un système appelé « glaneur de cellules » ou **garbage collector**. (Un objet est **accessible** si on peut y parvenir, à partir d'une variable, en suivant une série de pointeurs.) Cela offre confort et sûreté : on ne peut pas détruire, par erreur, un bloc dont on aura encore besoin. Toutefois, il peut être difficile de prédire à quel moment un bloc sera désalloué. Les Exercices 4.26 et 4.27 illustrent certaines des subtilités de ce mécanisme. ◇

Remarque 1.3 (Unité) En OCaml, le type `unit` est un type algébrique à une seule branche, défini par **`type`** `unit = ()`. Il existe donc une (et une seule) valeur de ce type, à savoir `()`. Le nom « *unit* » provient du fait que ce type est un produit de zéro champs, donc l'**unité** du produit. En langage mathématique, on le note parfois 1.

Ce type qui à première vue pourrait sembler dénué d'intérêt pratique joue en réalité un rôle très important. En effet, en OCaml, on considère que chaque fonction a exactement un argument et un résultat. Une fonction sans argument doit donc être simulée à l'aide d'une fonction à un argument de type `unit`. De même, une fonction sans résultat est simulée à l'aide d'une fonction à un résultat de type `unit`.

En Java, de façon analogue, on peut poser **`public class`** `Unit {}` et construire une valeur de ce type en écrivant **`new`** `Unit ()`. Le mot-clef **`void`** de Java, qui permet d'indiquer qu'une méthode renvoie zéro résultats, correspond en réalité au même concept. ◇

Remarque 1.4 (Pointeur nul et options) Java a un **pointeur nul**. Si une variable `x` a le type `HuffmanTree`, elle contient soit la valeur **`null`**, soit l'adresse dans le tas d'un bloc étiqueté `Leaf` ou `Node`. Bien que parfois utile pour des raisons d'efficacité, la présence du pointeur **`null`**

est source d'erreurs, puisqu'elle ajoute silencieusement une alternative que le programmeur risque d'oublier, ce qui conduit au lancement d'une exception `NullPointerException`. Nous éviterons autant que possible l'emploi du pointeur nul.

OCaml n'a pas de pointeur nul. Lorsqu'on souhaite décrire « soit rien du tout, soit une valeur de type A », on emploie un type somme à deux branches, de façon à représenter ces deux alternatives. En langage mathématique, ce type somme s'écrirait $1 + A$, où 1 est le type unité (Remarque 1.3). En OCaml, on emploie le type `'a option`, défini ainsi dans la bibliothèque : `type 'a option = None | Some of 'a`. Ainsi, un bloc de type `int option` porte soit l'étiquette `None`, soit l'étiquette `Some`, et dans le second cas, il a un champ de type `int`. Par exemple, `None` et `Some 42` sont des blocs de type `int option`. L'Exercice 3.8 donne un exemple d'utilisation du type `int option`. ◇

Remarque 1.5 (Champs modifiables et champs immuables) Une fois que l'on a alloué un bloc dans le tas, est-il permis de modifier le contenu de ses champs ? On peut, au choix, interdire ou autoriser cela. On dit qu'un champ est **modifiable** ou « *mutable* » s'il est permis de modifier son contenu. Il est **immuable** ou « *immutable* » dans le cas contraire. En Java, chaque champ est par défaut modifiable, mais on peut rendre un champ immuable en précédant sa déclaration du mot-clef `final`. En OCaml, la convention est inverse : chaque champ est par défaut immuable, mais (dans le cas d'un type enregistrement, voir le Détail 1.11) on peut rendre un champ modifiable en précédant sa déclaration du mot-clef `mutable`. L'Exercice 1.2 en donne un exemple. ◇

Remarque 1.6 (Variables locales) Une **variable locale** est soit un paramètre d'une fonction, soit une variable introduite dans le corps d'une fonction.

En Java, on écrit par exemple `int x` pour déclarer une variable `x` de type `int`. En OCaml, une variable locale est introduite par les constructions `fun x -> ...` ou `let x = ... in ...` ou encore `match ... with Leaf (x, _) -> ... | Node (x, _, _) -> ...`.

En Java, une variable locale est par défaut modifiable. On peut la rendre immuable en précédant sa déclaration du mot-clef `final`.

En OCaml, une variable locale est toujours immuable. Pour simuler une variable locale modifiable, on peut introduire une variable immuable `x` dans laquelle on trouve l'adresse d'une référence (Exercice 1.2), c'est-à-dire un bloc de mémoire doté d'un champ modifiable. On écrit `let x = ref (...) in ...` pour allouer une référence et introduire une variable immuable `x` qui représente l'adresse de cette référence. On écrit ensuite `!x` pour lire l'unique champ de cette référence et `x := ...` pour le modifier. ◇

Remarque 1.7 (Abréviations de types) En OCaml, un nouveau type peut être défini comme une **abréviation** pour un type existant. Si on pose `type alphabet = (char, int) Hashtbl.t`, par exemple, alors le type `alphabet` désigne le type d'une table de hachage dont les clefs sont des caractères et les valeurs des entiers. En Java, ce mécanisme n'existe pas : pour définir un nouveau type, il faut définir une nouvelle classe ou interface. On ne peut pas introduire une abréviation pour `HashMap<Character,Integer>`. ◇

Détail 1.8 (Tableaux) Le type des tableaux, qui s'écrit `E[]` en Java et `'a array` en OCaml, est **primitif** : il n'est pas défini par l'utilisateur, mais pré-existant. Nous avons affirmé plus haut que, en général, la longueur d'un bloc de mémoire (c'est-à-dire le nombre de ses champs) est déterminée par son étiquette. On peut dire que les tableaux obéissent à cette règle si l'on considère que la longueur d'un tableau fait partie de son « étiquette ». Java et OCaml fournissent des opérations primitives pour obtenir la longueur d'un tableau et pour accéder (en lecture ou en écriture) à ses champs. Contrairement aux blocs définis par l'utilisateur, les tableaux permettent l'accès aléatoire, ou « *random access* » : l'indice du champ auquel on souhaite accéder peut être le résultat d'un calcul. ◇

Détail 1.9 (Champ partagé) Dans une définition de type en Java, un champ peut être commun à toutes les branches. Il est alors possible d'y accéder sans examiner l'étiquette du bloc. C'est

le cas du champ `freq` de l'exemple précédent. Il est déclaré dans la classe mère, `HuffmanTree`, donc présent dans tous les blocs de ce type, indépendamment de leur étiquette, `Leaf` ou `Node`. Si on dispose d'un bloc `x` de type `HuffmanTree`, l'expression `x.freq` est valide : on peut accéder à ce champ sans examiner l'étiquette. En langage mathématique, on pourrait écrire $T = Int \times (Char + T \times T)$. En OCaml, de même, on pourrait poser `type tree = int * rest` et `type rest = Leaf of char | Node of tree * tree`. Ici, nous avons préféré répéter la définition du premier champ, de type `int`, dans les deux branches. Nous pouvons toutefois nous doter d'une fonction auxiliaire pour accéder à ce champ sans nous soucier de l'étiquette : `let freq t = match t with Leaf(f, _) | Node (f, _, _) -> f`. ◇

Détail 1.10 (Champs anonymes) En Java, chaque champ est nommé. En OCaml, comme on l'a vu ci-dessus, les champs d'un type somme ne sont pas nommés. Dans la définition du type `tree`, on lit que le constructeur `Leaf` a deux champs et que le constructeur `Node` a trois champs, mais on ne leur donne pas de nom. Ils sont identifiés par leur position : on parle par exemple du « premier champ » ou du « second champ » du constructeur `Leaf`.

Lorsqu'on écrit une expression `match`, on est libre de choisir les noms des variables locales qui apparaissent dans chaque branche. Par exemple, pour accéder au champ « fréquence » du bloc `x`, on peut écrire `match x with Leaf (f, c) -> f | Node (f, t0, t1) -> f` ou encore `match x with Leaf (freq, c) -> freq | Node (qerf, foo, bar) -> qerf`.

On peut toutefois nommer les champs d'un type enregistrement ; voir le Détail 1.11. ◇

Détail 1.11 (Enregistrements) Dans le cas où une définition de type est constituée d'une seule branche, et dans ce cas seulement, OCaml nous autorise à nommer les champs. On écrit par exemple :

```
type point = { x: int; y: int }
```

On dit alors que le type `point` est un **type enregistrement**, ou « *record type* ». On écrit par exemple `{ x = 0; y = 1 }` pour allouer un bloc de type `point`. On écrit `p.x` et `p.y` pour accéder aux champs du bloc `p`. D'un point de vue mathématique, le type `point` est un type produit ; on pourrait écrire $P = Int \times Int$.

En OCaml, on peut aussi écrire :

```
type point = Point of int * int
```

Le type `point` est alors un type somme à une seule branche. On doit écrire `Point (0, 1)` pour allouer un bloc de type `point`, et on doit utiliser `match p with Point (x, y) -> ...` ou bien `let Point (x, y) = p in ...` pour accéder aux champs du bloc `p`. La syntaxe n'est pas la même, mais la façon dont le bloc est représenté en mémoire est la même : un point a toujours une étiquette (sans intérêt, puisque c'est toujours la même) et deux champs. ◇

Détail 1.12 (Paires) Il est fréquent que l'on souhaite grouper deux valeurs v_1 et v_2 pour construire une **paire**, c'est-à-dire un bloc à deux champs, qui contiennent respectivement les valeurs v_1 et v_2. C'est utile, par exemple, lorsqu'une fonction doit renvoyer deux résultats.

Java n'offre aucune facilité particulière pour cela. Néanmoins, il est bien sûr possible de définir par soi-même une classe (paramétrée) `Pair<A,B>`. Voir par exemple la Figure 4.31, qui fait partie de la solution de l'Exercice 4.19.

De même, en OCaml, on pourrait définir par soi-même un type (paramétré) des paires en posant `type ('a, 'b) pair = Pair of 'a * 'b`. On écrirait `Pair (0, 1)` pour allouer une paire dont le type s'écrirait `(int, int) pair`. On écrirait `match p with Pair (x, y) -> ...` ou bien `let Pair (x, y) = p in ...` pour accéder aux composantes de la paire `p`.

Afin d'offrir une syntaxe plus concise, OCaml possède un type primitif des paires. On peut écrire simplement `(0, 1)` pour construire une paire, dont le type est noté `int * int`. On écrit `match p with (x, y) -> ...` ou bien `let (x, y) = p in ...` pour accéder aux composantes de la paire `p`. Encore une fois, ce sont là des détails syntaxiques, mais la façon dont une paire est représentée en mémoire est toujours la même : c'est un bloc doté d'une étiquette (sans intérêt) et de deux champs.

Plus généralement, pour chaque entier n, OCaml possède un type primitif des n-uplets, ou « *tuples* ». Par exemple, l'expression `("Paris", 75, "France")` alloue un bloc à trois champs, dont le type est `string * int * string`. D'autres langages de programmation suivent cet exemple : en Scala, par exemple, `("Paris", 75, "France")` est une façon concise d'écrire `new Tuple3 ("Paris", 75, "France")`, dont le type est `Tuple3[String, Int, String]`. ◇

Détail 1.13 (Types énumérés) En OCaml, lors de la déclaration d'un type algébrique, si un constructeur a zéro champs, alors on omet le mot-clef `of`. Ainsi, on écrit `type color = Red | Green | Blue` et non pas `type color = Red of | Green of | Blue of`. La définition du type `option` (Remarque 1.4) en donne un autre exemple.

Le type `color` est un exemple de **type énuméré**, c'est-à-dire un type somme dont chaque constructeur a zéro champs. En OCaml, ce concept n'est qu'un cas particulier du concept plus général de type algébrique. En Java, il existe un mot-clef `enum` pour déclarer un type énuméré.◇

Détail 1.14 (Syntaxe des constructeurs) En OCaml, lorsqu'on construit un bloc, les arguments du constructeur (c'est-à-dire les valeurs initiales des champs) sont entourés de parenthèses : on écrit `Leaf (f, c)` ou `Node (f, x0, x1)` ou `Some (42)`. Il y a deux exceptions à cette règle. Si le constructeur a zéro arguments, alors on **doit** omettre les parenthèses : on écrit `None` et non pas `None ()`. Si le constructeur a un argument, alors on **peut** omettre les parenthèses : on peut écrire `Some (42)` ou bien `Some 42`. ◇

Exercice 1.1 (Recommandé) En Java puis en OCaml, définissez un type de données pour représenter les entiers « étendus », c'est-à-dire les entiers relatifs auxquels on ajouté les deux éléments $-\infty$ et $+\infty$. (Vous utiliserez les entiers de la machine.) Définissez des moyens pour l'utilisateur de construire toutes les valeurs de type « entier relatif ». Définissez une opération pour convertir un entier relatif en une chaîne de caractères. Définissez une opération `leq` permettant de comparer (au sens large) deux entiers relatifs. Enfin, définissez une opération `max` sur les entiers relatifs. ◇ Solution page 71.

Exercice 1.2 (Recommandé) En OCaml, définissez un type `box` représentant un bloc doté d'un seul champ, modifiable, contenant un entier. (Le Détail 1.11 pourra être utile.) Écrivez trois fonctions `create`, `get`, `set` pour créer une boîte, c'est-à-dire un bloc de type `box` ; pour lire son contenu ; pour modifier son contenu. Généralisez ensuite votre code pour que le contenu de la boîte ne soit pas nécessairement une valeur entière, mais une valeur de type arbitraire `'a`. Les boîtes ainsi obtenues sont habituellement appelées **références**, et le type `'a box` est défini sous le nom `'a ref` dans la bibliothèque. ◇ Solution page 71.

Exercice 1.3 Comme dans l'Exercice 1.2, mais cette fois en Java, on souhaite définir des « boîtes », c'est-à-dire des blocs dotés d'un champ contenant une valeur de type `X`. De plus, on souhaite imposer sur chaque boîte le protocole suivant. Une nouvelle boîte est créée « vide », c'est-à-dire non initialisée : son champ ne contient pas de valeur de type `X`. Une méthode `set` permet d'initialiser ce champ ; elle est censée être appelée au plus une fois. Une méthode `get` permet de consulter le contenu de la boîte ; elle peut être appelée un nombre arbitraire de fois, mais après seulement que `set` a été appelée. Définissez une classe `Box<X>` représentant une boîte dont le contenu est de type `X`. Dotez-la d'un constructeur, d'une méthode `set` et d'une méthode `get`. Faites en sorte qu'une erreur ait lieu pendant l'exécution si le protocole décrit ci-dessus n'est pas respecté. ◇ Solution page 73.

Exercice 1.4 (Recommandé) Une liste est soit vide, soit constituée d'un élément (le premier élément de la liste) et d'une sous-liste (la liste des éléments suivants). En langage mathématique, si X est le type des éléments, alors le type L des listes est défini par l'équation $L = 1 + X \times L$. En Java puis en OCaml, transcrivez cette définition, dans le cas où les éléments sont des entiers. Définissez une opération pour convertir une liste en une chaîne de caractères. Définissez une opération pour déterminer si une liste contient ou non un élément pair. ◇ Solution page 74.

Solution page 76. **Exercice 1.5 (Recommandé)** On se donne en OCaml un type algébrique représentant des arbres binaires. Le constructeur `Leaf` représente une feuille ; il ne porte aucune information. Le constructeur `Node` représente un nœud interne ; il porte un élément de type `int` ainsi que deux sous-arbres.

```
type tree =
  | Leaf
  | Node of tree * int * tree
```

Écrivez une fonction récursive `elements`, de type `tree -> int list`, telle que `elements t` est la liste des éléments contenus dans l'arbre `t`, lue dans l'ordre infixe, de gauche à droite. Par exemple, si `t` est l'arbre `Node (Node (Leaf, 1, Leaf), 2, Node (Leaf, 3, Leaf))` alors la liste `elements t` doit être `[1; 2; 3]`. Vous pourrez utiliser la fonction `List.append`, également notée `@`, qui construit la concaténation de deux listes.

Quelle est la complexité asymptotique en temps de la fonction `elements` ?

Pour obtenir une meilleure complexité, écrivez une fonction `elements_append`, de type `tree -> int list -> int list`, telle que la liste `elements_append t tail` soit égale à la liste `(elements t) @ tail`. Quelle est la complexité asymptotique de `elements_append` ?

À l'aide de `elements_append`, définissez en une ligne une nouvelle version, plus efficace, de la fonction `elements`. ◇

Solution page 77. **Exercice 1.6** On souhaite programmer une calculatrice capable de manipuler des expressions symboliques à une variable, par exemple « 20 », « $20 + x$ », « $20 + 3 \times x + x \times x$ », etc. Une expression sera soit la variable « x », soit une constante, soit la somme de deux expressions, soit le produit de deux expressions. Cela se résume ainsi : $e ::= x \mid c \mid e + e \mid e \times e$, où la lettre c dénote une constante entière. Définissez en OCaml un type de données pour représenter ces expressions. Définissez une opération qui, étant donnée une expression e et étant donnée une valeur numérique pour la variable x, calcule la valeur numérique de l'expression e. Enfin, définissez une opération qui, étant donnée une expression e, calcule sa dérivée e' vis-à-vis de la variable x. (Notons que l'Exercice 4.8, en Java, concerne un sujet proche.) ◇

1.2 Comportement

Dans ce qui précède (§1.1), nous avons utilisé le vocabulaire des types algébriques – sommes, produits, récursivité – pour décrire des structures de données. Ce vocabulaire permet de décrire des listes, des arbres, et plus généralement toutes les structures de données construites à partir de blocs de mémoire et de pointeurs entre ces blocs.

Le vocabulaire des types algébriques, seul, ne permet de décrire que la **forme** des données, c'est-à-dire leur organisation en mémoire. Il ne permet pas de leur attribuer un **comportement**. Il permet, par exemple, de décrire la forme des arbres binaires de recherche ; mais, pour définir les opérations d'insertion ou de recherche dans un tel arbre, il faut écrire du code : des méthodes, en Java, ou bien des fonctions, en OCaml.

Dans le monde de la programmation « traditionnelle » ou **programmation procédurale**, données et comportement sont séparés. Si par exemple nous avons défini d'une part un type des listes et d'autre part une fonction qui convertit une liste en une chaîne de caractères (Exercice 1.4), alors nous avons affaire à deux entités bien distinctes. D'une part, une liste est représentée, dans la mémoire de la machine, par une série de blocs alloués dans le tas. D'autre part, notre fonction de conversion est représentée dans la mémoire de la machine sous forme d'une suite d'instructions, stockée en dehors du tas, dans une zone (non modifiable) de la mémoire, réservée au code. Pour appeler cette fonction, il suffit de connaître son adresse, c'est-à-dire l'adresse de sa première instruction. On parle parfois de « pointeur de code » pour désigner cette adresse.

La **programmation orientée objets**, dont Java est l'un des représentants, de même que la **programmation fonctionnelle**, dont OCaml est l'un des représentants, assouplissent cette

séparation totale entre données et comportement. Toutes deux permettent de **marier données et comportement** de façon plus subtile.

Dans le monde de la programmation orientée objets, on appelle **objet** une entité dotée d'un **état** et d'un **comportement**.

Le mot « état » désigne ici les données qui constituent cet objet, c'est-à-dire, au minimum, le bloc de mémoire sous-jacent à cet objet. L'état d'un objet peut éventuellement englober d'autres objets : par exemple, lorsque l'on parle abstraitement de « l'état » d'une table de hachage de type `HashMap<String,List<Integer>>`, on désigne tous les blocs qui participent à la représentation de cette table en mémoire, et non pas seulement le bloc de classe `HashMap` qui en constitue la racine. Cet état peut être modifiable ou non.

Le mot « comportement » désigne les méthodes de cet objet. On adopte souvent un point de vue anthropomorphe et l'on dit que l'objet « sait » effectuer certaines opérations : par exemple, un objet doté d'une méthode `toString` « sait » produire une description de lui-même sous forme d'une chaîne de caractères. Le code de la méthode `toString` n'est pas le même suivant la classe à laquelle appartient l'objet : ainsi, des objets appartenant à différentes classes ont différents comportements. Une méthode est parfois appelée **message** : on dit alors que l'objet « sait » répondre au message `toString`.

Un objet, vu comme un bloc de mémoire, contient une étiquette et des champs. Cette étiquette indique à quelle classe appartient l'objet, donc permet (indirectement, via quelques calculs) d'obtenir le pointeur de code de chacune des méthodes de cette classe. On constate donc que, dans ce bloc, sont stockés non seulement des données mais aussi un comportement (plus précisément, une étiquette, qui donne accès à des pointeurs de code).

Un objet marie donc état et comportement. Au-delà de la métaphore anthropomorphe, en quoi cela est-ce utile ou souhaitable ? Cela permet une forme d'**encapsulation**, ou d'**abstraction**. On entend par là le fait qu'on peut interdire tout accès direct depuis l'extérieur à l'état d'un objet. En Java, il suffit pour cela de déclarer que les champs de cet objet sont privés, ou `private`. Seules les méthodes de la classe à laquelle appartient l'objet ont alors accès à ces champs. Pour accéder depuis l'extérieur à l'état d'un objet, il est donc obligatoire de passer par l'intermédiaire d'un appel de méthode. Ceci permet au concepteur de la classe de contrôler ou de limiter la façon dont l'état de l'objet peut être consulté ou modifié. L'utilisateur sait alors quel est le **service** offert par cet objet, mais ne sait pas de quelle manière ce service est implémenté, donc ne peut ni en perturber le bon fonctionnement, ni le distinguer d'une autre implémentation de ce même service. Selon certains, c'est là l'essence de la programmation orientée objets (Aldrich, 2013).

Dans le monde de la programmation fonctionnelle, aussi, on trouve des entités qui associent état et comportement. On les appelle **fonctions** ou **clôtures**. Afin de mettre en évidence cette combinaison entre données et comportement sans toutefois entrer trop avant dans les détails pour le moment, supposons que l'on calcule ainsi la somme des éléments d'une liste d'entiers :

```
let sum (xs : int list) : int =
  let s = ref 0 in
  List.iter (fun x -> s := !s + x) xs;
  !s
```

On alloue dans le tas une référence `s`, dont la valeur initiale est `0`, puis on itère sur la liste `xs`. Pour cela, on appelle `List.iter`, en lui passant une fonction anonyme qui a accès à la référence `s`. Pour chaque élément `x` de la liste, `List.iter` appelle cette fonction anonyme, qui met alors à jour la somme partielle stockée dans `s`. Une fois l'itération terminée, `s` contient la somme de tous les éléments. (La fonction `List.iter` est étudiée plus en détail en §4.1.1.)

Quelle est la nature exacte de « l'objet » construit par l'expression `fun x -> s := !s + x` ? C'est, à première vue, « une fonction ». Comment est-il représenté en mémoire ? Il doit contenir un pointeur de code, c'est-à-dire l'adresse où est située la suite d'instructions correspondant à la fonction anonyme `fun x -> s := !s + x`. Cependant, ce pointeur de code ne suffit pas. Il faut que la fonction anonyme puisse accéder à la référence `s`, qui est allouée dans le tas. Or, l'adresse de `s` ne peut pas apparaître dans le code. Celui-ci est immuable et existe déjà, stocké

sur un disque, avant même que le programme soit exécuté ; la référence `s`, au contraire, n'existe pas au moment où le programme commence son exécution. Le bloc de mémoire qui représente la fonction anonyme doit donc contenir non seulement un pointeur de code, mais aussi une donnée, à savoir l'adresse de `s`. Ce bloc, appelé **clôture**, marie donc données et comportement.

Ceci suggère que **les fonctions d'OCaml sont très proches des objets de Java**. En effet, elles aussi associent données et comportement, et, pour cette raison, elles aussi permettent une forme d'encapsulation ou d'abstraction.

Dans la suite, nous rappelons d'abord ce que l'on appelle traditionnellement une **procédure** ou une **fonction** (§1.2.1). Puis, nous étudions de plus près les fonctions-**clôtures** d'OCaml, que nous expliquons en approfondissant l'analogie avec les objets de Java (§1.2.2). Nous présentons également les **clôtures de Java 8** (§1.2.3). Enfin, nous introduisons la notion de **suspension** (§1.2.4), qui combine une clôture et un mécanisme de mémoisation. Nous ne parlerons pas des classes et des objets d'OCaml, car la notion de fonction-clôture nous suffira amplement pour définir des abstractions utiles et intéressantes.

1.2.1 Procédures et fonctions traditionnelles

Les notions de méthode, au sens de Java, ou de **fonction**, au sens d'OCaml, descendent toutes deux de ce que l'on appelait aux débuts de l'informatique les « procédures » ou « sous-routines ». On lira avec amusement et intérêt les deux pages de l'article de Wheeler (1952), qui décrivent fort bien l'attrait de cette notion. Une fois que l'on a écrit une fonction pour réaliser une certaine tâche – par exemple, « étant donné un nombre x, calculer le nombre sin x » – tout se passe comme si on avait ajouté une nouvelle instruction élémentaire – ici, l'instruction « sin » – à une machine qui en était dépourvue. La **décomposition** d'un programme en fonctions permet d'en maîtriser la complexité : l'auteur de la fonction « sin » peut se concentrer sur l'écriture de la suite d'instructions (probablement complexe) qui constitue cette fonction, sans s'inquiéter du reste du programme, et peut **tester isolément** cette fonction – on parle en anglais de « *unit testing* ». Symétriquement, l'utilisateur de la fonction « sin » n'a pas à s'inquiéter de la manière dont le sinus est calculé. Il lui suffit de **savoir ce que fait la fonction, pas comment elle le fait**. C'est ce que Liskov et Guttag (2001) appellent **abstraction procédurale**.

Quels sont les points communs aux procédures, méthodes, ou fonctions que l'on trouve dans presque tous les langages de programmation, par exemple C, Java, OCaml ?

Un premier point commun est le mécanisme d'appel et de retour lui-même. L'exécution de l'**appelant** (la fonction qui effectue l'appel) est suspendue, pour donner la main à l'**appelé** (la fonction qui est appelée). Symétriquement, une fois que l'appelé a terminé son exécution, l'exécution de l'appelant reprend au point qui suit l'appel.

Un second point commun est le mécanisme de communication entre appelant et appelé. L'appelant transmet à l'appelé une valeur, appelée **argument** ; symétriquement, une fois son exécution terminée, l'appelé transmet à l'appelant une valeur, appelée **résultat**.

Un troisième point commun est la notion de **variable locale**. Chaque fonction peut créer autant de variables locales qu'elle le souhaite. Ces variables lui sont propres ; il n'y a pas d'interférence entre les variables locales de l'appelant et celles de l'appelé.

Ces trois points supposent un mécanisme d'allocation de mémoire. En effet, pour reprendre l'exécution de l'appelant au point qui suit l'appel, il est nécessaire de mémoriser l'**adresse de retour**, c'est-à-dire le point du programme où l'exécution de l'appelant doit reprendre. Cela demande d'allouer un nouvel emplacement en mémoire. De même, pour stocker les variables locales de l'appelé, de nouveaux emplacements en mémoire sont nécessaires. Enfin, l'argument transmis de l'appelant à l'appelé est placé dans une variable locale de l'appelé, qui demande elle aussi un emplacement en mémoire. Ces emplacements ne sont utilisés que pendant l'exécution de l'appelé : une fois celle-ci terminée, ils sont libérés. C'est pourquoi il est naturel d'allouer ces emplacements non pas dans le tas, qui est utilisé pour des blocs dont la durée de vie est a priori inconnue, mais dans la **pile**, une zone de mémoire utilisée pour allouer des emplacements dont la durée de vie coïncide avec l'exécution d'une fonction. La

pile est invisible pour le programmeur, puisqu'elle est entièrement gérée par le compilateur ; mais il faut garder à l'esprit le fait qu'un appel de fonction alloue une certaine quantité de mémoire sur la pile, qui est libérée lorsque la fonction termine. L'exécution du programme peut d'ailleurs être brutalement interrompue si la taille de la pile dépasse la limite fixée par le système d'exploitation. On observe alors une exception nommée `StackOverflowError` en Java et `Stack_overflow` en OCaml.

Un quatrième et dernier point commun est la **récursivité**. La récursivité joue un rôle essentiel lorsque l'on souhaite diviser un problème en plusieurs sous-problèmes (diviser pour régner) et/ou lorsque l'on manipule des structures de données arborescentes. Du point de vue de la machine et du compilateur, autoriser une fonction à s'appeler elle-même (ou, plus généralement, autoriser un groupe de fonctions à s'appeler les unes les autres) ne pose aucune difficulté. Du point de vue du programmeur, pour garder les idées claires, il faut (comme toujours) considérer l'appelé comme une boîte noire – **il suffit de savoir ce que fait l'appelé, pas comment il le fait** – et ignorer le fait qu'appelant et appelé sont la même fonction, car c'est en réalité un détail sans importance. En bref, **l'abstraction procédurale facilite la pensée récursive**.

Remarque 1.15 (Reconnaître une fonction traditionnelle) Quelles notions correspondent, en Java et en OCaml, à une « procédure » ou « fonction » au sens traditionnel de ces mots ?

En Java, une **méthode statique** joue ce rôle. Elle n'a pas d'argument `this` implicite. Les seules données auxquelles elle a accès sont ses arguments et les variables globales (appelés « champs statiques » en Java). Elle est traduite par le compilateur en une série d'instructions. Pour l'appeler, il suffit de connaître l'adresse du début de ces instructions, c'est-à-dire un pointeur de code. On peut dire qu'elle est représentée en mémoire par un pointeur de code.

En OCaml, une **fonction close** joue ce rôle. Une fonction est dite **close** si elle ne mentionne aucune variable définie à l'extérieur : par exemple, la fonction `(fun x -> x + 1)` est close, tandis que la fonction `(fun x -> x + y)` ne l'est pas, car elle mentionne `y`, qui a été définie préalablement. Les seules données auxquelles elle a accès sont ses arguments et les variables globales. Elle est représentée en mémoire par un simple pointeur de code.

Les méthodes statiques de Java et les fonctions closes d'OCaml sont donc analogues aux procédures et fonctions que l'on trouve dans les langages « procéduraux », dont le plus connu et le plus utilisé est probablement le langage C. ◇

1.2.2 Fonction = clôture = objet = service

Les blocs alloués dans le tas, dont nous avons décrit précédemment l'organisation (§1.1), constituent les « données ». La notion traditionnelle de fonction, dont nous venons de rappeler les principes (§1.2.1), permet de définir des « comportements ». Comme nous l'avons suggéré plus haut, il faut **marier données et comportement** pour définir une **abstraction**, c'est-à-dire **offrir un service, sans révéler de quelle manière ce service est implémenté**. Les objets de Java et les fonctions d'OCaml permettent cela.

À partir d'ici, le mot **fonction** désigne une fonction d'OCaml dans toute sa généralité, et non pas seulement une fonction « traditionnelle » ou fonction close (Remarque 1.15). Nous considérons ce mot comme synonyme de **clôture**.

À titre d'exemple, étudions une abstraction simple : un compteur que l'on peut incrémenter de `step` en `step`. On souhaite disposer de trois opérations :

1. La création d'un nouveau compteur, dont la valeur initiale est zéro. Cette opération attend un argument, à savoir un entier `step`, qui reste fixé par la suite.

2. L'incrémentation d'un compteur. La valeur actuelle du compteur augmente de `step`. Cette opération n'a ni argument ni résultat.

3. La consultation du compteur. Cette opération n'a pas d'argument. Son résultat est la valeur actuelle du compteur.

Il est facile d'implémenter cette abstraction en Java, sous forme d'une classe `Counter`, qui permettra ensuite de créer autant d'objets « compteur » qu'on le désire. Commençons donc par cela, comme le propose l'exercice suivant.

Solution page 78. **Exercice 1.7 (Recommandé)** Implémentez en Java une classe `Counter` qui offre le service décrit ci-dessus. Expliquez quel est l'état d'un objet de classe `Counter`, c'est-à-dire quelle est sa représentation en mémoire. ◇

Dans cet exemple exprimé en Java, un appel au constructeur provoque l'allocation d'un nouvel objet dans le tas. Un appel aux méthodes `increment` ou `get` permet ensuite de modifier ou de consulter l'état de cet objet.

Transcrivons à présent cet exemple en OCaml. Un appel au constructeur, par exemple `new Counter (2)`, est transcrit en un appel `new_counter 2`, où `new_counter` est une fonction qui crée un nouveau compteur. Mais, qu'est-ce qu'un compteur ? En Java, un compteur était un objet doté de deux méthodes `increment` et `get`. En OCaml, parce que les fonctions sont des valeurs comme les autres, on peut poser qu'un compteur est une paire de fonctions, qui permettent respectivement d'incrémenter et de consulter la valeur actuelle du compteur. Introduisons un type enregistrement (voir le Détail 1.11) qui décrit cette paire :

```
type counter =
  { increment: unit -> unit; get: unit -> int }
```

Il faut maintenant écrire la fonction `new_counter`, dont le type doit être `int -> counter`, puisqu'elle attend la valeur de l'entier `step`, et produit un nouveau compteur. Ceci est proposé sous forme d'exercice. Essayez de le résoudre, mais si vous n'y parvenez pas, considérez que sa solution fait partie du cours.

Solution page 78. **Exercice 1.8 (Recommandé)** Implémentez la fonction `new_counter`. ◇

Considérer un compteur comme une paire de fonctions peut sembler une idée mystérieuse. Quelle est alors la représentation en mémoire d'un compteur ? C'est une paire, donc un bloc composé de deux champs, qui contiennent chacun un pointeur vers une fonction. Nous parlons ici d'une fonction au sens le plus général, c'est-à-dire une **clôture**. Notre compteur est une paire (allouée dans le tas) de pointeurs vers des clôtures (elles-mêmes allouées dans le tas).

Contrairement à une fonction close, qui n'est que comportement (Remarque 1.15), une clôture marie données et comportement. Par exemple, la composante `get` d'un compteur, qui est une clôture de type `unit -> int`, a accès à certaines données, à savoir d'une part l'entier `step`, d'autre part l'adresse de la référence modifiable où est stockée la valeur actuelle du compteur. Elle a également un comportement, à savoir, lorsqu'elle est appelée, lire et renvoyer la valeur actuelle du compteur.

Pour mieux comprendre ce qu'est une fonction ou une clôture en OCaml, voyons comment, dans le cas général, on peut simuler ce concept en Java. Puisqu'une fonction allie données et comportement, il est naturel de la simuler en Java à l'aide d'un objet. Un objet de Java censé représenter une fonction pourra avoir différents champs, suivant les besoins ; ces champs contiennent les **données** dont la fonction a besoin. Cet objet aura toujours une méthode et une seule, que nous appellerons par exemple `call`. Cette méthode représente le **comportement** de la clôture. Définissons une **interface**, au sens de Java, pour décrire un objet doté d'une méthode `apply` :

```
public interface Function<T, R> {
  R apply (T argument);
}
```

Si `f` est un objet de type `Function<T,R>`, alors l'appel de méthode `f.apply(x)` en Java simule l'appel de fonction `f(x)` en OCaml.

Comme nous ne souhaitons pas fixer le type `T` de l'argument ni le type `R` du résultat, nous en avons fait des **paramètres** de l'interface `Function`. Ainsi, un objet de Java de type

`Function<T, R>` correspond à une fonction OCaml de type `T -> R`. L'interface `Function` fait en réalité partie de la bibliothèque de Java 8.

En conclusion, une bonne façon de comprendre ce qu'est une **clôture** en OCaml est de la considérer essentiellement comme un objet, au sens de Java, qui implémente l'interface `Function`. Donc, comme un objet de Java, **une clôture est allouée dans le tas**, et marie données (stockées dans ses champs) et comportement (déterminé par son étiquette).

Un objet de Java de type `Function<T, R>` ou une clôture d'OCaml de type `T -> R` est **abstrait**, c'est-à-dire opaque : on ne sait pas a priori combien de champs possède un tel objet, ni quel est son contenu. On sait seulement que cet objet rend un certain **service** : étant donné une valeur de type `T`, il produit une valeur de type `R`.

Détail 1.16 (Fonctions à plusieurs arguments en OCaml) En OCaml, une fonction a **toujours** exactement un argument et un résultat. Son type s'écrit `t -> u`, où `t` est le type de l'argument et `u` le type du résultat. Une fonction sans argument ou sans résultat est simulée à l'aide d'une fonction à un argument ou un résultat de type `unit` (Remarque 1.3). Une fonction à plusieurs arguments ou plusieurs résultats est simulée à l'aide d'une fonction à un argument ou un résultat de type « tuple » (Détail 1.12). Par exemple, une fonction qui attend deux entiers et calcule leur somme peut s'écrire `fun (x, y) -> x + y`. Son type est `(int * int) -> int`, que l'on peut écrire aussi `int * int -> int`. Si l'on souhaite nommer cette fonction `add`, on peut écrire :

```
let add = fun (x, y) -> x + y
```

ou bien employer la forme suivante, plus concise mais équivalente :

```
let add (x, y) = x + y
```

Pour appeler cette fonction, il faut bien sûr lui fournir un argument, lequel doit être une paire. On écrit donc, par exemple, `add (3, 4)`.

En mathématiques, au lieu de considérer l'addition comme une fonction qui à une paire d'entiers (x, y) associe l'entier $x + y$, on peut aussi la considérer comme une fonction qui à un entier x associe la fonction qui à un entier y associe la somme $x + y$. En OCaml, puisqu'une fonction est une valeur comme une autre, on peut faire de même. On écrit alors :

```
let add = fun x -> fun y -> x + y
```

ou bien, sous forme plus concise mais équivalente :

```
let add = fun x y -> x + y
```

ou encore :

```
let add x y = x + y
```

Dans ce cas, `add` est une fonction qui renvoie une fonction ; son type est naturellement `int -> (int -> int)`, que l'on peut écrire aussi `int -> int -> int`. Si l'on écrit `add 3`, on obtient une fonction de type `int -> int`, qui ajoute 3 à son argument. Si l'on applique encore cette fonction, en écrivant par exemple `(add 3) 4` ou plus simplement `add 3 4`, on obtient un entier. Cette façon d'écrire la fonction `add` est dite « dans le style de Curry », où « curryfiée ».

La syntaxe de l'appel de fonction est **toujours** `f x`, que l'on peut écrire si on le souhaite avec des parenthèses : `f(x)`. Bien sûr, encore faut-il que l'argument `x` ait bien le type attendu par la fonction `f`.

Tout ceci est remarquablement régulier et naturel. Néanmoins, il est malheureux qu'il existe deux façons de simuler une fonction à deux arguments. En pratique, il faut retenir que si une fonction `f` est de type `t * u -> v`, alors pour l'appeler il faut écrire `f(x, y)`, mais si son type est `t -> u -> v`, alors pour l'appeler il faut écrire `f x y`. On peut dire aussi que si la définition de `f` est de la forme `let f (x, y) = ...` alors pour l'appeler il faut écrire `f(x, y)`, mais si sa définition est de la forme `let f x y = ...` alors pour l'appeler il faut écrire `f x y`. ◇

Solution page 79. **Exercice 1.9 (Recommandé)** On définit en OCaml une notion de « cellule mémoire », qui stocke une valeur entière. Une cellule est présentée à l'utilisateur sous forme d'une fonction de type `int -> int`. Cette fonction a pour double effet de stocker dans la cellule la valeur reçue en argument et de renvoyer la valeur précédemment stockée dans la cellule. Le code s'écrit ainsi :

```
type cell =
  int -> int
let new_cell () : cell =
  let value : int ref = ref 0 in
  let exchange (new_value : int) =
    let old_value : int = !value in
    value := new_value;
    old_value
  in
  exchange
```

Traduisez ce code en Java de la façon suivante. Dans une classe `CellFactory`, écrivez une méthode statique `newCell`, sans argument, qui renvoie une nouvelle cellule, représentée par un objet de type `Function<Integer, Integer>`. Expliquez quelle est la représentation en mémoire de cet objet. ◇

Solution page 80. **Exercice 1.10** En guise de variation sur l'exemple du compteur (Exercice 1.8), on écrit en OCaml un compteur représenté par une seule fonction de type `unit -> int`. Cette fonction a pour double effet d'incrémenter le compteur et de renvoyer son ancienne valeur. Le code s'écrit ainsi :

```
type counter =
  unit -> int
let new_counter (step : int) : counter =
  let value = ref 0 in
  let get_and_increment () =
    let c : int = !value in
    value := c + step;
    c
  in
  get_and_increment
```

Traduisez ce code en Java de la façon suivante. Écrivez une classe `Counter`. Dotez-la d'un constructeur qui attend un argument `step` de type entier. Faites en sorte que cette classe implémente l'interface `Function<Unit, Integer>`. (À propos des types `unit` et `Unit`, voir la Remarque 1.3.) ◇

1.2.3 Clôtures en Java 8

Nous avons présenté un peu plus haut l'interface `Function`, qui est définie dans la bibliothèque de Java 8, et qu'on peut définir soi-même si on utilise une version antérieure de Java. Nous avons expliqué qu'une clôture n'est autre qu'un objet qui satisfait cette interface. Donc, en un sens, les clôtures existent en Java.

Néanmoins, dans les versions de Java antérieures à 8, il manque une syntaxe légère pour construire une clôture. Supposons, par exemple, que l'on dispose d'un entier `x`, et que l'on souhaite construire la fonction qui à `y` associe `x + y`. En OCaml, on écrirait simplement `fun y -> x + y` (Détail 1.16). En Java 7, il faut définir une **classe anonyme** qui implémente l'interface `Function<Integer,Integer>`. On doit donc écrire :

```
new Function<Integer,Integer> () {
  public Integer apply (Integer y) {
    return x + y;
```

```
  }
}
```

C'est bien lourd. Pour alléger cela, Java 8 introduit une syntaxe naturelle et concise pour définir une clôture. Il suffit alors d'écrire :

```
y -> x + y
```

Le compilateur remplace automatiquement cette expression concise par la version beaucoup plus lourde que nous avons donnée précédemment. Il n'est même pas nécessaire d'indiquer quelle interface nous cherchons à satisfaire : le compilateur détermine, d'après le contexte, qu'il s'agit de l'interface `Function<Integer,Integer>`. Il n'est pas non plus nécessaire de déclarer le type de `y` : le compilateur le détermine automatiquement.

Ce mécanisme permet aussi de construire une clôture qui satisfait une interface autre que `Function`. La seule restriction est que cette interface doit être **fonctionnelle** : elle ne doit exiger la présence que d'une seule méthode. Considérons par exemple l'interface `Comparator<T>`. Cette interface est fonctionnelle : elle exige uniquement une méthode `int compare (T o1, T o2)`. Si on souhaite construire un objet de type `Comparator<Integer>` qui représente la relation d'ordre habituelle sur les entiers, on peut écrire en Java 7 :

```
new Comparator<Integer> () {
  public int compare (Integer x, Integer y) {
    return x < y ? -1 : x == y ? 0 : 1;
  }
}
```

mais on préférera probablement écrire en Java 8 :

```
(x, y) ->
  x < y ? -1 : x == y ? 0 : 1
```

Une interface fonctionnelle est indiquée dans la documentation de Java par l'annotation `@FunctionalInterface`. La bibliothèque de Java 8 en définit un grand nombre, parmi lesquelles on trouve `Function<T,R>`, `BiFunction<T,U,R>`, `Consumer<T>`, `Predicate<T>`, `Supplier<T>`, `Callable<V>`, etc.

Cette traduction d'une fonction anonyme en un objet existe dans d'autres langages de programmation. En Scala, par exemple, l'expression `(x: Int) => x + 1` est traduite ainsi par le compilateur :

```
new Function1[Int, Int] {
  def apply(x: Int): Int = x + 1
}
```

Détail 1.17 (Syntaxe des clôtures de Java) Une fonction anonyme à deux arguments s'écrit `(x, y) -> ...`. Le corps de la fonction doit être soit une expression, soit une instruction entre accolades. La fonction `y -> x + y` peut donc s'écrire aussi `y -> { return x + y; }`. Une fonction peut ne pas renvoyer de résultat ; par exemple, `x -> { System.out.println(x); }`. Cette dernière fonction peut s'écrire aussi, sous forme plus concise, `System.out::println`. Elle satisfait (entre autres) l'interface `Consumer<Integer>`. Cette dernière notation est appelée une **référence à une méthode**. ◇

1.2.4 Suspensions

Une fonction d'OCaml de type `unit -> 'a` représente un calcul en attente, c'est-à-dire un calcul pas encore effectué. Le jour où on appelle cette fonction, le calcul a lieu, et on obtient un résultat de type `'a`. De même, en Java, un objet de type `Callable<T>` représente un calcul en attente. Le jour où on appelle sa méthode `call`, le calcul est effectué, et on obtient un résultat de type `T`.

Détail 1.18 (Quelle interface ?) On pourrait préférer employer `Function<Unit,T>` plutôt que `Callable<T>`. Ce choix est essentiellement arbitraire. Une différence mineure entre ces deux interfaces est que la méthode `call` de l'interface `Callable` est autorisée à lancer une exception quelconque. ◇

Le fait de pouvoir construire en mémoire une représentation d'**un calcul en attente** est fondamental. On peut ainsi construire (par exemple) en temps et en espace $O(1)$ une description d'une structure de données dont la taille peut être arbitrairement grande, voire infinie, et dont le contenu peut être arbitrairement coûteux à calculer. Les itérateurs (§4.2) et les flots (§4.3) en donnent un exemple, dans le cas où cette structure de données représente une séquence d'éléments. Seuls les éléments dont l'utilisateur a besoin seront effectivement calculés. En bref, l'idée fondamentale est de **ne calculer que si nécessaire**.

À cette première idée, il est parfois utile d'adjoindre une seconde idée fondamentale : **ne jamais effectuer deux fois un même calcul**. On aimerait que, si l'utilisateur demande deux fois le résultat d'un calcul en attente, alors ce calcul ne soit effectué que la première fois, et que le résultat soit ensuite **mémoisé**. On choisit ainsi d'utiliser plus d'espace, dans l'espoir de gagner du temps.

En OCaml ou en Java, cette mémoisation n'est pas automatique ; si on appelle deux fois `fact(5)`, le calcul de la factorielle de 5 est bien effectué deux fois. Cependant, il ne nous est pas difficile de construire par nous-mêmes un mécanisme de mémoisation. À partir d'une fonction d'OCaml de type `unit -> 'a` ou d'un objet de Java de type `Callable<T>`, nous allons obtenir une **suspension**, ou « *thunk* », c'est-à-dire un objet qui non seulement représente un calcul en attente, mais de plus fournit la garantit que ce calcul sera effectué au plus une fois. La construction de cette notion fait l'objet des exercices suivants.

Solution page 80. **Exercice 1.11 (Recommandé)** En OCaml, on peut définir une suspension comme une fonction particulière, qui la première fois qu'elle est appelée effectue un calcul (potentiellement coûteux) et en mémorise le résultat, de façon à pouvoir renvoyer ce résultat en temps $O(1)$ si jamais elle est appelée à nouveau. On pose donc :

```
type 'a thunk = unit -> 'a
```

Écrivez une fonction `thunk` de type `(unit -> 'a) -> 'a thunk` qui prend un calcul en attente et renvoie une suspension de ce même calcul. ◇

Solution page 82. **Exercice 1.12 (Recommandé)** Cet exercice est l'analogue en Java de l'exercice précédent. On choisit de représenter un calcul en attente par un objet de type `Callable<T>`. Pour représenter une suspension, définissez une classe `Thunk<T>`, dotée d'un constructeur dont la signature est `Thunk (Callable<T> f)`, et qui elle-même implémente l'interface `Callable<T>`. ◇

L'Exercice 1.11 ci-dessus montre que l'on peut définir soi-même, si on le souhaite, la notion de suspension. Néanmoins, pour des raisons de simplicité et d'efficacité, le langage OCaml propose une notion primitive de suspension.

On **construit** une suspension en écrivant `lazy (e)`, où `e` est une expression. Cette expression n'est pas évaluée immédiatement ; elle est suspendue. Si l'expression `e` a le type `t`, alors l'expression `lazy (e)` a le type `t Lazy.t`. Par exemple, posons :

```
let s = lazy (print_endline "Hello!"; 0)
```

Cette définition ne provoque pas l'affichage du message `Hello!`, puisque l'expression qui contient l'appel à `print_endline` n'est pas évaluée immédiatement. La suspension `s` admet le type `int Lazy.t`.

On **évalue** une suspension à l'aide de la fonction primitive `Lazy.force`, dont le type est `'a Lazy.t -> 'a`. Par exemple, posons :

```
let x = Lazy.force s
```

Cette définition provoque l'affichage du message `Hello!`. La variable `x` a pour type `int` et pour valeur 0. Si, une seconde fois, on demande l'évaluation de la suspension `s` :

```
let y = Lazy.force s
```

alors cette fois aucun message n'est affiché, puisque l'expression `print_endline "Hello!"; 0` a déjà été évaluée, et sa valeur mémoisée. La variable `y` a pour type `int` et reçoit la valeur 0.

Une suspension peut en un certain sens être considérée comme un objet **immuable**, puisque, une fois sa valeur calculée, celle-ci reste fixée à jamais. Néanmoins, son implémentation exploite la mémoisation, donc la **modification** d'un champ en mémoire. Il s'agit d'un subtil hybride entre un objet immuable et un objet modifiable.

Nous utilisons les suspensions, plus loin, pour définir les flots (§4.3).

1.3 Modèle de coût

Pour analyser les performances des programmes que nous écrivons, nous devons connaître le **modèle de coût** proposé par le langage de programmation que nous utilisons. En d'autres termes, nous devons connaître le coût, en temps et en espace, de chacune des opérations élémentaires offertes par ce langage.

Soulignons que nous ne parlons pas ici de coût réel, mesuré en secondes pour le temps ou en octets pour l'espace. Le coût réel dépend du compilateur, du système d'exploitation, de la machine, et des conditions d'utilisation de la machine (présence ou non d'autres processus, température !, etc.). Il est en général très difficile à analyser. Nous parlons donc seulement de coût asymptotique à une constante près.

Comme nous l'avons montré dans ce chapitre, Java et OCaml ont d'importants points communs. L'utilisation de la pile pour stocker les variables locales d'une méthode ou fonction est identique. L'organisation des données dans le tas, sous forme de blocs dotés d'une étiquette et de champs, est la même. Les objets de Java et les clôtures d'OCaml, alloués dans le tas, sont très proches. Si l'on a compris cela, on voit que les modèles de coût de Java et d'OCaml sont essentiellement les mêmes.

Considérons d'abord Java.

Les opérations élémentaires dont il faut connaître le coût sont peu nombreuses.

La **création d'un objet**, qui s'écrit `new A (...)`, demande l'allocation et l'initialisation d'un nouveau bloc de mémoire dans le tas. (Elle demande également un appel au constructeur, que l'on peut considérer séparément comme un appel de méthode.) Le nombre de champs de ce bloc est fixé : il ne dépend que du texte du programme. Donc, cette opération coûte $O(1)$ en temps et $O(1)$ en espace dans le tas. La **création d'un tableau**, qui s'écrit `new A [n]`, coûte $O(n)$ en temps et $O(n)$ en espace dans le tas. La **création d'une clôture**, qui s'écrit `x -> ...`, signifie en réalité la création d'un objet (§1.2.3), donc coûte $O(1)$ en temps et $O(1)$ en espace dans le tas. Dans ces trois situations, un bloc est alloué dans le tas. Il sera désalloué par le GC lorsqu'il deviendra inaccessible ; nous discutons cela plus loin.

L'**accès à un champ** en lecture, qui s'écrit `o.f`, et l'accès à un champ en écriture, qui s'écrit `o.f = v`, demandent un accès au tas. Ils ont un coût $O(1)$ en temps.

L'**appel d'une méthode**, qui s'écrit `o.f(...)`, exige d'abord de lire et analyser l'étiquette de l'objet `o` pour en déduire quelle méthode doit être appelée. Il exige ensuite de réserver sur la pile un espace suffisant pour stocker les variables locales de cette méthode. L'espace nécessaire pour cela est fixé : il ne dépend que du texte du programme. Donc, cette opération coûte $O(1)$ en temps et $O(1)$ en espace sur la pile. Cet espace est libéré dès que la méthode termine son exécution. Soulignons que nous mesurons ici uniquement le coût du mécanisme d'appel de méthode. Naturellement, la méthode qui est appelée peut elle-même avoir un coût arbitraire.

L'**accès à une variable locale** en lecture, qui s'écrit `x`, et l'accès à une variable locale en écriture, qui s'écrit `x = v`, ont un coût $O(1)$ en temps. En effet, deux situations peuvent se présenter. Si la méthode accède à une de ses propres variables locales, alors il faut accéder à la

pile. Si la méthode accède à une variable locale en provenance de l'extérieur, comme c'est le cas lors de l'accès à la variable `y` dans la fonction `x -> x + y`, alors il faut en réalité accéder à un champ de la clôture. Dans les deux cas, un accès à la mémoire suffit.

Considérons à présent OCaml.

La **création d'une valeur appartenant à un type algébrique**, qui s'écrit par exemple `Node (f, t1, t2)`, demande l'allocation et l'initialisation d'un nouveau bloc de mémoire dans le tas. Donc, cette opération coûte $O(1)$ en temps et $O(1)$ en espace dans le tas. La construction d'un tuple, par exemple `(t1, t2)`, ou d'un enregistrement, par exemple `{ x = 0; y = 0 }`, sont des notations différentes pour cette même opération. La **création d'un tableau**, qui s'écrit `Array.make n x`, coûte $O(n)$ en temps et $O(n)$ en espace dans le tas. La **création d'une clôture**, qui s'écrit `fun x -> ...`, exige comme en Java la création d'un bloc de mémoire dans le tas, donc coûte $O(1)$ en temps et $O(1)$ en espace dans le tas.

L'**accès à un champ** en lecture prend la forme `o.f` pour les enregistrements, mais aussi `let (x1, x2) = o in ...` pour les paires et `match o with Node (f, x1, x2) -> ...` pour les types algébriques. L'accès à un champ en écriture s'écrit `o.f <- v`. Ces opérations ont un coût $O(1)$ en temps.

L'**appel d'une fonction** coûte $O(1)$ en temps et $O(1)$ en espace sur la pile. Pour une analyse plus fine, on peut considérer qu'un appel terminal (Remarque 1.20) ne consomme pas d'espace sur la pile.

L'**accès à une variable locale**, qui se fait toujours en lecture, a comme en Java un coût $O(1)$.

Remarque 1.19 (Coût des références) Les opérations qui manipulent les références ne sont pas primitives, mais définies dans la bibliothèque (Exercice 1.2). L'appel `ref y` est équivalent à l'expression `{ contents = y }`, donc provoque l'allocation d'un bloc dans le tas. Les appels de fonction `!x` et `x := z` sont équivalents aux expressions `x.contents` et `x.contents <- z`, donc sont des accès en lecture et en écriture à un champ. ◇

Remarque 1.20 (Appel terminal) Informellement, on peut dire qu'un appel d'une fonction f à une fonction g est **terminal** si cet appel est la dernière opération effectuée par f ; ou en d'autres termes, si une fois que g a terminé et renvoyé un résultat, f ne fait que transmettre ce résultat à son propre appelant. Si f et g sont une seule et même fonction, on parle d'**appel récursif terminal**, et on dit que la fonction f est **récursive terminale**.

On pourrait donner une définition plus formelle de la notion d'appel terminal. Nous nous contenterons ici de deux exemples. Dans cette définition de la fonction `List.length`, l'appel récursif n'est pas terminal :

```
let rec length xs =
  match xs with
  | [] ->
      0
  | x :: xs ->
      1 + length xs
```

En effet, après l'appel de `length xs`, il reste encore à effectuer une addition `1 + ...`. Toutefois, en tirant parti de l'associativité de l'addition, on peut reformuler cette définition ainsi :

```
let rec length_aux accu xs =
  match xs with
  | [] ->
      accu
  | x :: xs ->
      length_aux (accu + 1) xs

let length xs =
  length_aux 0 xs
```

On voit que la fonction `length_aux accu xs` calcule la somme `accu + length xs`, et que, par conséquent, la nouvelle définition de `length xs` comme `length_aux 0 xs` est correcte. Dans cette nouvelle définition, l'appel récursif de `length_aux` à elle-même est terminal. On peut noter également (même si cela est moins important) que l'appel de `length` à `length_aux` est également terminal.

Comme nous l'avons noté plus haut, on peut considérer qu'un appel terminal ne consomme pas d'espace sur la pile. En effet, le compilateur OCaml traite un tel appel de façon particulière : les variables locales de l'appelant sont désallouées avant la création des variables locales de l'appelé. Il en résulte que l'appel récursif de `length_aux` à elle-même (par exemple) ne modifie pas la hauteur de la pile. La fonction `length_aux` travaille donc en espace $O(1)$. En fait, le code machine produit par le compilateur OCaml pour `length_aux` est une boucle.

Les solutions des Exercices 3.12 et 4.16, entre autres, offrent des exemples d'appels récursifs terminaux.

Scheme a été l'un des premiers langages à garantir que les appels terminaux sont compilés de façon efficace. Aujourd'hui, OCaml, Haskell, C#, Scala, entre autres, offrent cette garantie. Malheureusement, C et Java traitent les appels terminaux comme des appels ordinaires. ◇

Il nous reste à discuter le coût du **garbage collector** (Remarque 1.2). Celui-ci est appelé lorsqu'on souhaite allouer un bloc dans le tas et lorsque l'espace libre restant est insuffisant pour créer ce bloc. Le GC examine alors tout ou partie du contenu du tas. S'il détermine que certains blocs de mémoire sont inaccessibles, alors il peut détruire ces blocs, c'est-à-dire libérer l'espace qu'ils occupent.

Le temps nécessaire à l'exécution du GC est en général difficile à évaluer, car il dépend d'une part de l'algorithme utilisé par le GC pour déterminer quels objets sont inaccessibles, d'autre part de la taille du tas. On imagine bien, intuitivement, que si le tas est très grand, alors le GC sera appelé moins fréquemment et coûtera donc moins cher. À la limite, lorsque la taille du tas tend vers l'infini, le coût du GC tend vers zéro. Inversement, si le tas est très petit, alors le GC sera appelé plus fréquemment et coûtera donc plus cher. À la limite, si la taille du tas tend vers sa valeur minimum acceptable (à savoir, à un instant donné, la somme des tailles des blocs accessibles), chaque tentative d'allouer **un** bloc risque de demander l'examen par le GC de **tous** les blocs existants. On peut ainsi construire un programme dont la complexité en temps passe de $O(n)$ lorsqu'on lui attribue suffisamment de mémoire à $\Omega(n^2)$ lorsqu'on lui attribue trop peu de mémoire.

Bien que le GC puisse en pratique avoir un coût important, on néglige habituellement son influence lors de l'étude théorique de la complexité d'un algorithme ou d'un programme. Pour certains algorithmes de GC, on peut démontrer que si la taille du tas est à tout instant au moins 2 fois sa valeur minimum acceptable, alors le coût amorti du GC est nul. En d'autres termes, sous cette hypothèse, on peut considérer que l'allocation coûte $O(1)$ et que la désallocation ne coûte rien : cela conduit à une analyse valide du temps de calcul total.

Pour étudier le coût en espace d'un programme, on étudie séparément l'espace maximal requis sur la pile (c'est-à-dire, à une constante près, le nombre maximal d'appels de fonctions imbriqués) et l'espace maximal requis dans le tas (c'est-à-dire la valeur maximale au cours du temps de la somme des tailles des blocs accessibles). Cette dernière étude peut être assez subtile, comme l'illustrent par exemple les Exercices 4.26 et 4.27.

Chapitre 2

Abstraction

Au cours du Chapitre 1, nous avons mis en évidence d'importants points communs entre deux langages de programmation que l'on aurait pu croire au premier abord très différents, à savoir Java et OCaml. D'une part, nous avons comparé l'organisation des **données** en mémoire et constaté que, pour ces deux langages, elle peut être comprise et décrite en termes de sommes, de produits, et de récursivité. D'autre part, nous avons présenté la façon dont un **comportement** est représenté par un objet de Java ou par un groupe de fonctions d'OCaml.

Vis-à-vis de ces deux aspects, les **types** jouent un rôle fondamental. D'abord, ils constituent **un langage de description des données**. À l'aide d'un petit nombre de concepts universels, à savoir types primitifs, types produits, types sommes, et types récursifs, on peut décrire l'organisation en mémoire de n'importe quelle structure de données : liste, arbre, etc. Ensuite, ils offrent **un langage de description de composants logiciels**. En effet, à l'aide des types d'objet de Java (c'est-à-dire les classes et interfaces) ou bien à l'aide des types de fonction d'OCaml, on décrit la façon dont un composant logiciel est prêt à communiquer avec l'extérieur : « un compteur fournit une méthode ou fonction `get` qui n'attend aucun argument et renvoie un résultat entier ; une méthode ou fonction `increment` qui, etc. ». Bref, les types décrivent le **service** rendu par ce composant.

Cependant, le rôle des types n'est pas seulement descriptif. Leur principale force, en réalité, est de nous permettre de **construire et faire respecter des abstractions**. Selon Reynolds (1983), « *type structure is a syntactic discipline for enforcing levels of abstraction* ».

Que signifie ceci ? Souvent, décrire la structure concrète, détaillée, des données dans la mémoire n'est pas souhaitable. Une date, par exemple, est intuitivement un objet auquel on peut appliquer certaines opérations, comme « ajouter une semaine » ou « comparer à une autre date ». Certes, une date doit être représentée en mémoire d'une certaine manière, par exemple par un triplet d'entiers (année, mois, jour), par un unique entier (la norme POSIX représente une date comme le nombre de secondes écoulées depuis le 1er janvier 1970), ou encore par un nombre réel à virgule flottante. Mais, en dehors du composant logiciel qui définit la notion de date et les opérations associées, la façon dont les dates sont représentées en mémoire est un détail qui n'a pas et ne doit pas avoir d'importance. En d'autres termes, vue de l'extérieur, la notion de « date » est une **abstraction**. On souhaite que les utilisateurs manipulent des valeurs de type « date » comme s'il s'agissait d'un type primitif, au même titre que les types « entier » et « booléen ». On souhaite de plus qu'il soit impossible de confondre une date et un entier, même s'il est vrai aujourd'hui qu'une date est représentée en mémoire sous forme d'un entier. Une telle confusion soit serait une erreur d'inadvertance de la part de l'utilisateur, soit résulterait du fait que l'utilisateur sait (ou croit savoir) qu'une date est représentée en mémoire par un entier, et souhaite en tirer parti, par exemple en écrivant `let tomorrow (d : date) : date = d + 86400`. Même dans ce dernier cas, l'utilisateur a tort, car si « date » est une abstraction, alors son auteur conserve la liberté de modifier la manière dont une date est représentée en mémoire.

La notion d'abstraction n'est pas propre à l'informatique. Elle existe également, entre autres domaines, en mathématiques. La fable présentée par Reynolds (1983) souligne le fait qu'un nombre complexe ne doit pas être considéré comme une paire de réels sous forme cartésienne $a + ib$, ni comme une paire de réels sous forme polaire $re^{i\theta}$. Mieux vaut considérer un nombre complexe comme un élément d'un certain corps $\mathbb{C}$, dont on sait qu'il est une clôture algébrique de $\mathbb{R}$. En dehors de la leçon où on construit effectivement le corps des nombres complexes à l'aide d'une définition concrète, le corps $\mathbb{C}$ peut et doit être traité comme une abstraction.

En informatique, **comment construire et faire respecter une abstraction ?** Plus précisément, quels outils un langage de programmation fournit-il pour cela ? On peut apporter à cette question (au moins) deux réponses fondées sur deux outils distincts.

La première réponse suppose que la discipline des types est imposée statiquement, c'est-à-dire imposée par le compilateur. C'est le cas pour Java et pour OCaml. Alors, le concepteur de l'abstraction « date » peut indiquer au compilateur que le type « date » doit être considéré comme un **type abstrait**, c'est-à-dire un type dont la définition est connue du concepteur, mais inconnue des utilisateurs. Ainsi, à l'intérieur du composant qui définit l'abstraction « date », on sait, par exemple, que le type « date » est synonyme du type « entier », ou bien que le type « date » possède un champ de type « entier ». À l'extérieur de ce composant, toutefois, cette information ne peut pas être exploitée : le compilateur l'interdit. À l'extérieur, donc, on sait que « date » est un type, mais rien de plus. Si un programme écrit par un utilisateur fournit un entier là où une date est attendue, alors ce programme est considéré comme erroné, et est rejeté par le compilateur.

La seconde réponse s'appuie sur la notion de fonction, au sens de **clôture** (§1.2.2). Elle suppose qu'une clôture est opaque, c'est-à-dire que la seule manière possible d'utiliser une fonction est d'appeler cette fonction. C'est le cas des langages typés statiquement, comme Java 8 et OCaml, mais aussi de certains langages typés dynamiquement, comme Scheme et JavaScript. À l'aide de cette abstraction primitive, c'est-à-dire fournie par le langage de programmation, on peut alors construire de nouvelles abstractions. Un « compteur », par exemple, peut être défini comme un objet qui rend deux services, à savoir `increment` et `get` (§1.2.2). On peut proposer, en Java, une définition du type « compteur » sous forme d'une interface :

```
public interface ICounter {
  void increment ();
  int get ();
}
```

En OCaml, on peut poser qu'un « compteur » est une paire de clôtures :

```
type counter =
  { increment: unit -> unit; get: unit -> int }
```

Dans un langage typé dynamiquement, comme JavaScript, on n'écrit pas de définitions de types. Néanmoins, on pourrait poser (sous forme d'un commentaire) qu'un « compteur » est un objet doté de deux méthodes `increment` et `get`. Dans tous les cas, la notion de « compteur » est bien une abstraction, au sens où l'utilisateur n'a accès qu'aux deux opérations prévues par le concepteur, et ne sait pas comment un compteur est représenté en mémoire. Si un programme écrit par l'utilisateur demande à lire le champ `value` d'un compteur, alors ce programme est rejeté (dès la compilation si le langage est typé statiquement ; pendant l'exécution si le langage est typé dynamiquement), et ce indépendamment du fait que l'objet « compteur » ait ou non, en réalité, un champ nommé `value`.

Les deux outils présentés ci-dessus, à savoir les types abstraits et les clôtures, sont des mécanismes de **protection** : ils interdisent à l'utilisateur certaines opérations. Dans les deux cas, c'est grâce à une certaine discipline de types (imposée soit statiquement, soit dynamiquement) que l'on peut distinguer opérations permises et opérations interdites.

En résumé, dans un langage de programmation de haut niveau, **les types constituent un mécanisme de protection, donc permettent de construire des abstractions inviolables**.

Parmi les deux mécanismes de protection que nous venons de présenter, à savoir les types abstraits et les clôtures, l'un est-il préférable à l'autre, ou plus puissant que l'autre ? Tenter de répondre formellement à cette question nous conduirait un peu loin. Nous l'aborderons lors de certains exercices, dont les Exercices 3.21 et 4.13.

Entre les langages typés statiquement et ceux typés dynamiquement, lesquels sont préférables, ou plus puissants ? Les premiers offrent quelques avantages importants :

1. Une discipline de types statique, c'est-à-dire imposée par le compilateur, **garantit, avant l'exécution du programme, l'absence de certaines erreurs**. Par exemple, on ne peut pas fournir par erreur un argument entier à une fonction qui attend un tableau d'entiers. C'est en principe un avantage énorme vis-à-vis des langages typés dynamiquement : cela signifie que ces erreurs sont détectées avant la livraison, chez le fournisseur du composant logiciel, et non pas après la livraison, chez le client.

2. Si les types sont vérifiés par le compilateur, alors ils constituent un langage formel, précis, de description des données et des comportements. Ce ne sont pas des commentaires : leur exactitude est vérifiée par le compilateur. Ils offrent donc **une documentation précise, à jour, sans erreur**.

3. Enfin, comme nous l'avons expliqué plus haut, dans un langage de programmation typé statiquement, on peut employer un **type abstrait** pour imposer une abstraction, tandis que dans un langage typé dynamiquement, cet outil n'existe pas. En bref, les types nous interdisent certaines opérations, ce qui pourrait sembler néfaste ; mais **ils interdisent aussi à nos adversaires certaines opérations**, ce qui est clairement souhaitable dans une situation où on ne fait pas confiance au code écrit par d'autres.

Cela dit, une discipline de types statique peut être lourde ou restrictive : il peut être pénible voire impossible « d'expliquer au compilateur », à l'aide de définitions de types bien senties, pourquoi un programme est correct. C'est pourquoi les langages de programmation typés dynamiquement offrent l'avantage d'une plus grande flexibilité. [1]

Au sens où nous venons d'employer ce mot, **abstraire, c'est cacher**. Pour construire une abstraction, on délimite une frontière entre intérieur et extérieur, c'est-à-dire entre fournisseur et utilisateur d'un composant logiciel. L'abstraction va donc de pair avec le découpage d'un programme en différents composants, et implique une relative **isolation** entre composants, chacun ignorant la façon dont les autres sont construits. Cette isolation bénéficie à tous. Du point de vue du fournisseur, l'abstraction empêche le monde extérieur de perturber le bon fonctionnement du composant. Par exemple, si le composant implémente des ensembles à l'aide d'arbres binaires de recherche équilibrés, alors, parce que le type « ensemble » est abstrait, il est impossible, de l'extérieur, de construire de toutes pièces un arbre non équilibré. L'**invariant** de bon équilibre est garanti, et ce, **quelle que soit** la manière dont le composant est utilisé. On peut d'ailleurs effectuer un **test unitaire** pour vérifier que l'invariant de bon équilibre est bel et bien respecté. Du point de vue de l'utilisateur, l'abstraction permet de remplacer facilement un composant par un autre de même type. Si une implémentation du type abstrait « ensemble » à l'aide d'arbres équilibrés n'est pas satisfaisante, par exemple pour des raisons de performance, alors il est possible de la remplacer par une implémentation basée sur des tables de hachage. Cette nouvelle implémentation fournit en apparence le même type abstrait « ensemble » et les mêmes opérations, donc les deux implémentations sont **interchangeables**. L'abstraction favorise ainsi l'évolution des logiciels, ainsi que leur construction à partir de composants standardisés. Au lieu d'abstraction, on parle parfois d'**encapsulation** ou d'**information hiding**.

Cependant, le mot admet également un autre sens : **abstraire, c'est généraliser ; abstraire, c'est paramétrer**. En mathématiques, par exemple, le théorème fondamental de l'arithmétique,

1. À propos du débat entre partisans des deux bords, j'aime beaucoup cette phrase de Reynolds (*Three approaches to type structure*, 1985) : *The division of programming languages into two species, typed and untyped, has engendered a long and rather acrimonious debate. One side claims that untyped languages preclude compile-time error checking and are succinct to the point of unintelligibility, while the other side claims that typed languages preclude a variety of powerful programming techniques and are verbose to the point of unintelligibility. From the theorist's point of view,* ***both sides are right****, and their arguments are the motivation for seeking type systems that are more flexible and succinct than those of existing typed languages.*

qui affirme que tout nombre non nul admet une décomposition en facteurs premiers, unique à éléments inversibles près, n'est pas vrai seulement dans $\mathbb{Z}$, mais plus généralement dans tout anneau factoriel, par exemple dans l'anneau de polynômes $\mathbb{Z}[X]$. Si une construction, par exemple la construction de la fonction *pgcd*, s'appuie uniquement sur ce théorème, alors mieux vaut effectuer cette construction dans le cadre d'un anneau factoriel A quelconque, plutôt que dans le cas particulier de $\mathbb{Z}$. La construction de *pgcd* est alors **paramétrée** par l'anneau factoriel A, et peut ensuite être utilisée **pour tout** anneau factoriel A concret. En informatique, la situation est identique. La construction d'un algorithme de tri, par exemple, se fait non pas dans le cas particulier d'un tableau d'entiers, mais de préférence dans le cadre plus général d'un tableau d'éléments de type A, où A est arbitraire, pourvu que A soit muni d'un ordre total calculable. L'algorithme de tri est alors paramétré par le type A et par une fonction de comparaison. Il peut être utilisé pour trier des tableaux d'entiers, des tableaux de dates, etc. ; on dit qu'il est **polymorphe**. L'abstraction, comprise au sens de généralisation, favorise la **réutilisation** d'un composant logiciel dans différents scénarios.

Les deux sens du mot « abstraire » sont, naturellement, liés l'un à l'autre. En réalité, ils sont duaux, comme les quantificateurs existentiel et universel en mathématiques. Fournir une abstraction « ensemble », c'est affirmer qu'**il existe** un type « ensemble » (dont on ne révèle pas la définition) qui admet les opérations suivantes : ensemble vide ; insertion d'un élément ; test d'appartenance ; etc. Fournir une implémentation polymorphe d'un algorithme de tri, c'est affirmer que **quel que soit** le type A et quelle que soit sa fonction de comparaison, le tri est possible. Dans les deux scénarios, on est amené à travailler avec des types et des opérations dont on ne connaît pas la définition, soit parce qu'elle est volontairement cachée, soit parce qu'elle n'est pas fixée.

Au cours de ce chapitre, nous nous focalisons sur l'interprétation existentielle du mot « abstraction » ; son interprétation universelle fait l'objet du Chapitre 3.

Nous étudions d'abord (§2.1) comment définir une abstraction sous forme d'un type abstrait, c'est-à-dire en Java à l'aide d'une classe dont certains champs ou méthodes sont privés, et en OCaml à l'aide d'un type dont la définition n'est pas connue de l'utilisateur. Nous indiquons ensuite (§2.2) comment définir une abstraction sous forme d'une clôture, c'est-à-dire en Java à l'aide d'une interface, et en OCaml à l'aide d'une fonction ou d'un groupe de fonctions.

2.1 Abstraction fondée sur un type abstrait

On peut imaginer de nombreux exemples d'abstractions, ou types de données abstraits. Nous en avons déjà rencontré quelques exemples dans le chapitre précédent : par exemple, la notion de suspension (§1.2.4) constitue une abstraction, au sens où son utilisateur dispose de deux opérations sur les suspensions, à savoir la création et l'évaluation d'une suspension, mais ne sait pas et ne doit pas savoir comment les suspensions sont implémentées. Le livre de Conchon et Filliâtre (2014, chapitres 4 à 6) présente diverses sortes de tableaux, d'ensembles, et de files, implémentées en OCaml. Celui de Liskov et Guttag (2001, chapitre 5) propose deux exemples en Java : des ensembles modifiables dont les éléments sont des entiers, `IntSet` ; et des polynômes immuables dont les coefficients sont des entiers, `Poly`. Ici, nous prendrons un exemple très simple : celui d'une **pile** dont les éléments sont des entiers.

Naturellement, fixer le type des éléments n'est pas réellement souhaitable : cela rend notre implémentation des piles moins réutilisable. L'écriture de code polymorphe, où le type des éléments n'est pas fixé, sera l'objet du chapitre suivant (§3.1).

Une pile est une file d'attente modifiable obéissant à la discipline « *LIFO* » : « *last in, first out* ». En d'autres termes, une pile **représente** une liste d'éléments, et est typiquement munie de deux (et seulement deux) opérations permettant de modifier son contenu : l'opération `push` insère un élément en tête de la liste ; l'opération `pop` extrait l'élément situé en tête de la liste. Il faut également une opération qui crée une nouvelle pile vide. D'autres opérations peuvent éventuellement être offertes pour tester si une pile est vide, connaître le nombre d'éléments

présents dans une pile, itérer sur les éléments présents dans une pile, etc. Cette structure de données est utile dans de nombreuses situations, en particulier lorsqu'on souhaite écrire sous forme itérative un algorithme intrinsèquement récursif, par exemple un parcours en profondeur d'abord.

Il existe plusieurs manières, plus ou moins complexes, d'implémenter une pile. Par exemple, on peut choisir de la représenter en mémoire par une liste chaînée ; ou bien par un tableau, que l'on redimensionnera au besoin ; ou bien par une liste chaînée de tableaux de taille fixe. Chacune de ces possibilités offre des caractéristiques légèrement différentes en termes d'efficacité en temps et en espace.

La pile est un exemple simple mais typique d'abstraction, ou de type de données abstrait. L'utilisateur n'en connaît que la **spécification**. Il doit savoir seulement qu'il existe un type nommé `stack` ; qu'une valeur de type `stack` représente abstraitement une liste d'éléments ; qu'il existe des opérations `push`, `pop`, etc. ; que l'effet de `push` est abstraitement d'insérer un élément en tête de liste, et que l'effet de `pop` est abstraitement d'extraire un élément en tête de liste. L'utilisateur ne connaît pas l'**implémentation** concrète de cette structure de données. Il n'a pas à savoir comment la pile est réellement représentée en mémoire et comment les opérations modifient réellement le contenu de la mémoire. Cela laisse à l'implémenteur la liberté de choisir entre différentes techniques, et cela lui permet de protéger son code contre toute interférence.

La distinction entre spécification et implémentation est absolument fondamentale. C'est l'un des concepts centraux du génie logiciel (« *software engineering* »). Elle permet d'assembler des composants qui ont été développés indépendamment les uns des autres, et qui peuvent continuer à évoluer indépendamment les uns des autres, pourvu que leur spécification soit clairement définie et fixée.

Cette distinction prend différentes formes dans différents langages de programmation.

En Java, spécification et implémentation prennent typiquement place toutes deux au sein d'un même fichier, par exemple `IntStack.java`. Le mot-clef **`private`** signale alors les éléments (champs, constructeurs, méthodes) qui font partie de l'implémentation et auxquels l'utilisateur n'a pas accès, tandis que le mot-clef **`public`** signale ceux qui font partie de la spécification, auxquels l'utilisateur a accès.

Détail 2.1 (Modificateurs d'accès de Java) Il existe en fait quatre modificateurs d'accès en Java, à savoir **`private`**, **`public`**, **`protected`**, et « *package-private* », ce dernier étant signalé par l'absence de mot-clef. ◇

Exercice 2.1 (Recommandé) Définissez en Java une classe `IntStack` qui implémente une pile dont les éléments sont des entiers. Vous vous appuierez sur une liste chaînée immuable, que vous définirez également. ◇ Solution page 82.

Exercice 2.2 (Recommandé) On souhaite définir une classe `RichIntStack` analogue à la classe `IntStack` de l'Exercice 2.1 et qui offre de plus une méthode `peek`, qui renvoie l'élément situé au sommet de la pile, sans modifier celle-ci ; et une méthode `size`, qui renvoie le nombre d'éléments actuellement stockés dans la pile. On souhaite que ces deux méthodes aient une complexité en temps $O(1)$. Quelle que soit la technique algorithmique utilisée, il existe au moins trois manières d'organiser le code : Solution page 84.

1. créer une copie de la classe `IntStack`, nommée `RichIntStack`, et la modifier ;
2. définir une classe `RichIntStack` qui hérite de la classe `IntStack` ;
3. définir une classe `RichIntStack` qui possède un champ de type `IntStack`.

Réfléchissez à ces trois approches et implémentez (au moins) l'une d'elles. ◇

Exercice 2.3 Définissez en Java une classe `IntArrayStack` qui implémente une pile dont les éléments sont des entiers. Vous vous appuierez sur la classe `ArrayList`, fournie par la bibliothèque, qui implémente un tableau redimensionnable. ◇ Solution page 87.

En OCaml, la spécification et l'implémentation de l'abstraction `Stack` se font typiquement dans deux fichiers séparés, nommés par convention `Stack.mli` et `Stack.ml`. L'utilisateur de la pile ne connaît que ce qui est déclaré dans `Stack.mli`, et n'a pas accès aux définitions situées dans `Stack.ml`.

Le fichier `.mli` contient zéro, une ou plusieurs déclarations ou définitions de **types** et zéro, une ou plusieurs déclarations de **valeurs**, qui sont souvent (mais pas nécessairement) des fonctions. Le tout constitue ce que l'on appelle une **signature**.

Nous distinguons ici les mots « définition » et « déclaration ». Une **définition** introduit un nom et, en même temps, lui attribue une signification. Par exemple, en OCaml, la définition `let x = 5` introduit la variable `x` et lui attribue la valeur `5`. La définition `type t = int` définit le type `t` comme un synonyme du type `int`. La définition `type food = Meat | Grass` définit le type `food` comme un type somme à deux branches. Une **déclaration** introduit un nom mais n'en donne pas la signification. Par exemple, en OCaml, la déclaration `val x: int` affirme l'existence d'une variable `x` de type `int`, mais n'en donne pas la valeur. La déclaration `type t` affirme l'existence d'un type nommé `t`, mais n'en donne pas la définition concrète.

La signature des piles dont les éléments sont des entiers, contenue dans un fichier `.mli`, par exemple `Stack.mli`, est, sous sa forme la plus réduite, la suivante :

```
type stack
val create: unit -> stack
val push: int -> stack -> unit
val pop: stack -> int option
```

Cette signature affirme d'abord l'existence d'un type `stack`. Ce type est **abstrait**, puisqu'il est déclaré, mais non défini. Sa définition concrète doit être donnée dans le fichier `Stack.ml`, mais n'est pas connue à l'extérieur. Cette signature affirme ensuite l'existence de trois fonctions nommées `create`, `push`, `pop`, dont elle donne les types. La signature ne dit pas formellement, mais on peut écrire sous forme d'un commentaire, que :

1. `create()` renvoie une nouvelle pile vide.
2. `push x s` a pour effet d'insérer l'élément `x` en tête de la pile `s`, et ne renvoie « rien », ou plus précisément, renvoie la valeur `()`, dont le type est `unit`.
3. `pop s` a pour effet d'extraire l'élément situé en tête de la pile `s`, si cette pile est non vide. La fonction `pop` produit un résultat de type `int option`, ce qui signifie intuitivement qu'elle renvoie soit « rien », soit un entier. Plus précisément, elle renvoie soit la valeur `None`, soit une valeur de la forme `Some x`, où `x` est un entier (Remarque 1.4).

Ce commentaire constitue la **spécification** de l'abstraction « pile ». Si on le souhaite, on peut le reformuler de manière encore plus précise en affirmant que, à tout instant, une pile **représente** une certaine liste d'éléments (entiers), et en utilisant ce **prédicat de représentation** pour décrire de façon parfaitement rigoureuse le comportement de chaque opération :

1. `create()` renvoie une nouvelle pile qui représente la liste vide.
2. Si la pile `s` représente une liste `xs`, alors après l'appel `push x s`, la pile `s` représente la liste `x :: xs`.
3. Si la pile `s` représente une liste `x :: xs`, alors après l'appel `pop s`, la pile `s` représente la liste `xs`, et l'appel renvoie `Some x`. Si la pile `s` représente la liste vide, alors après l'appel `pop s`, la pile `s` représente toujours la liste vide, et l'appel renvoie `None`.

L'implémentation des piles, contenue dans le fichier `Stack.ml`, doit donner la définition du type `stack`, ainsi que les définitions des fonctions `create`, `push`, `pop`. Elle peut également définir d'autres types et d'autres fonctions, si l'implémenteur le juge utile pour ses propres besoins ; cependant, ces types et fonctions ne seront pas accessibles à l'utilisateur.

Nous avons souligné le fait que l'utilisateur connaît l'existence du type `stack`, mais ne connaît pas sa définition : c'est un type abstrait. Il est intéressant de souligner que, de même,

le prédicat de représentation, « `s` représente `xs` », est un **prédicat abstrait**. L'utilisateur connaît l'existence de ce prédicat – il sait que chaque pile représente à chaque instant une certaine liste d'éléments – mais ne sait pas comment il est défini. Si on le souhaite, sa définition peut être donnée, sous forme d'un commentaire, dans le fichier `Stack.ml`. Par exemple, si le type `stack` est défini comme un enregistrement contenant un tableau `content` et un entier `limit`, comme dans notre solution à l'Exercice 2.6, alors on peut affirmer que « la pile `s` **représente** la liste des éléments du tableau `s.content` compris entre les indices `0` inclus et `s.limit` exclus ». Ceci constitue une définition du prédicat « `s` représente `xs` ».

Nous avons suggéré de donner la spécification sous forme d'un commentaire dans le fichier `.mli`, et de donner la définition du prédicat de représentation sous forme d'un commentaire dans le fichier `.ml`. C'est le mieux qu'on puisse faire ici, car le compilateur ne comprend pas ces notions. Toutefois, il faut savoir qu'il existe des outils de **preuve de programmes**, pour Java, pour OCaml, et pour d'autres langages de programmation, qui permettent de définir formellement la spécification et le prédicat de représentation, et qui peuvent alors vérifier que le code est conforme à la spécification.

Le livre de Liskov et Guttag (2001, §5.5.1) décrit la notion de prédicat de représentation dans le cadre de Java. Il est présenté sous forme d'une **fonction d'abstraction** qui à une pile `s` associe la liste d'éléments `xs` que cette pile représente.

Exercice 2.4 Afin d'illustrer la notion de clôture, nous avons présenté précédemment (§1.2.2) une abstraction « compteur » réalisée à l'aide d'une paire de clôtures : Solution page 88.

```
type counter =
  { increment: unit -> unit; get: unit -> int }
```

Pour construire un compteur, on avait une fonction `new_counter` de type `int -> counter`. On souhaite maintenant adopter une approche différente et définir l'abstraction « compteur » sous forme d'un type abstrait doté de trois opérations `make`, `increment`, et `get`. Donnez cette définition, sous forme de deux fichiers `Counter.mli` et `Counter.ml`. ◇

Exercice 2.5 (Recommandé) Proposez en OCaml une implémentation des piles conforme à la signature ci-dessus. Vous vous appuierez sur les listes immuables fournies par OCaml et sur une référence modifiable, de façon à proposer l'implémentation la plus simple possible. ◇ Solution page 88.

Exercice 2.6 Proposez en OCaml une implémentation des piles conforme à la signature ci-dessus. Vous vous appuierez sur un tableau, que vous redimensionnerez au besoin, c'est-à-dire que vous allouerez un nouveau tableau, plus grand, dans lequel vous copierez le contenu du tableau existant. Les opérations dont vous disposez sur les tableaux sont fournies par le module `Array` de la bibliothèque. ◇ Solution page 89.

Exercice 2.7 Quels sont, selon les vous, les avantages et les inconvénients de l'implémentation des piles à base de listes chaînées, vis-à-vis de l'implémentation des piles à base de tableaux redimensionnables (Exercices 2.5 et 2.6) ? Proposez en OCaml une implémentation des piles qui offre une forme de compromis entre les deux approches précédentes. ◇ Solution page 89.

Exercice 2.8 Alex travaille sur un projet pour lequel il a besoin d'une structure de file d'attente « *first-in, first-out* » modifiable. Il est pressé et ne sait pas encore si l'efficacité de cette structure de données sera essentielle ou non pour son projet ; aussi écrit-il l'implémentation la plus simple qui lui vient à l'esprit, en s'appuyant sur les listes immuables fournies par OCaml et sur une référence modifiable : Solution page 91.

```
type 'a queue =
  'a list ref
let create () =
  ref []
let insert q x =
  q := !q @ [x]
```

```
let extract q =
  match !q with
  | []      -> None
  | x :: xs -> q := xs; Some x
```

Comme il sait que cette implémentation est loin d'être idéale, Alex prend soin de présenter cette structure de données sous forme d'un type abstrait. Il sait que, plus tard, il pourra remplacer ce code par une autre implémentation de la même signature, sans que cela puisse affecter le reste de son projet.

Écrivez la signature attribuée par Alex à ce module. Expliquez pourquoi son code est mauvais. Enfin, proposez une implémentation plus efficace de la même signature. ◇

Solution page 95.

Exercice 2.9 En général, plus les opérations proposées par une abstraction sont nombreuses, plus cette abstraction est difficile à implémenter efficacement. Ajouter une seule opération à une signature, ou supprimer une seule opération, peut avoir un impact important sur la manière dont on peut implémenter efficacement cette signature. En voici quelques exemples. Les trois questions qui suivent sont indépendantes les unes des autres. On demande à chaque fois de modifier la signature et l'implémentation des piles données dans la solution de l'Exercice 2.5 en ajoutant ou en supprimant une opération, et en donnant l'implémentation la plus efficace possible.

1. Modifiez la signature et l'implémentation des piles pour ajouter une opération `size` de type `stack -> int` telle que `size s` renvoie le nombre actuel d'éléments de la pile `s`.
2. Modifiez la signature et l'implémentation des piles pour ajouter une opération `member` de type `int -> stack -> bool` telle que `member x s` indique si `x` figure actuellement parmi les éléments de la pile `s`.
3. Modifiez la signature et l'implémentation des piles pour supprimer l'opération `pop`. ◇

2.2 Abstraction fondée sur une clôture

Comme nous l'avons expliqué au début de ce chapitre, pour faire respecter une abstraction, il faut un mécanisme de protection. Dans la partie précédente (§2.1), nous nous sommes appuyés sur des interdictions imposées par le compilateur, à savoir en Java l'interdiction d'accéder depuis l'extérieur à un composant marqué **`private`**, et en OCaml l'interdiction d'exploiter depuis l'extérieur la définition d'un type abstrait. Dans ce qui suit, nous exploitons un autre mécanisme de protection, à savoir l'impossibilité d'inspecter le contenu d'une clôture. Cette interdiction peut être imposée soit par le compilateur (c'est le cas des langages typés statiquement, dont Java et OCaml), soit par le système d'exécution (c'est le cas des langages typés dynamiquement, par exemple JavaScript).

Nous employons ici le mot **clôture** en un sens légèrement élargi. Du côté d'OCaml, nous l'employons pour désigner une clôture au sens habituel ; du côté de Java, nous l'utilisons pour désigner n'importe quel objet dont le type est une **interface**. (Cela inclut donc les clôtures de Java 8.) Par exemple, parce que `Iterable` est une interface, nous pourrons dire qu'un objet de type `Iterable<Integer>` est une clôture.

D'un point de vue concret, une clôture est représentée par un bloc alloué dans le tas. Comme tout autre bloc de mémoire, elle est dotée d'une étiquette et d'un certain nombre de champs (§1.1). Cependant, il est impossible d'accéder directement au contenu de ce bloc de mémoire. En effet, **la seule façon d'utiliser une clôture est de l'appeler**. Plus précisément, en OCaml, on ne peut rien faire avec une fonction, sinon l'appeler ; et en Java, on ne peut rien faire avec un objet dont le type est une interface, sinon appeler une de ses méthodes.

On peut tirer parti de cela pour protéger une abstraction. Par exemple, en OCaml, la notion de « pile » dont les éléments sont des entiers peut être présentée comme une paire de fonctions :

```
type stack =
  { push: int -> unit; pop: unit -> int option }
```

L'opération de construction d'une nouvelle pile est alors donnée par une fonction `new_stack` de type `unit -> stack`. On retrouve le style déjà employé plus tôt (§1.2.2) pour présenter un « compteur » comme une paire de fonctions `increment` et `get`. Nous laissons au lecteur le soin de vérifier comment on écrit la fonction `new_stack`.

En Java, de façon analogue, on peut définir une **interface** qui décrit les services offerts par une « pile » à éléments entiers :

```
public interface IIntStack {
  void push (int x);
  int pop () throws NoSuchElementException;
}
```

Pour pouvoir construire des objets de type `IIntStack`, il faut définir une classe qui satisfait (ou qui « implémente ») l'interface `IIntStack`. On peut par exemple ajouter à la définition de la classe `IntStack` (Exercice 2.1) une clause **`implements`** `IIntStack` :

```
public class IntStack implements IIntStack { ... }
```

Le compilateur Java vérifie d'abord que les opérations exigées par l'interface `IIntStack` sont bel et bien fournies par la classe `IntStack`. Si c'est bien le cas, alors le compilateur considère qu'il existe une relation de **sous-typage** entre les types `IntStack` et `IIntStack`. En termes plus clairs, cela signifie que tout objet de type `IntStack` **est** aussi un objet de type `IIntStack`. Ainsi, par exemple, l'expression **`new`** `IntStack ()` peut être utilisée pour construire un objet de type `IIntStack`.

En Java 8, si on souhaite construire un objet analogue à la fonction `new_stack` mentionnée ci-dessus, on peut employer l'expression `IntStack::new`. Cette expression, appelée « référence à une méthode » (Détail 1.17), construit un objet doté d'une seule méthode dont le corps est `{ return new IntStack (); }`. On peut donc écrire, par exemple,

```
Supplier<IIntStack> new_stack = IntStack::new;
```

On dispose ainsi d'un objet `new_stack` de type `Supplier<IIntStack>`, tout à fait analogue à la fonction `new_stack` de type `unit -> stack` définie en OCaml. Pour obtenir une nouvelle pile, on appelle `new_stack.get()`. L'objet `new_stack` est parfois appelé **usine** ou **factory** car son unique rôle est de construire un autre objet.

Le mécanisme employé ici pour protéger l'abstraction « pile » n'est pas le même que dans la partie précédente (§2.1). Du côté d'OCaml, le type `stack` ci-dessus n'est pas stricto sensu un « type abstrait », puisque sa définition est connue de l'utilisateur : c'est un type enregistrement doté de deux champs, qui contiennent des fonctions. Du côté de Java, le type `IIntStack` ci-dessus n'a aucune composante marquée **`private`** : il révèle simplement l'existence de deux méthodes. Dans les deux cas, l'utilisateur n'a accès qu'aux méthodes `push` et `pop`, et rien de plus, parce qu'une clôture est un objet opaque. On s'appuie donc sur le fait que la notion de clôture est une abstraction primitive. On ne s'appuie pas fondamentalement sur le fait que le langage est typé statiquement : en effet, dans un langage typé dynamiquement, on peut exploiter les clôtures de la même manière, comme le montre l'exercice qui suit.

Exercice 2.10 À titre de curiosité, sauriez-vous définir en JavaScript un « compteur » doté de deux méthodes `increment` et `get`, dont l'état interne est inaccessible à l'utilisateur ? ◇ Solution page 96.

L'utilisation de types de fonction en OCaml ou d'interfaces en Java facilite l'écriture de composants **interchangeables**. Si différentes implémentations des piles (voir par exemple les Exercices 2.1 et 2.3) satisfont l'interface `IIntStack`, alors un utilisateur qui s'appuie uniquement sur cette interface peut employer n'importe laquelle de ces implémentations. De même, dans la bibliothèque de Java, les classes `HashSet`, `TreeSet`, et d'autres encore satisfont l'interface `Set`. Donc, un utilisateur qui s'appuie uniquement sur cette interface peut employer n'importe

laquelle de ces implémentations. Cette idée toute simple est parfois appelée **strategy design pattern**. Toutefois, nous constaterons lors de l'Exercice 3.21 que l'interface `Set` et sa parente `Collection` ne sont pas toujours satisfaisantes, en particulier lorsque l'on a besoin d'opérations qui agissent sur **deux** ensembles, par exemple l'opération d'union. Nous proposerons alors une solution fondée sur une interface légèrement différente.

Solution page 96.

Exercice 2.11 (Recommandé) On souhaite écrire en Java, dans une classe `DFS`, un algorithme de parcours en profondeur d'abord pour un graphe orienté. On suppose que les sommets du graphe sont des entiers. On suppose les arêtes du graphes données par une fonction `successors`, de type `Function<Integer,Iterable<Integer>>`, qui est fournie au constructeur de la classe `DFS`.

1. Écrivez le constructeur de la classe `DFS`.
2. Écrivez une méthode `IIntStack newStack ()`, qui produit une nouvelle pile (vide) de sommets. Faites en sorte que cette méthode puisse être redéfinie dans une sous-classe.
3. Écrivez une méthode `Set<Integer> newSet ()`, qui produit un nouvel ensemble (vide) de sommets. Faites en sorte que cette méthode puisse être redéfinie dans une sous-classe.
4. Déclarez une méthode `void process (int node)`, qui devra être définie dans une sous-classe. Elle représente l'action que l'on souhaite effectuer lorsque le sommet `node` est découvert.
5. Écrivez une méthode `void dfs (Iterable<Integer> roots)`, qui effectue le parcours en profondeur d'abord du graphe défini par la fonction `successors` et qui appelle la méthode `process` à chaque fois qu'un nouveau sommet est découvert. ◇

Chapitre 3

Paramétrisation

Nous avons souligné, lors de l'introduction du Chapitre 2, le fait que le mot « abstraction » admet deux sens duaux.

Au sens existentiel, **abstraire, c'est cacher**. Si un composant A publie un nom x mais en cache la définition, alors les utilisateurs de ce composant peuvent faire référence à ce nom, mais ne peuvent pas s'appuyer sur sa définition. Leur code est donc nécessairement indépendant de cette définition : il doit être écrit de sorte à être accepté par le compilateur, quelle que soit cette définition. Le nom x ainsi publié sous forme abstraite est habituellement le nom d'un type, par exemple `stack`, le nom d'un type des piles. Nous avons étudié au Chapitre 2 l'intérêt de cette forme d'abstraction, aussi bien pour l'auteur du composant que pour ses utilisateurs.

Au sens universel, **abstraire, c'est paramétrer**. Si un composant B a besoin de faire référence à un nom x, mais n'a pas besoin de savoir quelle est sa définition, alors B fonctionne quelle que soit cette définition. Il peut donc sembler utile de paramétrer B vis-à-vis de x. On peut alors considérer B non plus comme « un composant », mais comme « une fonction qui à x associe un composant », ou encore comme « une famille de composants » correspondant aux différentes valeurs possibles de x.

Le paramètre x peut être un type, mais aussi une valeur (un entier, une fonction, ...), voire même un autre composant tout entier.

Par exemple, une bibliothèque qui définit un type des listes et offre des opérations sur les listes est typiquement indépendante du type des éléments. Elle peut donc être **paramétrée vis-à-vis du type** des éléments. Ainsi, dans la bibliothèque de Java, la classe `LinkedList` toute entière est paramétrée vis-à-vis du type `E` des éléments. Le type `E` apparaît dans les définitions de méthodes : ainsi, la méthode `set` a pour signature `E set (int index, E element)`. La classe `LinkedList` peut être utilisée pour construire et manipuler des listes d'entiers, dont le type s'écrit `LinkedList<Integer>`, des listes de listes de booléens, dont le type s'écrit `LinkedList<LinkedList<Boolean>>`, etc. D'une manière analogue, dans la bibliothèque d'OCaml, le type des listes, qui s'écrit `'a list`, est paramétré par le type `'a` des éléments. Le type `'a` apparaît dans les types des opérations : ainsi, la fonction `append` a pour type `'a list -> 'a list -> 'a list`. On peut appliquer `append` à des listes d'entiers, dont le type s'écrit `int list`, à des listes de listes de booléens, dont le type s'écrit `(bool list) list` ou simplement `bool list list`, etc. On peut considérer que la fonction `append` admet à la fois le type `int list -> int list -> int list`, le type `bool list -> bool list -> bool list`, etc. On dit que la fonction `append` est **polymorphe**.

Par exemple encore, de nombreux algorithmes de tri exigent que l'on spécifie un ordre sur les éléments de la liste ou du tableau que l'on souhaite trier, mais sont indépendants de la façon dont cet ordre est défini, et sont donc utilisables quel que soit l'ordre choisi. La fonction `sort` qui implémente un tel algorithme peut donc être **paramétrée vis-à-vis d'une valeur**, à savoir une fonction de comparaison `cmp`. Si `sort` est écrite spécifiquement pour trier une liste d'employés, son type s'écrit `(employee -> employee -> int) -> employee list -> employee list` en

OCaml. Ceci signifie que son premier argument est une fonction capable de comparer deux employés, que son second argument est une liste d'employés, et que son résultat est une liste d'employés. Parce que `sort` est une fonction qui attend une fonction, on dit que `sort` est une **fonction d'ordre supérieur**. Naturellement, parce qu'un algorithme de tri standard ignore tout de la structure des employés et se contente de comparer les employés deux à deux à l'aide de la fonction de comparaison qu'on lui a fournie, la fonction `sort` peut être paramétrée aussi vis-à-vis du type des éléments. Elle devient alors une fonction polymorphe, de type `('a -> 'a -> int) -> 'a list -> 'a list`. De façon analogue, en Java, la classe `Collections` offre une méthode statique `sort` dont la signature (légèrement simplifiée) est `static <T> void sort (List<T> list, Comparator<T> c)`. L'annotation `<T>` située en tête est un quantificateur universel. Elle signifie que cette méthode est polymorphe : elle fonctionne pour tout type `T`. La méthode `sort` attend deux arguments, à savoir la liste à trier, `list`, et la fonction de comparaison, `c`. Plus précisément, `c` est un objet de type `Comparator<T>`. Si l'on consulte la définition de l'interface `Comparator`, on constate que cela signifie que l'objet `c` doit être muni d'une méthode `compare` capable de comparer deux éléments de type `T`. En Java 8, on peut employer la syntaxe des clôtures (§1.2.3) pour construire un objet de type `Comparator<T>`.

Les deux paragraphes qui précèdent montrent qu'un composant peut être paramétré vis-à-vis d'un type, ou bien vis-à-vis d'une valeur, ou bien les deux à la fois. Plus généralement, on peut imaginer **paramétrer un composant vis-à-vis d'un autre composant** tout entier. Par exemple, un composant D qui implémente l'algorithme de Dijkstra pour calculer les plus courts chemins dans un graphe a typiquement besoin d'un composant Q qui implémente une file de priorités. Cependant, il n'a pas besoin de savoir comment Q est implémenté. Il existe plusieurs structures de données qui représentent une file de priorités. Il se peut que l'auteur du composant D n'ait pas envie d'en fixer une, mais préfère déléguer ce choix à l'utilisateur de D. Il écrit donc D sous la forme d'une fonction qui, étant donné Q, produit une fonction de calcul des plus courts chemins. Nous verrons au cours de ce chapitre quelle forme concrète prend cette idée en OCaml et en Java. Comme le composant Q définit à la fois un type (le type des files de priorités) et des valeurs (les opérations sur les files de priorités), paramétrer D vis-à-vis de Q revient à paramétrer D à la fois vis-à-vis de types et de valeurs.

En bref, un « composant » logiciel est une unité qui **fournit** un certain nombre de types et de valeurs, et qui **attend** elle-même un certain nombre de types et de valeurs. Chaque composant prend ainsi la forme d'une « pièce de puzzle », et différents composants s'assemblent, les uns fournissant les services dont les autres ont besoin, pour former soit des composants plus complexes, soit (ultimement) une application toute entière.

3.1 Paramétrer vis-à-vis d'un type

3.1.1 Polymorphisme en OCaml

En OCaml, une **variable de types**, c'est-à-dire un type inconnu, s'écrit `'a`, `'b`, etc. La définition d'un type algébrique peut être paramétrée par une variable de types. Par exemple, plutôt que de définir un type des listes immuables dont les éléments sont des entiers (Exercice 1.4) :

```
type list =
| Nil
| Cons of int * list
```

il est préférable de paramétrer ce type vis-à-vis du type `'a` des éléments :

```
type 'a list =
| Nil
| Cons of 'a * 'a list
```

Ainsi, une liste d'entiers admet le type `int list`, une liste de booléens admet le type `bool list`, etc. La définition du type `'a list` se trouve dans la bibliothèque d'OCaml. Toutefois, pour

des raisons de concision, OCaml propose une notation spéciale pour les constructeurs `Nil` et `Cons`. Le premier est noté `[]`, tandis que le second est noté `::`, sous forme infixe. La liste vide s'écrit donc `[]`, et une liste non vide `x :: xs`, où `x` est l'élément situé en tête de la liste et `xs` est le reste de la liste.

La définition d'une abréviation de types (Remarque 1.7) peut elle aussi être paramétrée par une variable de types. Par exemple, supposons que l'on pose la définition suivante :

```
type 'a comparator =
  'a -> 'a -> int
```

Alors, `employee comparator` est le type d'une fonction capable de comparer deux valeurs de type `employee`. Par convention, elle renvoie une valeur négative, nulle, ou positive, suivant que son premier argument est inférieur, égal, ou supérieur à son second argument. On notera l'analogie avec l'interface paramétrée `Comparator<T>` de la bibliothèque de Java.

En OCaml, une fonction peut être polymorphe, c'est-à-dire paramétrée par une ou plusieurs variables de types. Le plus souvent, cela se fait de manière implicite : on écrit le code de la fonction, et le compilateur OCaml **infère** un type polymorphe. Par exemple, la fonction qui calcule la longueur d'une liste s'écrit ainsi :

```
let rec length xs =
  match xs with
  | []      -> 0
  | x :: xs -> 1 + length xs
```

Si on le souhaite, on peut préciser le type de l'argument et du résultat au moment où on définit la fonction :

```
let rec length (xs : 'a list) : int =
  match xs with
  | []      -> 0
  | x :: xs -> 1 + length xs
```

Dans les deux cas, le compilateur infère que la fonction `length` admet le type `'a list -> int`. Dans le fichier list.mli, on déclare donc ainsi l'existence de la fonction `length` :

```
val length: 'a list -> int
```

Il faut comprendre que, **pour tout** `'a`, la fonction `length` admet le type `'a list -> int`. Il y a par convention une quantification universelle implicite.

Les fonctions ne sont pas les seules valeurs polymorphes. Si l'on définit `nil` comme un synonyme pour la liste vide, par exemple :

```
let nil = []
```

alors `nil` n'est pas une fonction, et pourtant `nil` est polymorphe. On pourrait déclarer dans un fichier `.mli` :

```
val nil: 'a list
```

Exercice 3.1 (Recommandé) En OCaml, définissez une fonction `append` telle que, si `xs` et `ys` sont deux listes, alors `append xs ys` est leur concaténation. Quel est le type de cette fonction ?◇ Solution page 98.

Exercice 3.2 En OCaml, définissez une fonction `combine` qui, étant données deux listes de même longueur n, produit une liste de paires de longueur n. Quel est son type ? ◇ Solution page 98.

Exercice 3.3 En OCaml, définissez deux fonctions `take` et `drop` telles que, si `n` est un entier positif ou nul et si `xs` est une liste, alors `take n xs` renvoie la liste des `n` premiers éléments de `xs`, et `drop n xs` renvoie la liste `xs` privée de ses `n` premiers éléments. Dans le cas où `n` est supérieur (ou égal) à la longueur de `xs`, on demande que `take n xs` renvoie `xs` et que `drop n xs` renvoie la liste vide. Quels sont les types de ces fonctions ? ◇ Solution page 99.

Solution page 99. **Exercice 3.4** En OCaml, le type primitif des tableaux, qui s'écrit `'a array`, est paramétré par le type `'a` des éléments. On rappelle que l'accès à un tableau en lecture s'écrit `t.(i)` et que l'accès en écriture s'écrit `t.(i) <- x`. On peut écrire cela, de façon équivalente, sous forme d'appels de fonction, `Array.get t i` et `Array.set t i x`. Par ailleurs, la longueur d'un tableau est donnée par la fonction `Array.length`. Quels sont, selon vous, les types des fonctions `Array.length`, `Array.get` et `Array.set` ? À l'aide de ces fonctions, définissez une fonction `fill` à deux arguments `x` et `a`, telle que `fill x a` ait pour effet d'écrire la valeur `x` dans toutes les cases du tableau `a`. Quel est le type de `fill` ? ◇

3.1.2 Polymorphisme en Java

En Java, une classe peut être paramétrée par une ou plusieurs variables de types. C'est le cas des classes `LinkedList` et `HashMap`, que vous connaissez. Une interface aussi peut être paramétrée. C'est le cas de l'interface `List`, qui est implémentée par la classe `LinkedList`.

Pour illustrer cela, définissons une classe abstraite représentant une liste immuable. Elle est paramétrée par le type `E` des éléments :

```
public abstract class List<E> {
  // The first element of this list, if this list is non-empty.
  public abstract E head () throws NoSuchElementException;
  // The rest of this list, if this list is non-empty.
  public abstract List<E> tail () throws NoSuchElementException;
  // The length of this list.
  public abstract int length ();
  // The concatenation of this list with that list.
  public abstract List<E> append (List<E> that);
}
```

On voit que le type `E` peut apparaître dans les signatures des méthodes. Le fait que `E` ne soit pas un type fixé, mais un paramètre qui pourra être instancié de différentes manières pour donner `List<Integer>`, `List<Boolean>`, etc. ne pose pas en soi de difficulté. Le type `List<E>` lui aussi apparaît dans les signatures des méthodes : en Java, les définitions de classes et d'interfaces (paramétrées ou non) sont récursives et peuvent donc faire référence à elles-mêmes.

Solution page 99. **Exercice 3.5 (Recommandé)** Implémentez deux sous-classes `Nil` et `Cons` de la classe abstraite `List` ci-dessus, de façon à ce qu'une liste immuable soit représentée en mémoire sous forme simplement chaînée. Implémentez les méthodes `head`, `tail`, `length` et `append` dans chacune de ces deux sous-classes. Comment construit-on une liste de longueur 1 contenant l'entier 42 ? ◇

En Java, une méthode peut être polymorphe, c'est-à-dire paramétrée par une ou plusieurs variables de types. (Cela, indépendamment du fait que la classe à laquelle appartient cette méthode soit paramétrée ou non ; et indépendamment du fait que cette méthode soit statique ou non.) Pour en donner un exemple simpliste, considérons une méthode qui implémente la fonction identité : elle renvoie son argument, quel qu'il soit. On peut donc lui passer un `Integer` et obtenir un `Integer` en retour, ou bien lui passer un `Boolean` et obtenir un `Boolean` en retour. Plus généralement, pour tout type `E`, on peut lui passer un argument de type `E` et obtenir un résultat de type `E`. Cette quantification universelle s'exprime en Java en écrivant `<E>` en tête de la déclaration ou définition de la méthode :

```
static <E> E identity (E x) { return x; }
```

La deuxième occurrence de `E` est le type du résultat, et la troisième est le type du paramètre `x`. Une fois la méthode `identity` ainsi définie, on peut l'utiliser pour différentes valeurs de `E`. L'instanciation est implicite : si tout va bien, le compilateur Java infère automatiquement quelle valeur de `E` est utilisée. Par exemple, on peut écrire :

```
System.out.println(identity(0) + 1);
System.out.println(identity(false) && true);
System.out.println(identity("hello") + "world");
```

Ce fragment de code affiche successivement `1`, **`false`**, et `hello world`. On a utilisé ici la méthode `identity` avec trois instances différentes de `E`, à savoir `Integer`, `Boolean`, et `String`.

Détail 3.1 (Types primitifs et polymorphisme) En Java, **une variable de types `E` peut être instanciée** (c'est-à-dire remplacée) **par n'importe quel type** formé à partir d'une **classe** ou d'une **interface**, mais malheureusement **pas par un type primitif**. Les types `List<Object>`, `List<List<Object>>`, `List<Set<Object>>`, etc. sont donc valides, mais le type `List<int>` ne l'est pas. À sa place, il faut écrire `List<Integer>`, qui est valide, car `Integer` n'est pas un type primitif, mais une classe (voir le Détail 3.2 ci-dessous). Les raisons de cette restriction sont historiques et techniques. C#, Scala, ou OCaml, par exemple, n'ont pas cette restriction. Le lecteur qui souhaiterait en savoir plus est renvoyé au livre de Naftalin et Wadler (2006), qui est excellent mais se destine aux experts de Java. ◇

Détail 3.2 (Les types `int` et `Integer`) En Java, comme en C, le type **`int`** est primitif : c'est le type prédéfini des entiers. (Plus précisément, il s'agit des entiers représentables sur 32 bits en complément à deux.) Le type `Integer`, quant à lui, n'est pas primitif, mais défini dans la bibliothèque de Java. La classe `Integer` est définie de manière tout à fait ordinaire comme une classe dotée d'un champ immuable `value` de type **`int`**.

En bref, une valeur de type **`int`** est donc un entier « tout nu ». Il n'occupe que 32 bits et peut être stocké, par exemple, dans une variable locale. Une valeur de type `Integer` est un pointeur vers un objet. Le pointeur lui-même peut être stocké dans une variable locale, mais sa cible est un objet alloué dans le tas.

On peut bien sûr convertir, dans les deux sens, entre ces représentations. Le constructeur de la classe `Integer` permet de convertir de **`int`** vers `Integer`, tandis que la méthode `intValue` de cette même classe permet de convertir en sens inverse.

Ces conversions sont pénibles à écrire. Heureusement, le compilateur Java est capable, la plupart du temps, de les insérer automatiquement là où elles sont nécessaires. On appelle cela l'emballage et déballage automatiques ou « *auto-boxing and unboxing* ».

Par exemple, lorsqu'on a écrit `identity(0) + 1` plus haut, la méthode `identity` a été instanciée en remplaçant `E` par `Integer`, puisqu'on ne peut pas remplacer `E` par **`int`** (Détail 3.1). Cette méthode attend donc un argument de type `Integer`, et renvoie un résultat de type `Integer`. Avant l'appel, l'entier `0` est automatiquement converti de **`int`** vers `Integer`. Au retour, le résultat est automatiquement converti de `Integer` vers **`int`**, afin de permettre l'addition.

Il existe malheureusement quelques situations où la conversion automatique ne se fait pas. C'est le cas des éléments d'un tableau : le compilateur ne convertit pas automatiquement entre les types **`int[]`** et `Integer[]`. Ce n'est pas surprenant : pour effectuer une telle conversion, il faudrait allouer un nouveau tableau, ce qui n'est pas anodin. C'est donc au programmeur que revient cette décision.

Cette dernière remarque explique pourquoi la classe `Arrays` de la bibliothèque propose non seulement une méthode `sort` polymorphe, qui attend un tableau de type `T[]`, mais aussi une version spécialisée qui attend un tableau de type **`int[]`**, une version spécialisée qui attend un tableau de type **`long[]`**, etc. La version polymorphe est applicable à tous les types `T` sauf les types primitifs : aussi on doit définir une version spécialisée pour chaque type primitif. ◇

Détail 3.3 (Création d'un tableau de type `E[]`) Une autre restriction étonnante apparaît lorsqu'on tente de créer un tableau dont les éléments ont le type `E`, où `E` est une variable de types. L'expression **`new`** `E [n]`, qui devrait en principe allouer un tableau de `n` éléments de type `E`, est interdite par le compilateur Java, qui affiche : `error: generic array creation`. Le lecteur qui souhaiterait comprendre les raisons de cette restriction est à nouveau renvoyé à l'excellent mais difficile livre de Naftalin et Wadler (2006).

Malheureusement, ce problème est relativement difficile à contourner. Citons trois solutions possibles.

1. On peut employer la méthode `copyOf` de la classe `Arrays`, qui copie un tableau de type `E[]` existant, et ce pour tout `E`. Malheureusement, cela n'est possible que si on a déjà à disposition un tableau de même type et de même taille que celui que l'on souhaite créer.

2. On peut abandonner l'idée d'utiliser un tableau et employer plutôt les classes `Vector<E>` ou `ArrayList<E>`, qui sont censées représenter une séquence d'éléments. Cependant, aucune de ces classes ne fournit un constructeur permettant de construire initialement une séquence de `n` éléments identiques. Il faut donc construire une séquence vide puis employer une boucle pour y insérer `n` éléments.

3. La méthode `newInstance` de la classe `Array` permet de créer un tableau de type et de taille arbitraires. Malheureusement, elle est un peu difficile d'emploi : elle exige un argument de type `Class<E>`. De plus, le type de son résultat est officiellement `Object`, alors qu'il devrait en toute logique être `E[]`.

Bref, tableaux et polymorphisme ne font pas bon ménage en Java. Il n'y a pas de raison profonde à cela : pour s'en convaincre, il suffit de constater que tableaux et polymorphisme cohabitent parfaitement en OCaml (ainsi que dans d'autres langages). ◇

Solution page 101. **Exercice 3.6 (Recommandé)** Dans une classe `ArrayToHashMap`, écrivez une méthode statique polymorphe `convert` qui, étant donné un tableau, produit une table de hachage au contenu équivalent. On considère ici un tableau et une table de hachage comme équivalents s'ils représentent la même fonction partielle des entiers vers des valeurs. Vous utiliserez la classe `HashMap` de la bibliothèque. ◇

Solution page 101. **Exercice 3.7** Dans une classe `InvertHashMap`, écrivez une méthode statique polymorphe `invert` qui, étant donnée une table de hachage qui représente une certaine fonction partielle f des clefs vers les valeurs, produit une table de hachage qui représente la fonction partielle f^{-1} des valeurs vers les clefs. ◇

3.2 Paramétrer vis-à-vis d'une fonction

Le polymorphisme, illustré précédemment (§3.1), rend un morceau de code plus général, donc plus réutilisable. L'idée est de paramétrer le code par un type, lui permettant ainsi de s'appliquer à plusieurs types différents.

Pour rendre un morceau de code plus général, on peut aussi le **paramétrer par une fonction**, lui permettant ainsi de s'appliquer à plusieurs situations différentes. Les exercices qui suivent illustrent cette idée.

Solution page 101. **Exercice 3.8 (Recommandé)** En OCaml, définissez une fonction `find_zero`, de type `int list -> bool`, qui détermine si une liste d'entiers contient ou non l'entier 0. Définissez ensuite une fonction `find_odd`, de type `int list -> int option`, qui détermine si une liste d'entiers contient ou non un entier impair, et si oui renvoie cet entier. Ces deux fonctions étant très semblables, on souhaite écrire une seule fonction `find` qui détermine si une liste contient ou non un élément qui satisfait une certaine propriété P, et si oui renvoie cet élément. Quel doit être le type de `find` ? Écrivez cette fonction et expliquez comment redéfinir `find_zero` et `find_odd` en termes de `find`. ◇

Solution page 104. **Exercice 3.9 (Recommandé)** On souhaite implémenter en Java l'algorithme MergeSort pour trier un tableau d'entiers. On souhaite que le code soit utilisable non seulement pour trier les éléments du tableau par ordre croissant, mais aussi pour les trier par ordre décroissant, et plus généralement, pour les trier vis-à-vis d'une relation d'ordre arbitraire, choisie par l'utilisateur. L'algorithme sera donc paramétré par un objet `c` de type `Comparator<Integer>`. D'après la

documentation de l'interface `Comparator` de la bibliothèque de Java, un tel objet offre une méthode `int compare (Integer i1, Integer i2)`, qui renvoie un entier négatif si `i1` est inférieur à `i2`, renvoie zéro si si `i1` est égal à `i2`, et renvoie un entier positif si `i1` est supérieur à `i2`.

Placez-vous dans une classe `MergeSortInteger`. Définissez les trois méthodes statiques suivantes. (Vous pouvez les écrire dans l'ordre qui vous convient.)

1. Une méthode `merge`, qui fusionne deux sous-tableaux déjà triés :

   ```
   static void merge (Comparator<Integer> c, int[] B, int[] T,
                                  int lo, int limit, int hi)
   ```

   Cette méthode présuppose que les segments `[lo, limit[` et `[limit, hi[` du tableau `T` contiennent deux séquences déjà triées. Elle écrit alors dans le segment `[lo, hi[` du tableau `B` la fusion triée de ces deux séquences.

2. Une méthode récursive `mergeSort`, qui constitue le cœur de l'algorithme de tri :

   ```
   static void mergeSort (Comparator<Integer> c, int[] B, int[] A,
                                       int lo, int hi, int[] T)
   ```

   Cette méthode lit dans le segment `[lo, hi[` du tableau `A` une séquence, la trie, et écrit la séquence ainsi triée dans le segment `[lo, hi[` du tableau `B`. Elle a le droit d'utiliser le segment `[lo, hi[` du tableau `T` comme espace de travail temporaire.

3. Une méthode non récursive `mergeSort`, qui constitue le point d'entrée de l'algorithme :

   ```
   static void mergeSort (Comparator<Integer> c, int[] B, int[] A)
   ```

   Cette méthode lit la séquence contenue dans le tableau `A`, la trie, et écrit la séquence ainsi triée dans le tableau `B`.

À titre d'illustration, indiquez comment faire appel à cet algorithme lorsque l'on souhaite trier un tableau par ordre décroissant. ◇

Paramétrer un fragment de code vis-à-vis d'un type (§3.1) et vis-à-vis d'une fonction (§3.2) sont deux mécanismes indépendants mais qui naturellement sont souvent utilisés ensemble. Les exercices suivants en donnent quelques illustrations.

Exercice 3.10 (Recommandé) Implémentez en OCaml une fonction `maximum` qui renvoie le maximum d'une liste (éventuellement vide) d'entiers. Paramétrez ensuite cette fonction par des types et/ou des valeurs de façon à la rendre aussi générale que possible. Quel est le type de la fonction ainsi obtenue ? ◇ Solution page 106.

Exercice 3.11 Écrivez en OCaml une fonction `filter` qui, appliquée à une liste d'éléments entiers, renvoie la sous-liste des éléments pairs. Paramétrez ensuite cette fonction par des types et/ou des valeurs de façon à la rendre aussi générale que possible. Quel est le type de la fonction ainsi obtenue ? Répondez ensuite aux même questions à propos d'une fonction `partition` qui, appliquée à une liste d'éléments entiers, renvoie la sous-liste des éléments pairs et la sous-liste des éléments impairs. ◇ Solution page 108.

Exercice 3.12 On souhaite implémenter en OCaml l'algorithme QuickSort de telle manière qu'il soit applicable à un tableau d'éléments de type quelconque et à une relation d'ordre quelconque sur ce type. Quel doit être le type de la fonction `sort` ? Solution page 108.

Pour implémenter QuickSort, on décide d'utiliser une fonction auxiliaire `partition`. Celle-ci est paramétrée par une fonction de comparaison `compare`, un élément `pivot`, un tableau `a`, et trois indices entiers `lt`, `eq`, `gt`, avec `lt <= eq <= gt`. Cette fonction fait l'hypothèse qu'un certain segment `[lo, hi[` du tableau est actuellement découpé en quatre sous-segments consécutifs `[lo, lt[`, `[lt, eq[`, `[eq, gt[`, et `[gt, hi[`, dont le premier contient uniquement des

éléments strictement inférieurs au pivot, le second contient uniquement des éléments égaux au pivot, le troisième contient des éléments indéterminés, et le dernier contient uniquement des éléments strictement supérieurs au pivot. La fonction `partition` déplace les éléments du tableau de façon à ce que le troisième segment, celui des éléments indéterminés, devienne vide. À la fin de l'exécution de `partition`, le segment `[lo, hi[` est donc partitionné en trois sous-segments seulement, à savoir `[lo, lt[`, `[lt, gt[`, et `[gt, hi[`. La fonction `partition` renvoie les valeurs finales des indices `lt` et `gt`.

Dans un premier temps, implémentez la fonction `sort` à l'aide de la fonction `partition`. Dans un second temps, implémentez `partition`. Quel est le type de cette dernière ? ◇

Solution page 109. **Exercice 3.13** Généralisez votre solution (ou bien la solution officielle) de l'Exercice 3.9 de façon à ce que, pour tout type `E`, l'algorithme de tri soit applicable à un tableau de type `E[]`. Vous pourrez utiliser la méthode statique `java.util.Arrays.copyOf` pour créer un nouveau tableau de type `E[]` à partir d'un tableau existant de type `E[]`. ◇

3.3 Paramétrer vis-à-vis d'un type muni d'opérations

Les langages OCaml et Java fournissent tous deux des outils (essentiellement identiques) pour paramétrer un fragment de code vis-à-vis d'un ou plusieurs types (§3.1) et/ou vis-à-vis d'une ou plusieurs opérations (§3.2). Nous avons constaté que ces deux mécanismes sont souvent utilisés de concert.

Lorsqu'un fragment de code doit être paramétré vis-à-vis de plusieurs types et plusieurs opérations, il semble souvent naturel d'organiser ces paramètres en une ou plusieurs **structures** correspondant à des notions abstraites familières. Par exemple, plutôt que de paramétrer une implémentation des arbres binaires de recherche équilibrés séparément par « un type » et par « une relation d'ordre pour ce type », on préférerait la paramétrer par « un type ordonné ». Plutôt que de paramétrer une implémentation des tables de hachage séparément par « un type », « un test d'égalité pour ce type » et « une fonction de hachage pour ce type », on préférerait la paramétrer par « un type hachable ». Dans la solution de l'Exercice 3.10, plutôt que de paramétrer la fonction `maximum` séparément par « un type `'a` », « une opération `join` pour ce type », et « un élément `bottom` de ce type », on pourrait préférer la paramétrer par « un sup-demi-treillis ».

Les exemples ci-dessus mentionnent un type muni d'une ou deux opérations. Souvent, cependant, on souhaite paramétrer un composant vis-à-vis d'un type muni d'un grand nombre d'opérations. Par exemple, on peut souhaiter paramétrer un algorithme de parcours de graphe vis-à-vis d'une implémentation des tables de hachage, c'est-à-dire vis-à-vis d'un type des tables de hachage et vis-à-vis des opérations usuelles sur ce type (création d'une table vide, insertion d'un élément dans une table, recherche d'un élément dans une table, etc.). Nous avons rencontré cela lors de l'Exercice 2.11. De même, on peut souhaiter paramétrer un algorithme de calcul des plus courts chemins vis-à-vis d'une implémentation des files de priorité, munie de toutes les opérations usuelles. Etc.

Pour répondre à ce souhait, OCaml et Java proposent des mécanismes adaptés.

D'abord, les **signatures** d'OCaml et **interfaces** de Java permettent de nommer les différentes sortes de structures qui nous intéressent. Par exemple, la signature `OrderedType` de la bibliothèque d'OCaml et les interfaces `Comparator` et `Comparable` de Java correspondent toutes trois d'une certaine manière à la notion intuitive de « type ordonné », ou « type muni d'une relation d'ordre ». On dispose ainsi d'une façon concise de décrire un groupe de types et d'opérations.

Ensuite, les **foncteurs** d'OCaml permettent de paramétrer un composant d'un seul coup vis-à-vis d'une structure, c'est-à-dire vis-à-vis d'un groupe de types et d'opérations. Par exemple, le foncteur `Map.Make` de la bibliothèque d'OCaml est paramétré par « un type ordonné » `key`, et produit une implémentation des arbres binaires de recherche équilibrés dont les clefs sont de type `key`. Du côté de Java, nous verrons qu'on peut paramétrer une telle implémentation

soit séparément par un type `K` et par un objet de type `Comparator<K>`, soit d'un seul coup par un type `K` contraint à être sous-type de `Comparable<K>`. Cette deuxième forme est appelée **polymorphisme contraint** ou **polymorphisme borné**.

3.3.1 Structures, signatures, et foncteurs en OCaml

En OCaml, on appelle **structure** ou **module** une collection de définitions de types et de valeurs. Par exemple, le type des entiers, muni de sa relation d'ordre habituelle, forme une structure :

```
module I = struct
  type t = int
  let compare (x : t) (y : t) : int =
    if x < y then -1 else if x > y then 1 else 0
end
```

Ceci définit un module `I` doté de deux composantes à savoir un type `I.t`, qui est synonyme de `int`, et une valeur `compare`, dont le type est `I.t -> I.t -> int`.

Détail 3.4 (Syntaxe des structures) Une structure est délimitée par les mots-clef **`struct`** et **`end`**. On lui donne un nom, par exemple `A`, à l'aide de la définition **`module A = struct ... end`**. Si, à l'intérieur de cette structure, on a défini une composante `x`, par exemple en écrivant **`let`** `x = ...` alors, à l'extérieur, cette composante est nommée `A.x`. Si l'on emploie la directive **`open`** `A` alors on peut écrire simplement `x` au lieu de `A.x`.

Par convention, si on place une série de définitions dans un fichier nommé `a.ml` ou `A.ml`, alors OCaml considère que ces définitions forment une structure nommée `A`. Il n'est pas nécessaire dans ce cas d'écrire **`module A = struct ... end`**. ◇

Le type des chaînes de caractères, muni de l'ordre lexicographique, constitue une structure similaire :

```
module S = struct
  type t = string
  let compare (s1 : t) (s2 : t) : int =
    ...
end
```

Exercice 3.14 Implémentez la fonction `compare` du module `S` ci-dessus. La longueur d'une chaîne de caractères est donnée par la fonction `String.length`. L'accès au i-ème caractère d'une chaîne `s` s'écrit `s.[i]` ou bien `String.get s i`. La comparaison de deux caractères est effectuée via la fonction `Char.compare`. ◇ Solution page 112.

Les structures `I` et `S` présentées ci-dessus présentent un point commun : elles définissent toutes deux un type nommé `t` et une fonction nommée `compare` de type `t -> t -> int`. On peut dire informellement que toutes deux représentent un « type ordonné ». Pour expliciter cette notion, on peut définir une **signature** `OrderedType` qui décrit quelles composantes une structure doit offrir pour mériter l'appellation de « type ordonné » :

```
module type OrderedType = sig
  type t
  val compare: t -> t -> int
end
```

En OCaml, on appelle **signature** ou **type de module** une collection de déclarations ou de définitions de types et de déclarations de valeurs. Ainsi, la signature `OrderedType` ci-dessus contient la déclaration d'un type `t` et la déclaration d'une fonction `compare`.

Les structures `I` et `S` ci-dessus définissent toutes deux des types ordonnés : toutes deux **satisfont** la signature `OrderedType`. On peut vérifier cela en modifiant ainsi la définition de `I`, par exemple :

```
module I : OrderedType = struct
  type t = int
  let compare (x : t) (y : t) : int =
    if x < y then -1 else if x > y then 1 else 0
end
```

Vis-à-vis de la définition initiale, on a ajouté l'affirmation `I : OrderedType`. Le compilateur OCaml vérifie alors que la structure `I` définit bien toutes les composantes exigées par la signature `OrderedType`. On peut, de la même manière, affirmer que la structure `S` est conforme à la signature `OrderedType`.

Détail 3.5 (Syntaxe des signatures) Une signature est délimitée par les mots-clef **`sig`** et **`end`**. On lui donne un nom, par exemple `S`, à l'aide de la définition **`module type S = sig ... end`**.

Par convention, si on place une série de déclarations dans un fichier nommé `a.mli` ou `A.mli`, alors OCaml considère que ces déclarations forment une signature. Il n'est pas nécessaire dans ce cas d'écrire **`sig ... end`**. OCaml vérifie alors que la structure définie dans le fichier `A.ml` est conforme à la signature donnée dans le fichier `A.mli`. ◇

La signature `OrderedType`, telle que définie plus haut, exige la présence d'un type `t`, mais n'en donne pas la définition. De ce fait, si on attribue à la structure `I` la signature `OrderedType`, comme plus haut :

```
module I : OrderedType = struct type t = int ... end
```

alors OCaml considère que le type `I.t` est abstrait. Le fait que ce type est en fait égal à `int` est connu à l'intérieur de la définition de `I`, mais pas à l'extérieur. Ce comportement est conforme à ce que l'on obtiendrait si l'on plaçait la définition **`type t = int`** dans un fichier `I.ml` et la déclaration **`type t`** dans un fichier `I.mli`.

Si l'on souhaite que l'égalité entre `I.t` et `int` soit connue à l'extérieur, alors il faut placer dans la signature non pas la déclaration **`type t`**, mais la définition **`type t = int`**. On peut écrire cela soit de façon explicite, comme ceci :

```
module I : sig
  type t = int
  val compare: t -> t -> int
end =
struct
  type t = int
  ...
end
```

soit, sous forme plus concise et plus claire :

```
module I : (OrderedType with type t = int) =
struct
  type t = int
  ...
end
```

On affirme ainsi que `I` est une structure de la forme « type ordonné », dont la composante `t` est égale à `int`.

Détail 3.6 (Syntaxe de `with type`) Si `S` est une signature qui déclare un type `t` sans en donner la définition, alors la signature **`S with type t = foo`** est une copie de la signature `S` où la déclaration **`type t`** a été remplacée par la définition **`type t = foo`**. ◇

Quel est l'intérêt de définir une signature, par exemple `OrderedType`, et de noter que certaines structures, par exemple `I` et `S` ci-dessus, satisfont cette signature ? Cela ne devient réellement utile qu'à partir du moment où on peut écrire du code qui fonctionne **pour tout**

type ordonné. La bibliothèque Map, par exemple, définit des tables d'association représentées par des arbres binaires de recherche équilibrés. Elle est applicable à n'importe quel type de clefs muni d'une relation d'ordre. En d'autres termes, elle est applicable à n'importe quelle structure qui satisfait la signature `OrderedType`. Nous pouvons l'appliquer à notre structure `I` pour obtenir des tables d'association dont les clefs sont des entiers ; l'appliquer à notre structure `S` pour obtenir des tables d'association dont les clefs sont des chaînes de caractères ; etc.

Voyons plus précisément quel mécanisme est utilisé ici.

Pour qu'un fragment de code soit applicable à n'importe quel type ordonné, on doit en faire un **foncteur**, c'est-à-dire une fonction dont les arguments et le résultat sont des structures. Ici, il faut écrire un foncteur dont l'argument `X` est décrit par la signature `OrderedType`. La définition d'un tel foncteur `F` prend la forme suivante :

```
module F (X : OrderedType) = struct
  ...
end
```

Un foncteur a en général plusieurs arguments, qui sont des structures, et un résultat, qui est lui aussi une structure. Pour donner un exemple, considérons la construction de l'ordre lexicographique sur les paires. On sait que, si deux types `t1` et `t2` sont munis d'une structure de type ordonné, alors le type produit `t1 * t2` peut lui aussi être considéré comme un type ordonné, pour la relation d'ordre lexicographique. Quels que soient `t1` et `t2` et quelles que soient les relations d'ordre choisies pour ces types, cette construction est la même. On peut donc l'écrire, une fois pour toutes, sous forme d'un foncteur.

```
module Lexico (T1 : OrderedType) (T2 : OrderedType) = struct
  type t = T1.t * T2.t
  let compare (x1, x2) (y1, y2) =
    let c1 = T1.compare x1 y1 in
    if c1 <> 0 then c1
    else T2.compare x2 y2
end
```

Ce foncteur a deux arguments, `T1` et `T2`, qui tous deux sont des structures et satisfont la signature `OrderedType`. Dans le corps du foncteur `Lexico`, on sait donc qu'il existe deux types `T1.t` et `T2.t` et deux fonctions `T1.compare` et `T2.compare` de types respectifs `T1.t -> T1.t -> int` et `T2.t -> T2.t -> int`. Bien sûr, nous ne pouvons pas connaître la définition des types `T1.t` et `T2.t`. En effet, celle-ci ne sera fixée que le jour où le foncteur `Lexico` sera appliqué.

Le foncteur `Lexico` produit lui-même une structure qui contient la définition d'un type `t` et d'une fonction `compare`. Le premier est défini comme le produit des types `T1.t` et `T2.t`. La seconde est définie comme la combinaison lexicographique des ordres `T1.compare` et `T2.compare`.

Quelle est la signature de la structure produite par le foncteur `Lexico` ? On peut bien sûr l'écrire sous la forme explicite suivante :

```
sig
  type t = T1.t * T2.t
  val compare: (T1.t * T2.t) -> (T1.t * T2.t) -> int
end
```

ou encore, sous une forme équivalente plus concise, toujours explicite :

```
sig
  type t = T1.t * T2.t
  val compare: t -> t -> int
end
```

ou enfin, en tirant parti de la construction **`with type`** (Détail 3.6) :

```
OrderedType with type t = T1.t * T2.t
```

En résumé, si dans un fichier `.ml` on a placé le code du foncteur `Lexico` ci-dessus, alors dans le fichier `.mli` on a pourra déclarer ceci :

```
module Lexico (T1 : OrderedType) (T2 : OrderedType) :
  OrderedType with type t = T1.t * T2.t
```

On déclare ainsi que le foncteur `Lexico` attend deux types ordonnés `T1` et `T2` et produit un type ordonné dont la composante `t` est le type produit `T1.t * T2.t`.

Pour utiliser le foncteur `Lexico`, il faut l'appliquer à deux structures, par exemple aux structures `I` et `S` définies plus haut.

```
module  I = struct type t = int    ... end
module  S = struct type t = string ... end
module IS = Lexico(I)(S)
```

La signature de la structure `IS`, calculée automatiquement par le compilateur OCaml, est `OrderedType with type t = I.t * S.t`. Parce que les équations `I.t = int` et `S.t = string` sont connues, cette signature est équivalente à `OrderedType with type t = int * string`. La fonction `IS.compare` peut donc être utilisée pour comparer des paires d'un entier et d'une chaîne de caractères.

La bibliothèque Map, déjà mentionnée plus haut, définit elle aussi un foncteur, nommé `Make` ou (de l'extérieur) `Map.Make`. La documentation donne le type de ce foncteur, telle qu'on la trouve dans le fichier `map.mli` :

```
module Make (Key : OrderedType) :
  sig
    type key = Key.t
    type 'a t
    val empty: 'a t
    val add: key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a
    ...
  end
```

Ce type indique que le foncteur `Map.Make` est applicable à toute structure `Key` de signature `OrderedType` et produit une structure dans laquelle on trouve un type `key`, un type `t`, et des valeurs `empty`, `add`, `find`, etc. Le type `key` est concret : il est égal à `Key.t`. Le type `t` est abstrait : on ne révèle pas comment les tables d'association sont implémentées.

Le type `t` s'écrit en fait `'a t`, car il est paramétré par le type `'a` des éléments que l'on stocke dans la table d'association. Une valeur de type `int t` est donc une table qui à des clefs associe à des entiers ; une valeur de type `bool t` est une table qui à des clefs associe des Booléens ; etc. Les valeurs `empty`, `add`, `find`, etc. sont polymorphes : elles fonctionnent pour toute valeur de `'a`.

Une certaine dissymétrie entre clefs et valeurs semble apparaître, puisque le type des clefs est fixé au moment où on applique le foncteur `Map.Make` à un argument, tandis que le type des valeurs n'est jamais fixé. On aurait pu éviter cette dissymétrie en présentant `Map.Make` comme un foncteur à deux arguments, deux structures `Key` et `Value`. La signature de `Value` aurait été simplement `sig type t end`, car on n'a pas besoin de relation d'ordre sur les valeurs. On a préféré la version où `Map.Make` n'a qu'un seul argument `Key` parce que cette version est plus agréable à l'emploi.

Si on a besoin de tables dont les clefs sont des paires d'un entier et d'une chaîne de caractères, alors on applique le foncteur `Map.Make` à la structure `IS` que nous avons définie plus haut, qui munit le type `int * string` d'une structure de type ordonné :

```
module ISMap = Map.Make(IS)
```

Nous pouvons ensuite utiliser la structure `ISMap` :

```
let m = ISMap.add (42, "quarante-deux") true ISMap.empty
```

La valeur `m` ainsi définie est une table d'association de type `bool ISMap.t`.

L'application du foncteur `Map.Make` à la structure `IS` a produit une structure, que nous avons nommée `ISMap`, qui contient (entre autres) un type abstrait `'a ISMap.t`. Il faut souligner que, si nous appliquions le foncteur `Map.Make` à une autre structure, nous obtiendrions un autre type abstrait. En général, **appliquer un même foncteur à deux structures distinctes produit deux types abstraits distincts**. Ce phénomène est connu sous le nom de **générativité**. L'Exercice 3.18 souligne pourquoi c'est essentiel.

Exercice 3.15 Au lieu d'écrire `Lexico` sous forme d'un foncteur, comme nous l'avons fait plus haut, on peut écrire `lexico` sous forme d'une fonction d'ordre supérieur, dont le type est `'a comparator -> 'b comparator -> ('a * 'b) comparator`. Faites-le. ◇ Solution page 112.

Exercice 3.16 (Recommandé) Écrivez un foncteur qui, étant donné un type `t` muni d'une relation d'ordre, munit le type `t list` de la relation d'ordre lexicographique. Si l'on place le code de ce foncteur dans un fichier `.ml`, que peut-on ou doit-on déclarer dans le fichier `.mli` correspondant ? ◇ Solution page 112.

Exercice 3.17 (Recommandé) Définissez une signature `Queue` que doit (ou peut) avoir une structure de files d'attente, c'est-à-dire une structure proposant un type des files d'attente ainsi que des opérations de création d'une nouvelle file, d'insertion d'un élément, et de retrait d'un élément. ◇ Solution page 113.

Exercice 3.18 Supposons que l'on applique le foncteur `Map.Make` séparément à deux structures qui définissent respectivement l'ordre croissant et l'ordre décroissant sur les entiers : Solution page 114.

```
module M1 = Map.Make(struct type t = int let compare x y = x - y end)
module M2 = Map.Make(struct type t = int let compare x y = y - x end)
```

Les modules `M1` et `M2` admettent la même signature, à savoir `Map.S with type key = int`. Cependant, les types `'a M1.t` et `'a M2.t` sont considérés comme distincts. Expliquez pourquoi cela est souhaitable. On signale que, dans l'implémentation du foncteur `Map.Make`, qui se trouve dans le fichier map.ml de la bibliothèque, le type `'a t` est défini comme un type algébrique qui représente des arbres binaires de recherche équilibrés. ◇

3.3.2 Signatures paramétrées et polymorphisme contraint en Java

On peut établir une certaine analogie entre les structures et signatures d'OCaml, d'une part, et certains concepts de Java, d'autre part.

En Java, une **classe** `A` définit un type, également nommé `A`, accompagné d'un certain nombre d'opérations, représentées par les méthodes de la classe `A`. Dans une certaine mesure, cette notion est analogue à la notion de structure OCaml, qui permet également de définir un type accompagné d'opérations. Par exemple, la classe `HashMap` pourrait être définie ainsi :

```
public class HashMap<K,V> {
  // Constructor.
  public HashMap() { ... }
  // Insertion.
  public V put (K key, V value) { ... }
  // Etc.
}
```

En Java, une classe est un type. Par exemple, une table de hachage des entiers vers les entiers a le type `HashMap<Integer,Integer>`.

En Java toujours, définir une **interface** `I` permet d'attribuer un nom conventionnel à un groupe d'opérations. Dans une certaine mesure, cette notion est analogue à la notion de signature OCaml. Par exemple, la notion de table d'association est reflétée par l'interface `Map`, qui est définie ainsi :

```
public interface Map<K,V> {
  // Insertion.
  V put (K key, V value) { ... }
  // Etc.
}
```

En Java, une interface est aussi un type. Si un objet admet le type `Map<Integer,Integer>`, alors on sait qu'il a une méthode `put`, même si on ne sait pas à quelle classe il appartient. En bref, on sait que cet objet est une table d'association, même si on ne sait pas comment cette table est implémentée.

En OCaml, le compilateur est capable de reconnaître a posteriori qu'une certaine structure satisfait une certaine signature, même si cela n'a pas été déclaré a priori. En Java, le fait que la classe `HashMap` satisfait l'interface `Map` doit être déclaré au moment où la classe est définie à l'aide d'une clause **`implements`** :

```
public class HashMap<K,V> implements Map<K,V> {
  ...
}
```

Il existe une similarité entre la signature `OrderedType` d'OCaml, qui décrit un type `t` dont on peut comparer les éléments grâce à une fonction `compare`, et les interfaces paramétrées `Comparator` et `Comparable` de Java. Mais pourquoi la bibliothèque de Java propose-t-elle deux interfaces qui correspondent intuitivement à une même notion, à savoir la notion de « type ordonné » ? Ces interfaces diffèrent quant au choix de l'objet où réside la capacité à effectuer une comparaison.

```
public interface Comparator<T> {
  int compare (T o1, T o2);
}
```

Dans la première approche, un objet distingué, le « comparateur », possède une méthode `compare` à deux arguments, donc est capable de comparer deux éléments de type `T`. Cet objet est analogue à une fonction OCaml de type `t -> t -> int`. D'ailleurs, en Java 8, on peut employer la syntaxe des clôtures (§1.2.3) pour construire un comparateur. Par exemple, `(x, y) -> x < y ? -1 : x == y ? 0 : 1` est un objet de type `Comparator<Integer>` qui représente l'ordre croissant sur les entiers. De même, il est facile de construire un comparateur qui représente l'ordre décroissant sur les entiers. Pour un type `T` fixé, il peut donc exister plusieurs objets de type `Comparator<T>`, qui représentent différents ordres. Un algorithme de tri peut être paramétré par un comparateur `c` de type `Comparator<T>` de façon à pouvoir trier suivant différents ordres (Exercice 3.9).

```
public interface Comparable<T> {
  int compareTo (T that);
}
```

Dans la seconde approche, au lieu de confier la responsabilité de comparer deux éléments de type `T` à un objet « comparateur » distingué, on préfère que les éléments eux-mêmes soient capables de se comparer les uns aux autres. On souhaite donc que chaque élément de type `T` soit doté d'une méthode `compareTo`. Cette méthode n'a qu'un seul argument explicite, nommé `that` ci-dessus. Naturellement, elle a aussi un argument implicite, **`this`**. Sa mission est de comparer les objets **`this`** et `that`.

L'interface `Comparable` est paramétrée par un type `T`, qui est le type de l'argument `that`. C'est inévitable : si l'interface n'était pas paramétrée, alors il faudrait donner à `that` un type fixé. On pourrait lui donner par exemple le type `Object`, mais cela ne serait pas satisfaisant. En effet, cela reviendrait à exiger qu'un objet de type `T` soit capable de se comparer à un objet de type arbitraire. Or, on veut bien exiger qu'une pomme sache se comparer à une autre pomme, et une orange à une autre orange, mais mieux vaut ne pas exiger qu'une pomme sache se comparer à une orange ; cela n'a guère de sens. (Voir, à ce propos, le Détail 3.7 plus bas.) Une

autre idée serait de déclarer que `that` a « le même type que **`this`** ». C'est ce que l'on appelle parfois un « *self type* ». Cependant, Java ne permet pas cela.

Parce que l'interface `Comparable` est paramétrée, on voit apparaître un phénomène un peu étrange. La classe `Integer`, qui possède une méthode `int compareTo (Integer that)`, satisfait l'interface `Comparable<Integer>`. De même, la classe `Boolean` satisfait l'interface `Comparable<Boolean>`. De façon générale, une classe `C` dont les éléments sont capables de se comparer les uns aux autres implémente l'interface `Comparable<C>`. Il y a là une forme de récursivité un peu inhabituelle : le nom de la classe `C` apparaît dans une interface que la classe `C` implémente.

Si l'on souhaite qu'un algorithme de tri soit applicable à tout type `T` dont les éléments sont capables de se comparer les uns aux autres, alors il faut déclarer que cet algorithme fonctionne pour tout type `T` qui implémente `Comparable<T>`. On parle alors de **polymorphisme contraint**, car cet algorithme ne fonctionne pas pour tout type `T`, mais seulement pour tout type `T` qui satisfait la contrainte « `T` implémente `Comparable<T>` ». Dans la syntaxe de Java, au lieu d'écrire `<T>` pour indiquer que la méthode `sort` est polymorphe, par exemple comme ceci :

```
static <T> void sort (List<T> list) { ... }
```

on écrit `<T extends Comparable<T>>`, comme ceci :

```
static <T extends Comparable<T>> void sort (List<T> list) { ... }
```

Le mot-clef **`extends`** semble ici mal choisi, car `Comparable` est une interface et on attendrait plutôt ici le mot-clef **`implements`**. Cependant, c'est ainsi ; il faut écrire **`extends`** lorsqu'on écrit une contrainte que le type `T` doit respecter.

La classe `Collections` propose deux méthodes statiques nommées `sort` qui illustrent les deux approches présentées ci-dessus. La première est polymorphe vis-à-vis de `T` et exige un argument de type `Comparator<T>`. La seconde est polymorphe vis-à-vis d'un type `T` qui implémente `Comparable<T>`. (Voir, à ce sujet, le Détail 3.8.)

La seconde approche, fondée sur `Comparable<T>`, peut sembler offrir un avantage en termes de légèreté. Puisque les éléments « savent » se comparer les uns aux autres, pas besoin de comparateur : la méthode `sort` ne prend qu'un argument, à savoir la liste à trier, et non deux. L'inconvénient de cette approche, toutefois, est qu'on peut ainsi associer à chaque type `T` **au plus une** relation d'ordre. La documentation de Java fait référence à cet ordre privilégié sous le nom d'ordre naturel. Si on souhaite pouvoir trier une liste d'entiers tantôt par ordre croissant, tantôt par ordre décroissant, alors cette seconde approche ne convient pas. Il faut revenir à la première approche et employer explicitement deux comparateurs distincts pour un même type `T`.

Détail 3.7 (La méthode `equals`) On remarquera que la méthode `equals` de la classe `Object` a pour signature `boolean equals (Object obj)`. Par conséquent, une comparaison entre deux objets de nature différente, par exemple `new Boolean (true).equals(new Integer (1))`, est permise, et produit la valeur **`false`**. De ce fait, certaines erreurs de programmation risquent d'être détectées très tardivement. On peut argumenter qu'il aurait mieux valu ne pas doter la classe `Object` d'une méthode `equals`. ◇

Détail 3.8 (Polymorphisme contraint et wildcards) En réalité, la méthode `sort` de la classe `Collections` n'est pas déclarée ainsi :

```
static <T extends Comparable<T>> void sort(List<T> list)
```

Elle est déclarée sous la forme suivante :

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

Cette seconde forme exprime une contrainte légèrement plus souple, ce qui rend la méthode `sort` applicable dans des situations plus nombreuses. En effet, la première forme exige que

le type `T` soit **sous-type** du type `Comparable<T>`, ce que l'on pourrait résumer en termes mathématiques par une contrainte de sous-typage entre ces deux types :

$$T \leq \mathit{Comparable}(T)$$

La seconde forme exige que `T` soit sous-type d'un certain type, noté `?`, qui lui-même doit être **super-type** de `T`. On pourrait résumer cela en termes mathématiques par une conjonction de deux contraintes de sous-typage, où U est inconnu :

$$\exists U. \left(\begin{array}{l} T \leq \mathit{Comparable}(U) \\ T \leq U \end{array} \right)$$

Si les contraintes ci-dessus sont satisfaites et si `x` et `y` ont le type T, alors l'appel `x.compareTo(y)` est bien typé. En effet, `x` a le type T, donc (d'après la première contrainte) a aussi le type $\mathit{Comparable}(U)$. De plus, `y` a le type T, donc (d'après la deuxième contrainte) a aussi le type U. Puisque `x` et `y` ont respectivement les types $\mathit{Comparable}(U)$ et U, l'appel `x.compareTo(y)` est valide.

Il en découle que, si les fruits sont comparables entre eux (i.e., la classe `Fruit` implémente l'interface `Comparable<Fruit>`) et si les pommes sont des fruits (i.e., `Apple` est une sous-classe de `Fruit`), alors la méthode `sort` est capable de trier une liste de pommes, et cela, bien que la classe `Apple` n'implémente pas l'interface `Comparable<Apple>`.

Le symbole `?` est appelé **wildcard**. Il représente un type inconnu, éventuellement soumis à certaines contraintes, par exemple la contrainte `? super T`, que nous avons traduite par $T \leq U$ ci-dessus. À mon avis, il s'agit d'un aspect relativement obscur et inélégant de Java. Je n'y reviendrai pas. ◇

Solution page 114.

Exercice 3.19 Supposons donnée une interface `IStack` qui décrit une pile. Cette interface est paramétrée par le type `E` des éléments. Elle généralise donc l'interface `IIntStack` du chapitre précédent (§2.2) :

```
public interface IStack<E> {
  void push (E x);
  E pop () throws NoSuchElementException;
}
```

Cette interface minimaliste ne propose que deux opérations. On peut souhaiter disposer d'autres opérations, par exemple `size`, qui renvoie le nombre actuel d'éléments de la pile. On définit donc une interface plus riche :

```
public interface IRichStack<E> extends IStack<E> {
  int size ();
}
```

On a utilisé l'**héritage** entre interfaces pour éviter de répéter la déclaration des méthodes `push` et `pop`. On souhaite maintenant se doter d'une façon systématique de transformer une implémentation de `IStack` en une implémentation de `IRichStack`. Écrivez donc une méthode statique `enrich` qui, étant donnée une fonction de type `Supplier<IStack<E>>`, produit une fonction de type `Supplier<IRichStack<E>>`. ◇

Solution page 115.

Exercice 3.20 (Recommandé) On se place pour cet exercice dans une hypothétique version de Java où la classe `Object` ne définit pas les méthodes `equals` et `hashCode`. De ce fait, la classe `HashMap<K, V>` ne peut pas être utilisée avec n'importe quel type `K` des clefs : elle n'a de sens que pour les types `K` qui sont munis de méthodes `equals` et `hashCode`. Définissez une interface `Hashable` (paramétrée si besoin) qui décrit ces méthodes. Indiquez ensuite comment présenter la classe `HashMap` pour qu'elle ne s'applique qu'aux types de clefs « hachables ». Donnez pour cela la première ligne de la définition de la classe `HashMap`. ◇

Solution page 116.

Exercice 3.21 L'interface `Collection<E>` de la bibliothèque de Java décrit un objet censé représenter une collection (modifiable) d'éléments de type `E`. En voici une version simplifiée :

```
public interface Collection<E> {
  boolean add (E e);
  boolean contains (E e);
  Iterator<E> iterator();
  boolean addAll (Collection<E> c);
}
```

Cette interface très générale est implémentée par de nombreuses structures de données : listes chaînées, tables de hachage, arbres binaires de recherche, etc.

Cette interface peut être critiquée à propos d'un point : sa méthode `addAll` ne peut être implémentée que de façon naïve et (par conséquent) potentiellement inefficace. Expliquez pourquoi : comment peut-on implémenter `addAll`, et quelle est sa complexité asymptotique ? Pourquoi cela peut-il être insatisfaisant ?

On souhaite écrire un algorithme qui exige une implémentation des ensembles d'éléments de type `E`. Cet algorithme exige bien sûr la possibilité de créer un nouvel ensemble vide. Il exige de plus la présence des méthodes `add`, `contains`, et `addAll`. On souhaite laisser à l'utilisateur le choix de cette structure, et on souhaite lui laisser la liberté de choisir une structure dotée d'une opération d'union (`addAll`) très efficace. Comment faut-il paramétrer l'algorithme pour permettre cela ? ◇

Exercice 3.22 (Quiz ; difficile) La classe `TreeMap<K,V>` de la bibliothèque de Java implémente une table d'association sous forme d'un arbre binaire de recherche. Par conséquent, elle a besoin que le type `K` des clefs soit doté d'une relation d'ordre. Cette relation est déterminée au moment où la table est construite. Si l'utilisateur emploie le constructeur suivant : Solution page 117.

```
public TreeMap (Comparator<? super K> comparator)
```

alors, clairement, l'ordre est déterminé par le comparateur `comparator`, choisi par l'utilisateur. Cependant, la classe `TreeMap` offre d'autres constructeurs, dont celui-ci :

```
public TreeMap ()
```

Celui-ci semble mystérieux. D'une part, aucun comparateur n'est exigé. D'autre part, on n'exige pas non plus `K extends Comparable<K>`, donc semble-t-il, rien ne garantit que les clefs sont capables de se comparer les unes aux autres. Comprenez-vous quelle relation d'ordre est employée dans ce cas ? Quels sont les risques inhérents à cette approche ? ◇

Chapitre 4

Itération

Chacun sait que l'**itération**, comprise comme la capacité à effectuer une tâche de façon répétée, est un trait fondamental de nos langages de programmation et de nos machines. Elle leur permet d'exprimer et d'exécuter n'importe quel algorithme : en termes théoriques, elle leur confère la Turing-complétude. En termes plus pratiques, la très grande vitesse des machines modernes rend cette capacité à effectuer des calculs itératifs particulièrement impressionnante et utile.

En quoi l'itération mérite-t-elle notre intérêt dans ce cours ? N'est-elle pas très simple ? Après tout, il suffit d'écrire en Java :

```
for (int i = 0; i < n; i++) s += a[i];
```

pour calculer la somme des éléments d'un tableau ; et il suffit d'écrire en OCaml :

```
let rec fact n = if n = 0 then 1 else n * fact (n - 1)
```

pour définir (une version de) la fonction factorielle. Les boucles (`while`, `for`, etc.) comme les fonctions récursives sont des concepts supposés connus dans ce cours. Que reste-t-il donc à discuter ?

Le problème qui nous intéresse ici est celui du **découpage modulaire** d'un calcul itératif pour isoler d'une part un **producteur** et d'autre part un **consommateur**.

Un tel découpage est souvent **naturel**, parce que la question de « comment produire » une certaine suite d'éléments est indépendante de la question de « quoi faire avec » cette suite d'éléments. Il multiplie les manières dont on peut assembler ou composer différents producteurs et consommateurs.

De plus, un tel découpage est parfois **imposé** par une barrière d'abstraction existante. Si, par exemple, on a implémenté des ensembles à l'aide d'arbres binaires de recherche équilibrés, et si on a présenté le type des ensembles comme un type abstrait, comment un utilisateur peut-il énumérer les éléments d'un ensemble ? Il n'a pas accès à la représentation interne des ensembles, donc ne peut pas écrire (par exemple) une fonction récursive qui parcourt un arbre. Il faut par conséquent que la bibliothèque qui implémente les ensembles propose elle-même une ou plusieurs opérations permettant d'itérer sur un ensemble. Cependant, l'auteur de la bibliothèque ne sait pas ce que l'utilisateur souhaite faire avec chaque élément. La bibliothèque est producteur, l'utilisateur est consommateur, et chacun ignore tout de l'autre.

Une solution à ce problème serait que producteur et consommateur s'accordent sur le choix d'une structure de données concrète pour représenter en mémoire une suite d'éléments. Une liste simplement chaînée (ou bien un tableau) serait un choix naturel. Le producteur construirait à la demande cette liste, que l'utilisateur pourrait ensuite exploiter à sa guise. Ainsi, la bibliothèque d'ensembles évoquée ci-dessus offrirait une fonction `elements` qui produirait la liste des éléments d'un ensemble. Une telle approche, cependant, est trop naïve. Ses défauts sont nombreux :

1. Il faut un espace $\Omega(n)$ en mémoire pour construire la liste des n éléments. Or, si le consommateur n'a besoin que d'un élément à la fois, on pourrait espérer que l'itération se fasse en espace $O(1)$.

2. Il faut un temps $\Omega(n)$ pour construire la liste des n éléments. Or, si le consommateur n'a en fait besoin que des k premiers éléments, on pourrait espérer que l'itération se fasse en temps $O(k)$. Ce ne sera pas toujours possible : comme le montre l'Exercice 4.15, un itérateur sur un arbre équilibré exige un temps $\Omega(\log n)$ pour produire le premier élément. Néanmoins, on peut espérer faire mieux que l'approche naïve.

3. Même si le consommateur a effectivement besoin des n éléments, auquel cas il est normal que l'itération demande au total un temps $\Omega(n)$, on pourrait espérer que l'itération se fasse en **temps réel**, c'est-à-dire que le k-ième élément soit disponible au bout d'un temps $O(k)$, par exemple. Or, ce n'est pas le cas : il faut déjà un temps $\Omega(n)$ pour obtenir le premier élément.

4. Enfin, si la suite construite par le producteur est conceptuellement infinie (par exemple, s'il s'agit de la suite des nombres premiers), alors cette approche est inapplicable. Pourtant, produire une suite infinie a bien un sens, pourvu que le consommateur n'en exploite qu'une partie finie.

Dans cette approche naïve, c'est d'abord le producteur qui travaille à produire les éléments, tandis que le consommateur est inactif ; puis le consommateur travaille et traite les éléments, tandis que le producteur a terminé et est donc inactif à son tour. Pour éviter les défauts ci-dessus et proposer une solution satisfaisante au problème, il semble nécessaire que **producteur et consommateur travaillent tour à tour**, l'un produisant un élément, l'autre consommant cet élément, et ainsi de suite. Or, pour cela, il est nécessaire de mettre en place une forme de **dialogue** entre producteur et consommateur, chacun devant interrompre son exécution et signaler à l'autre qu'il doit reprendre la sienne.

À l'aide de quels mécanismes et concepts existants peut-on autoriser ce dialogue entre producteur et consommateur ? La notion de **fonction** semble pouvoir offrir l'outil nécessaire : en effet, appeler une fonction ou (symétriquement) mettre fin à l'exécution d'une fonction sont deux façons de transmettre le contrôle du producteur au consommateur ou vice-versa, et de transmettre en même temps (si besoin) un élément. Dans ce chapitre, nous étudions plusieurs solutions basées soit sur la notion de fonction, soit sur les notions (proches) d'objet et de **suspension**.

Il existe d'autres solutions encore, qui s'appuient par exemple sur les « coroutines », ou bien sur les « *threads* » et les « canaux », etc. Nous ne les étudierons pas ici. Nous n'étudierons pas non plus comment on pourrait accélérer l'itération en exploitant plusieurs producteurs et/ou plusieurs consommateurs répartis sur des cœurs ou sur des machines distincts. Je mentionne toutefois ces pistes pour suggérer à quel point le problème apparemment simple de l'itération mène en réalité à des questions profondes et complexes.

Une fois admise l'idée que producteur et consommateur doivent dialoguer via des appels (et retours) de fonctions, il reste à répondre à une question simple : lequel des deux contrôle l'autre ? Lequel joue le rôle de l'appelant, lequel joue le rôle de l'appelé ? Si **le producteur a le contrôle**, alors il appelle le consommateur à chaque fois qu'un élément est disponible : « tiens, voilà un élément ; traite-le et rends-moi la main ». Si **le consommateur a le contrôle**, alors il appelle le producteur à chaque fois qu'il a besoin d'un élément : « s'il te plaît, trouve le prochain élément et donne-le moi ».

Si l'on choisit de donner le contrôle au producteur, on est amené à écrire celui-ci sous forme d'une fonction, traditionnellement appelée « ***fold*** », qui effectue l'itération et soumet chaque élément au consommateur. Ce dernier est alors représenté par une fonction `f`, censée traiter un élément, qui doit être passée en argument à `fold`. Une fonction `fold` est donc une **fonction d'ordre supérieur** (§3.2). La fonction « *fold* » et ses variantes sont également connues sous d'autres noms traditionnels, par exemple « *reduce* » en Lisp, en Python, et dans la bibliothèque stream de Java 8. Nous les appellerons des **réducteurs** (§4.1).

Si l'on choisit au contraire de donner le contrôle au consommateur, on est amené à écrire le producteur sous forme d'une fonction (ou d'un objet) qui, lorsqu'elle est appelée par le consommateur, produit un élément (s'il en reste). Le producteur se présente donc sous la forme d'un **itérateur**, c'est-à-dire une entité capable de produire un élément à la demande. Il existe deux variantes de cette approche, selon que l'itérateur est modifiable ou non. S'il est modifiable, chaque appel à l'itérateur produit un nouvel élément (s'il en reste), et rien d'autre. Ces **itérateurs modifiables** (§4.2) sont ceux que l'on trouve en Java sous le nom d'`Iterator`. Si l'itérateur n'est pas modifiable, chaque appel à itérateur produit une paire d'un nouvel élément et d'un nouvel itérateur, qui donne accès aux éléments suivants. Nous parlerons alors d'**itérateurs immuables** ou bien, lorsqu'on leur ajoute une forme de mémoisation, de **flots** (§4.3) ou « ***streams*** ». Les flots jouent un rôle fondamental dans les langages de programmation dits « paresseux », comme Haskell, où ils sont connus tout simplement sous le nom de « listes ».

4.1 Réducteurs

De façon générale, nous utilisons le mot **réducteur** pour désigner un producteur paramétré par un consommateur. Ce terme est tiré du mot anglais « *reduce* », qui est l'un des noms traditionnellement associés à cette notion ; nous expliquerons un peu plus loin l'origine de ce mot.

4.1.1 Sans accumulateur explicite

Prenons l'exemple simple d'un producteur chargé de produire la suite des entiers compris entre `i` au sens large et `j` au sens strict. Ce producteur doit avoir accès d'une part aux entiers `i` et `j`, d'autre part au consommateur. Ce dernier est représenté par une fonction ou un objet `f` que l'on peut appeler pour lui soumettre un élément. En OCaml, cela s'écrit :

```
let rec on_interval i j f =
  if i < j then begin
    f i;
    on_interval (i + 1) j f
  end
```

Ce code se lit ainsi. Si `i` est strictement inférieur à `j`, alors l'entier `i` appartient à l'intervalle considéré, et en est le premier élément. On le soumet donc au consommateur via l'appel de fonction `f i`, que l'on pourrait aussi écrire `f(i)`. Ensuite, on continue, via un appel récursif (terminal) à `on_interval`. Si la condition `i < j` n'est pas satisfaite, alors l'intervalle considéré est vide ; on a terminé.

Le type de la fonction `f`, qui représente le consommateur, est `int -> unit`. En effet, cette fonction attend un élément (donc, ici, un entier), et ne renvoie aucun résultat. La fonction `on_interval`, qui a trois arguments `i`, `j` et `f`, admet donc le type suivant :

```
val on_interval : int -> int -> (int -> unit) -> unit
```

Pour effectuer une itération, on appelle donc `on_interval` en lui fournissant d'une part les bornes de l'intervalle et d'autre part un consommateur, qui souvent prend la forme d'une fonction anonyme `fun k -> ...`. Par exemple, pour afficher les entiers de `0` à `9`, on peut écrire :

```
on_interval 0 10 (fun k -> Printf.printf "%d\n" k);
```

Dans la bibliothèque d'OCaml, les fonctions `iter` que l'on trouve dans les modules `List`, `Array`, `String`, `Hashtbl`, `Set`, `Map`, `Queue`, `Stack`, etc. sont construites sur ce modèle. Ces fonctions permettent d'itérer sur les éléments stockés dans une structure de données. Cependant, comme le montre `on_interval` ci-dessus, un producteur peut également produire une séquence qui n'est pas explicitement stockée en mémoire.

Les choses s'écrivent de façon analogue en Java. Nous utilisons ici Java 8, car il permet d'écrire des fonctions anonymes sous une forme naturelle. Toutefois, nous avons expliqué (§1.2.3) qu'il ne s'agit là que d'une notation concise pour des objets qui implémentent une **interface fonctionnelle**. Par exemple, un consommateur, c'est-à-dire un objet capable de recevoir un élément, implémente l'interface `Consumer`. Cette interface, paramétrée par le type `T` des éléments, exige une seule méthode, nommée `accept` :

```
public interface Consumer<T> {
  void accept (T t);
}
```

Le type `Consumer<T>` de Java est donc essentiellement identique au type `'a -> unit` d'OCaml : dans les deux cas, ces types décrivent une fonction ou un objet à qui on peut soumettre un élément (et qui ne renvoie aucun résultat).

Le producteur `onInterval` s'écrit alors facilement, ici sous forme d'une méthode statique :

```
import java.util.function.Consumer;
public class OnInterval {
  public static void onInterval
  (int i, int j, Consumer<Integer> f) {
    for (int k = i; k < j; k++)
      f.accept(k);
  }
}
```

On a employé une boucle, parce que cela est plus idiomatique (et plus efficace) en Java qu'une fonction récursive. Pour chaque entier `k` compris dans l'intervalle considéré, on soumet `k` au consommateur, via l'appel `f.accept(k)`.

Comme en OCaml, pour effectuer une itération, on appelle `onInterval` en lui fournissant d'une part les bornes de l'intervalle et d'autre part un consommateur, qui souvent prend la forme d'une fonction anonyme `k -> ...`.

```
onInterval(0, 10, k -> System.out.println(k));
```

L'analogie avec OCaml est frappante ! Et ce style de programmation est courant dans de nombreux langages, par exemple Haskell et Scala, ou encore Python et R. Par exemple, la « boucle foreach » de Scala n'est rien d'autre qu'un appel à la méthode `foreach`, dont la signature est donnée par l'interface Traversable :

```
def foreach [U] (f: Elem => U) : Unit
```

On reconnaît le style présenté plus haut pour OCaml et Java : la fonction `f` doit recevoir un élément, le traiter, et renvoyer un résultat qui sera ignoré (ici, le type `U` de ce résultat est arbitraire).

En bref, grâce aux fonctions `on_interval` ou `onInterval`, nous avons l'impression d'avoir ajouté à notre langage de programmation une nouvelle forme de boucle : « **pour chaque** $k \in [i, j[$, **faire** ... ». Cela nous permet d'exprimer notre pensée de façon concise, en termes de haut niveau, sans devoir répéter à chaque fois les détails de l'écriture d'une fonction récursive terminale ou d'une boucle. On voit que le « pseudo-code » des cours d'algorithmique n'est rien d'autre, en réalité, que du code exécutable, précis, exprimé dans un langage de haut niveau !

Une fonction, comme `on_interval` ou `onInterval`, qui semble étendre ainsi le langage de programmation, et permet de s'exprimer en termes de plus haut niveau, est parfois appelée **combinateur**. En combinant les appels à de telles fonctions, on écrit de façon concise un algorithme complexe.

Solution page 118.

Exercice 4.1 (Recommandé) Cet exercice fait suite à l'Exercice 1.5, où l'on a défini en OCaml un type algébrique `tree` des arbres binaires contenant des éléments entiers. Écrivez une fonction `iter`, de type `tree -> (int -> unit) -> unit`, telle que `iter t f` énumère les éléments de l'arbre `t` (dans l'ordre infixe, de gauche à droite) et les soumet au consommateur `f`. Par

exemple, si `t` est l'arbre `Node (Node (Leaf, 1, Leaf), 2, Node (Leaf, 3, Leaf))` alors l'appel `iter t f` doit être équivalent à la séquence d'appels `f 1; f 2; f 3`. Pouvez-vous utiliser `iter` pour ré-implémenter de façon concise la fonction `elements` de l'Exercice 1.5 ? ◇

Exercice 4.2 Écrivez en Java ou en OCaml un producteur qui énumère toutes les permutations de l'intervalle $[0, n[$. Le consommateur reçoit donc, à chaque fois qu'il est appelé, un tableau d'entiers dans lequel il trouve une telle permutation. Le même tableau peut être utilisé à chaque appel, son contenu étant modifié par le producteur entre deux appels. ◇ Solution page 118.

4.1.2 Avec accumulateur explicite

Jusqu'ici, nous avons donné au consommateur le type `'a -> unit`, ou `Consumer<T>`. Ceci implique que, si le consommateur souhaite faire évoluer un certain état (par exemple, un compteur des éléments qu'on lui a soumis, ou une liste des éléments qu'on lui a soumis, etc.), alors il doit avoir accès à une structure de données modifiable. L'Exercice 4.1 illustre cela : pour implémenter `elements` à l'aide de `iter`, le consommateur doit employer une référence, interne au consommateur, qui mémorise la liste des éléments rencontrés jusqu'ici.

Or, il peut être souhaitable de ne pas obliger le consommateur à employer une structure de données modifiable. En effet, un programme **pur**, c'est-à-dire un programme qui n'emploie que des données non modifiables, est essentiellement une définition mathématique, et (pour cette raison) peut être plus facile à comprendre et à analyser.

Si l'on ne veut pas obliger le consommateur à stocker son état courant dans une structure modifiable, alors il faut que l'état courant devienne explicitement un argument et un résultat du consommateur et du producteur. Ainsi, si `'state` représente le type de l'état, le consommateur doit avoir le type `'a -> 'state -> 'state` : on lui soumet un élément ainsi que son état actuel, et (après avoir traité cet élément) il renvoie un nouvel état. Le producteur doit avoir un type de la forme `... -> ('a -> 'state -> 'state) -> 'state -> 'state` : on lui donne le consommateur ainsi qu'un état initial, et (après avoir effectué n appels au consommateur, qui à chaque fois ont produit un nouvel état) il renvoie l'état final. Cette valeur de type `'state`, transportée tout au long du calcul, est parfois appelée **accumulateur**, parce qu'elle sert souvent à « accumuler » une certaine information.

Quelle forme prend la fonction `on_interval` (§4.1.1) lorsqu'on lui ajoute un accumulateur ?

```
let rec on_interval_accu i j f accu =
  if i < j then begin
    let accu = f i accu in
    on_interval_accu (i + 1) j f accu
  end
  else
    accu
```

Le code est presque identique au précédent. L'appel à `f i` devient `f i accu`, car `f` attend à présent deux arguments. De plus, cet appel produit maintenant un résultat, à savoir un nouvel état. Nous nommons ce résultat à nouveau `accu` (voir la Remarque 4.1 ci-dessous), et l'utilisons lors de l'appel récursif à `on_interval_accu`. L'état courant évolue ainsi à chaque itération, et l'état qui est renvoyé ultimement (dans la branche `else`) est l'état final.

Remarque 4.1 (Masquage intentionnel) En OCaml, on peut donner le même nom à deux variables locales ; la plus récente masque la plus ancienne. Dans l'exemple précédent, la variable locale introduite par `let accu = ...` masque le paramètre `accu` de la fonction `on_interval`. Le fait de choisir le même nom, plutôt que de numéroter nos variables `accu0`, `accu1`, etc. garantit que seule la variable la plus récente peut être utilisée, ce qui permet d'éviter une erreur d'inattention. Ce style nous donne l'illusion que nous utilisons une variable locale `accu` modifiable, alors qu'en réalité les variables d'OCaml sont bel et bien immuables. ◇

Exercice 4.3 (Recommandé) Quel est le type de la fonction `on_interval_accu` ? ◇ Solution page 119.

Ce style peut sembler alourdir un peu l'écriture du producteur. Cependant, en contrepartie, il peut alléger l'écriture du consommateur. Par exemple, l'appel suivant renvoie la somme des éléments de l'intervalle $[0, 9[$:

```
on_interval_accu 0 10 (fun x accu -> x + accu) 0
```

tandis que celui-ci renvoie la liste (renversée) de ces éléments :

```
on_interval_accu 0 10 (fun x accu -> x :: accu) []
```

De façon générale, l'appel `on_interval_accu 0 10 f accu` est équivalent à :

```
let accu = f 0 accu in
let accu = f 1 accu in
...
let accu = f 8 accu in
let accu = f 9 accu in
accu
```

On peut l'écrire aussi sous la forme plus concise suivante :

```
f 9 (f 8 (... (f 1 (f 0 accu)) ... ))
```

Pour mieux comprendre ce que cela signifie, considérons le cas particulier où la fonction `f` implémente une opération binaire associative $\bullet$. En d'autres termes, supposons que l'appel `f x y` calcule $x \bullet y$. Alors, la valeur calculée par l'expression ci-dessus est :

$$9 \bullet 8 \bullet \ldots \bullet 1 \bullet 0 \bullet \text{accu}$$

On a pu omettre les parenthèses parce que l'opération $\bullet$ est associative. Si de plus la valeur initiale `accu` est l'élément neutre de l'opération $\bullet$, alors on peut écrire simplement :

$$9 \bullet 8 \bullet \ldots \bullet 1 \bullet 0$$

C'est pourquoi il doit être intuitivement évident que l'appel `on_interval_accu 0 10 (+) 0` (déjà mentionné plus haut, sous une forme moins concise) calcule la somme $9 + 8 + \ldots + 1 + 0$.

Les fonctions qui, comme `on_interval_accu`, appliquent ainsi une opération binaire à une suite d'éléments, sont souvent appelées « *reduce* », ou **réducteurs**, parce qu'elles « réduisent » une suite d'éléments à une seule valeur qui résume cette suite. Dans l'exemple précédent, une suite d'entiers est « réduite » à sa somme. Ces fonctions sont également traditionnellement appelées « *fold* ».

Notons que, si l'on demande que `f` implémente une opération associative, alors il faut que le type des éléments et le type de l'accumulateur coïncident, sans quoi l'équation qui exprime l'associativité de `f` n'aurait pas de sens. Si l'on ne pose pas cette condition d'associativité, alors le type des éléments et le type de l'accumulateur peuvent être distincts, comme c'est le cas par exemple dans l'expression `on_interval_accu 0 10 (fun x accu -> x :: accu) []`.

Dans la bibliothèque d'OCaml, les fonctions `fold` que l'on trouve dans les modules `List`, `Array`, `Hashtbl`, `Set`, `Map`, `Queue`, etc. sont construites sur ce modèle. La bibliothèque propose parfois plusieurs variantes, comme `List.fold_left` versus `List.fold_right` et `Array.fold_left` versus `Array.fold_right`, qui diffèrent par l'ordre dans lequel les éléments sont présentés au consommateur. Dans le cas de `List.fold_left` versus `List.fold_right`, il y a également une importante différence en termes de complexité en espace (Exercice 4.6).

Dans la bibliothèque de Java, la méthode `T reduce (T accu, BinaryOperator<T> f)` de l'interface `Stream` est un exemple de réducteur. Ce réducteur énumère les éléments d'un flot. Nous abordons la notion de flot en §4.3.

Solution page 119.

Exercice 4.4 (Recommandé) Écrivez en Java un réducteur pour un tableau. (Si cet énoncé vous paraît trop court, réfléchissez encore.) Indiquez comment on l'utilise pour calculer la somme des éléments d'un tableau de type `Integer[]`. ◇

Exercice 4.5 Cet exercice fait suite à l'Exercice 4.1, où l'on a défini en OCaml une fonction `iter` sur les arbres de type `tree`. Définissez à présent une fonction `fold` sur ces mêmes arbres. Cette fonction doit admettre le type `tree -> (int -> 'a -> 'a) -> 'a -> 'a`, où `'a` est le type de l'accumulateur. Indiquez comment utiliser `fold` pour calculer le plus grand élément (au sens de l'ordre usuel sur les entiers) contenu dans un arbre. ◇ Solution page 121.

Exercice 4.6 Écrivez en OCaml deux réducteurs pour les listes, à savoir `fold_left`, qui présente les éléments au consommateur dans l'ordre où ils apparaissent dans la liste, et `fold_right`, qui les lui présente dans l'ordre inverse. Quelles sont les complexités en temps et en espace de ces deux fonctions ? Peut-on ré-implémenter l'un des deux plus efficacement à l'aide de l'autre ? ◇ Solution page 121.

Exercice 4.7 On s'intéresse à des expressions mathématiques construites à partir de constantes entières à l'aide des opérateurs binaires + et ×. On souhaite représenter ces expressions sous forme symbolique, c'est-à-dire sous forme d'arbres. Pour cela, on définit en OCaml un type algébrique : Solution page 122.

```
type expr =
  Const of int
| Sum of expr * expr
| Prod of expr * expr
```

Écrivez une fonction `eval` de type `expr -> int` qui évalue une expression, c'est-à-dire qui en calcule la valeur numérique, en supposant que l'addition et le produit ont leur signification usuelle.

On souhaite ensuite varier l'interprétation des constantes et des opérateurs + et ×. Par exemple, on peut souhaiter évaluer une expression dans $\mathbb{Z}/n\mathbb{Z}$, pour un certain n fixé. Ou bien encore, on peut souhaiter l'évaluer dans l'algèbre $(\mathbb{Z}, \max, +)$, c'est-à-dire interpréter l'opérateur + comme le maximum dans $\mathbb{Z}$ et l'opérateur × comme la somme dans $\mathbb{Z}$. Pour cela, on pourrait écrire de nouvelles variantes de la fonction `eval`. Toutefois, une autre possibilité est d'écrire une fois pour toutes une fonction `fold` paramétrée par trois fonctions `const`, `sum`, et `prod`, qui donnent respectivement l'interprétation souhaitée pour les constantes et pour les opérateurs + et ×.

Donnez le type de cette fonction `fold`, puis écrivez-la. Utilisez-la ensuite pour redéfinir la fonction `eval` et pour définir les deux fonctions d'évaluation non standard mentionnées ci-dessus. ◇

Exercice 4.8 Cet exercice fait suite à l'Exercice 4.7. On souhaite à nouveau écrire une fonction d'évaluation canonique pour un type d'arbres. Cette fois, on étudie le problème en Java. La solution adoptée est essentiellement la même, et nous amène à (re)découvrir un motif de programmation connu sous le nom de **visiteur**. Solution page 123.

On s'intéresse toujours à des expressions mathématiques construites à partir de constantes entières à l'aide des opérateurs binaires + et ×. Définissez en Java une classe abstraite `Expr` et trois sous-classes `Const`, `Sum`, et `Prod`, de façon à représenter en mémoire une expression sous forme d'un arbre.

Définissez dans la classe `Expr` et dans ses sous-classes une méthode `int eval ()` qui évalue l'expression `this` suivant l'interprétation standard.

On voit que si l'on souhaite définir d'autres interprétations des expressions (par exemple l'interprétation dans $\mathbb{Z}/n\mathbb{Z}$ ou l'interprétation dans $(\mathbb{Z}, \max, +)$, comme dans l'Exercice 4.7) alors (à première vue) chaque nouvelle interprétation va exiger l'ajout d'une nouvelle méthode dans chacune des quatre classes. Cela n'est pas très agréable : pour des raisons de modularité, on aimerait pouvoir définir une nouvelle interprétation sans devoir modifier ces classes. Pour résoudre ce problème, on va écrire une fois pour toutes une méthode d'évaluation paramétrée par l'interprétation souhaitée.

Définissez une interface `Interpretation<A>` dotée de trois méthodes `constant`, `sum`, et `prod`. Pour définir l'interprétation des constantes, de la somme, et du produit, le client devra

fournir un objet de type `Interpretation<A>`. Dotez ensuite la classe `Expr` et ses sous-classes d'une méthode `<A> A fold (Interpretation<A> interpretation)` qui évalue l'expression `this` suivant l'interprétation définie par le paramètre `interpretation`.

Définissez un objet de type `Interpretation<Boolean>` qui représente l'interprétation dans les Booléens, où une constante i est interprétée par le Booléen $i \neq 0$ et où les opérateurs + et × sont interprétés respectivement comme la disjonction et la conjonction. Indiquez comment on évalue une expression `e` suivant cette interprétation non standard. ◇

4.2 Itérateurs modifiables

Confier le contrôle au producteur, comme nous l'avons fait plus haut (§4.1), n'est pas toujours très naturel ni très pratique.

Une situation typique qui pose problème est celle où **un** consommateur souhaite obtenir des éléments alternativement en provenance de **plusieurs** producteurs. Ce peut être, par exemple, pour **comparer deux séquences** d'éléments issus de producteurs distincts ; pour **produire une séquence de paires** à partir de deux séquences issues de producteurs distincts ; ou encore pour **fusionner deux séquences** issues de producteurs distincts. (Ces problèmes illustratifs font plus loin l'objet d'exercices. Voir l'Exercice 4.16 pour la comparaison, l'Exercice 4.19 pour le produit, et les Exercices 4.22 et 4.30 pour deux formes de fusion.) Si le contrôle est confié au producteur, c'est-à-dire si une séquence est représentée de façon implicite par une fonction « *fold* » qui permet d'examiner successivement **tous** les éléments de la séquence, alors aucun de ces problèmes ne peut être résolu de façon satisfaisante. La seule solution serait de convertir d'abord chaque séquence en une liste explicitement stockée en mémoire, et de travailler ensuite avec ces listes. Nous avons expliqué, au début de ce chapitre, pourquoi cette approche naïve n'est pas satisfaisante.

Une autre situation typique qui pose problème est celle où un consommateur s'écrit sous forme d'une ou plusieurs fonctions récursives et, au beau milieu d'une fonction, veut accéder au prochain élément d'une séquence. Ce peut être, par exemple, un analyseur syntaxique, écrit en style descendant, qui consomme une séquence de caractères. Si on a donné le contrôle au consommateur, il est facile d'accéder au prochain élément, sans perturber la structure du code : un **appel** au producteur suffit. Si au contraire on a donné le contrôle au producteur, alors le consommateur, pour accéder au prochain élément, doit non pas effectuer un appel de fonction, mais au contraire **rendre la main** au producteur. Or, cela est impossible lorsqu'on est au beau milieu de plusieurs appels de fonctions imbriqués. La seule solution est de réécrire le consommateur en style itératif, avec une pile explicite. Celui-ci peut alors s'interrompre pour accéder au prochain élément, puis reprendre là où il s'était interrompu, grâce à sa pile. Ce style n'est pas naturel : il modifie profondément le code du consommateur.

Pour résoudre ces problèmes, il faut confier le contrôle au consommateur. En d'autres termes, il faut que la notion de séquence, ou de producteur, se présente comme **un objet que l'on peut interroger** (à tout moment, quand on le souhaite) pour obtenir le prochain élément de la séquence, s'il en reste.

Nous appelons un tel objet un **itérateur**. Un itérateur produit, à la demande, un élément. Il donne donc accès à une séquence d'éléments. Cette séquence peut pré-exister ou non en mémoire. Souvent, un itérateur donne accès à une structure de données qui existe déjà en mémoire, comme une liste, un arbre, ou une table de hachage ; mais un itérateur peut aussi calculer le prochain élément au moment où cet élément est exigé.

Quel est le type d'un itérateur ? Sous quelle forme se présente-t-il ?

Un itérateur est défini principalement par le **service** qu'il propose, à savoir sa capacité à produire un élément à la demande. On peut donc le représenter sous la forme d'une fonction qui, lorsqu'on l'appelle, produit un élément (s'il en reste) ; ou bien sous la forme d'un objet doté d'une unique méthode qui, lorsqu'on l'appelle produit un élément (s'il en reste).

Cette approche offre l'avantage de présenter un itérateur comme une **abstraction**, c'est-à-

dire une entité opaque. Le consommateur sait que l'itérateur donne accès à une séquence, mais ne sait pas par quels moyens cette séquence est construite. Ainsi, on peut facilement modifier le code du producteur, ou remplacer un producteur par un autre, sans affecter le code du consommateur.

Un itérateur doit donc être une fonction au sens d'OCaml ou un objet au sens de Java. Afin de préciser tout à fait quel doit être le type d'un itérateur, une question demeure toutefois : si l'on appelle deux fois cette fonction, doit-elle produire et renvoyer deux éléments successifs de la séquence sous-jacente, ou bien doit-elle renvoyer à chaque appel le même élément ?

Les deux approches ont un sens, mais conduisent à des styles de programmation différents. Dans le premier cas, il faut que l'itérateur ait accès à un état interne **modifiable**, sans quoi il ne pourrait pas produire un nouvel élément à chaque fois qu'il est appelé. L'itérateur est alors **éphémère** : on ne peut l'utiliser qu'une fois, après quoi on atteint la fin de la séquence et l'itérateur devient inutile. Dans le second cas, il faut au contraire que l'itérateur s'appuie sur un état interne **immuable**, ce qui garantit que, même si on l'appelle plusieurs fois, il renvoie le même élément. Il faut alors que l'itérateur renvoie non seulement un élément, mais aussi un nouvel itérateur (immuable lui aussi) qui donne accès aux éléments suivants de la séquence. De tels itérateurs sont **persistants** : un itérateur immuable, une fois construit, représente pour toujours une séquence fixée, et peut être interrogé autant de fois qu'on le souhaite. Il est analogue en ce sens à une liste immuable ; mais, tandis qu'une liste est explicitement stockée en mémoire, un itérateur ne produit les éléments qu'à la demande.

Dans ce qui suit (§4.2), nous étudions les itérateurs modifiables, ou **itérateurs** tout court. Plus loin (§4.3), nous présentons les **flots**, que l'on peut considérer comme des itérateurs immuables dotés d'un mécanisme de mémoisation.

Nous pouvons à présent répondre à la question : quel est le type d'un itérateur ?

En OCaml, on peut appeler « itérateur » toute fonction capable à la demande de produire un élément ou bien d'indiquer qu'il n'en reste plus. Une telle fonction n'a pas d'argument, ou plutôt, en OCaml, on dit qu'elle a un argument de type `unit`. Elle renvoie un résultat de type `'a option`, où `'a` est le type des éléments. (Le type `option` est décrit par la Remarque 1.4.) Nous pouvons définir une abréviation de types :

```
type 'a iterator =
  unit -> 'a option
```

À peu de chose près, n'importe quelle fonction de type `'a iterator` est un itérateur valide. Il faut exiger seulement que, si la fonction renvoie un jour `None`, alors ensuite elle renvoie toujours `None`. En d'autres termes, `None` signale la fin de la séquence.

En Java, on peut imaginer différentes variantes de ce concept.

L'interface `Iterator`, qui existe depuis Java 1.2, exige la présence de deux méthodes, à savoir `hasNext`, qui renvoie un Booléen indiquant s'il reste un élément, et `next`, qui renvoie le prochain élément s'il existe et lance l'exception `NoSuchElementException` dans le cas contraire.

Détail 4.2 (À propos de `hasNext` et `remove`) La méthode `hasNext` peut à première vue sembler inutile, puisqu'au lieu d'appeler `hasNext` puis `next`, on peut appeler `next` directement et rattraper l'exception qui est lancée s'il n'y a plus d'éléments. Toutefois, `hasNext`, utilisée seule, permet de déterminer s'il existe encore un élément, sans le consommer ; ce que `next` ne permet pas. Voir, à ce sujet, l'Exercice 4.21.

La méthode `remove` de l'interface `Iterator` est optionnelle et n'a de sens que si l'itérateur parcourt une structure de données modifiable explicitement stockée en mémoire. Nous n'en parlerons pas. ◇

L'interface `Supplier` de Java 8 exige la présence d'une seule méthode, à savoir `get`, qui renvoie un élément. La documentation ne dit pas ce qui doit se passer s'il ne reste plus d'éléments ; on peut imaginer lancer `NoSuchElementException` ou bien renvoyer une valeur spéciale. Cette valeur spéciale pourrait être `null` ou bien, si l'on a défini une classe `Option` dotée de deux sous-classes `None` et `Some`, elle pourrait être `new None ()`. Dans ce dernier cas,

l'itérateur a le type `Supplier<Option<T>>`, qui est l'équivalent direct du type `'a iterator` ci-dessus.

D'autres objets plus complexes sont également des itérateurs au sens où nous l'entendons ici. Par exemple, la classe `InputStream` représente une séquence d'octets en provenance d'une source qui peut être un fichier, une connexion réseau, etc. Sa méthode `get` renvoie un octet (compris entre 0 et 255, mais présenté comme une valeur de type `int`) ou bien, s'il ne reste plus d'éléments, renvoie -1.

Solution page 125. **Exercice 4.9** Écrivez en OCaml une fonction `interval` de type `int -> int -> int iterator`, de sorte que `interval i j` renvoie un nouvel itérateur sur l'intervalle des entiers compris entre `i` au sens large et `j` au sens strict. ◇

Solution page 126. **Exercice 4.10** Écrivez une méthode `static Iterator<Integer> interval (int i, int j)`, de sorte que `interval(i, j)` renvoie un nouvel itérateur sur l'intervalle des entiers compris entre `i` au sens large et `j` au sens strict. ◇

Les exercices ci-dessus mettent en évidence le fait que si l'on choisit de donner le contrôle au consommateur, plutôt qu'au producteur, alors on complique l'écriture du producteur. Les fonctions `on_interval` et `onInterval` (§4.1.1) s'écrivaient en deux ou trois lignes, et leur structure était claire : elles contenaient une boucle ou un appel récursif terminal. Lorsqu'on écrit un itérateur, au contraire, la structure itérative du code n'est plus visible : il n'y a plus de boucle ou d'appel récursif. De plus, il faut introduire une structure de données modifiable (ici, un simple compteur) pour mémoriser l'état de l'itérateur et pouvoir, à chaque nouvel appel, reprendre l'itération au point où elle s'était arrêtée.

Comment utilise-t-on un itérateur ? Si on souhaite obtenir tous les éléments successivement, on écrit naturellement une boucle. Par exemple, ce fragment de code Java construit un itérateur à l'aide de la méthode `interval` de l'Exercice 4.10, puis en extrait les éléments et les affiche au fur et à mesure :

```
Iterator<Integer> it = interval(0, 10);
while (it.hasNext())
  System.out.println(it.next());
```

Cet idiome, où l'on construit d'abord un nouvel itérateur puis on itère sur ses éléments, est très courant en Java. Aussi les concepteurs du langage ont-ils décidé depuis Java 5 d'ajouter un **sucre syntaxique**, c'est-à-dire une notation plus légère pour cet idiome.

Ils ont introduit d'abord l'interface `Iterable<T>`, dotée d'une unique méthode `iterator`, qui renvoie un nouvel itérateur de type `Iterator<T>`. Un « itérable » est donc un objet capable de produire, à la demande, un nouvel itérateur ; on dit parfois que c'est une « usine à itérateurs », ou « *iterator factory* ». Ce sont là de grands mots ; en OCaml on dirait simplement que c'est une fonction de type `unit -> 'a iterator`.

Ils ont ensuite posé que, si `foo` est de type `Iterable<T>`, alors plutôt que d'écrire ceci :

```
Iterator<T> it = foo.iterator();
while (it.hasNext()) {
  T t = it.next();
  ...
}
```

on peut écrire cela :

```
for (T t : foo) {
  ...
}
```

Cette syntaxe est connue sous le nom de **boucle foreach**.

En OCaml, on ne dispose pas d'un sucre syntaxique analogue à la « boucle foreach » de Java. Néanmoins, on peut écrire une fois pour toutes un réducteur, c'est-à-dire une fonction « *fold* », qui tire ses éléments d'un itérateur. Ceci est le sujet de l'Exercice 4.12.

Exercice 4.11 À l'aide de la méthode `interval` définie dans l'Exercice 4.10, définissez une classe `Interval` qui implémente l'interface `Iterable<Integer>`. Dotez-la d'un constructeur approprié. Indiquez alors comment utiliser la « boucle foreach » pour itérer facilement sur un intervalle. ◇ Solution page 126.

Exercice 4.12 Écrivez en OCaml une fonction `on_iterator`, dont le type est `'a iterator -> ('a -> 'b -> 'b) -> 'b -> 'b`, telle que si `it` est un itérateur alors `on_iterator it f accu` a pour effet d'appliquer la fonction `f` successivement à tous les éléments produits par `it`, tout en transportant un accumulateur dont la valeur initiale est `accu`. Indiquez comment appliquer cette fonction à l'itérateur `interval 0 10` de l'Exercice 4.9 pour afficher les éléments de l'intervalle $[0, 10[$. Indiquez comment appliquer cette fonction à ce même itérateur pour calculer la somme des éléments de l'intervalle $[0, 10[$. ◇ Solution page 127.

Exercice 4.13 (Recommandé) Arthur, un ingénieur convaincu de l'intérêt des types abstraits (Chapitre 2) pense que considérer un itérateur comme une fonction, comme nous l'avons fait plus haut en définissant le type `'a iterator` comme synonyme de `unit -> 'a option`, n'est pas une bonne idée. Selon lui, si l'on implémente par exemple un itérateur sur les tableaux, mieux vaut déclarer un type abstrait `'a array_iterator` accompagné de deux fonctions : Solution page 128.

```
type 'a array_iterator
val create: 'a array -> 'a array_iterator
val next: 'a array_iterator -> 'a option
```

Selon Arthur, un itérateur n'est donc pas une fonction, mais un objet opaque que l'on peut créer à l'aide de `create` et que l'on peut utiliser à l'aide de `next`.

Proposez des définitions concrètes de ce type et de ces deux opérations.

Est-il possible d'appliquer la fonction `on_iterator` de l'Exercice 4.12 à un objet de type `'a array_iterator` ? Peut-on convertir un objet de type `'a array_iterator` en un itérateur de type `'a iterator` ? Quelles conclusions tirez-vous de cette réflexion ? ◇

Exercice 4.14 Cet exercice fait suite à l'Exercice 1.5, où l'on a défini en OCaml un type algébrique `tree` des arbres binaires contenant des éléments entiers. Écrivez une fonction `iterator` de type `tree -> int iterator` telle que l'itérateur `iterator t` énumère les éléments de l'arbre `t` dans l'ordre infixe, de gauche à droite, comme dans l'Exercice 4.1. ◇ Solution page 129.

Exercice 4.15 Cet exercice fait suite à l'Exercice 4.14. Étudiez le code de l'itérateur. Quelle est sa complexité en espace ? Quelle est la complexité en temps d'un appel à `next` ? Quelle est la complexité en temps d'une suite d'appels à `next` qui énumère tous les éléments de l'arbre ? ◇ Solution page 130.

Exercice 4.16 Le type `'a comparator` a été défini au Chapitre 3 comme une abréviation de `'a -> 'a -> int`. Écrivez une fonction `lexico` de type `'a comparator -> 'a iterator comparator` qui utilise l'ordre lexicographique sur les séquences pour comparer les séquences d'éléments produites par deux itérateurs. ◇ Solution page 130.

Exercice 4.17 Cet exercice fait suite à l'Exercice 1.5, où l'on a défini en OCaml un type algébrique `tree` des arbres binaires contenant des éléments entiers. Écrivez une fonction `same_elements` de type `tree -> tree -> bool` qui détermine si deux arbres ont la même séquence d'éléments, lorsqu'on considère les éléments dans l'ordre infixe, de gauche à droite. Vous vous appuierez sur les Exercices 4.14 et 4.16. ◇ Solution page 131.

Exercice 4.18 Écrivez en Java une classe `ArrayIterator<E>` qui permet d'avoir accès à la séquence des éléments d'un tableau et qui implémente l'interface `Iterator<E>`. Dotez-la d'un constructeur approprié. ◇ Solution page 131.

Exercice 4.19 À partir de deux itérateurs de types respectifs `Iterator<A>` et `Iterator<B>`, on souhaite construire un nouvel itérateur de type `Iterator<Pair<A, B>>`. Proposez en Java une implémentation de cette idée. ◇ Solution page 131.

Solution page 131. **Exercice 4.20** Écrivez en OCaml une fonction `map` de type `('a -> 'b) -> 'a iterator -> 'b iterator`. À l'aide de cette fonction et à l'aide de la fonction `interval` de l'Exercice 4.9, qui construit un itérateur sur un intervalle, définissez une fonction `array_iterator` de type `'a array -> (int * 'a) iterator` qui, étant donné un tableau, donne accès à la séquence des paires indice-élément de ce tableau. ◇

Solution page 132. **Exercice 4.21 (Difficile)** L'interface `Supplier<T>` de Java 8 exige la présence d'une méthode `get` qui, à la demande, produit un élément de type `T`. Nous supposerons que, lorsqu'il ne reste plus d'éléments, cette méthode lance l'exception `NoSuchElementException`. L'interface `Iterator<T>` exige la présence de méthodes `next` et `hasNext`. Écrivez un **adaptateur**, c'est-à-dire un objet qui implémente l'interface `Iterator<T>` en s'appuyant sur un objet sous-jacent de type `Supplier<T>`. Cet adaptateur est-il satisfaisant à tous points de vue ? Commentez. ◇

Solution page 135. **Exercice 4.22 (Difficile)** On suppose fixé un certain type `E` des éléments. On suppose ce type muni d'une relation d'ordre total. On suppose donnés k itérateurs qui produisent chacun une séquence croissante au sens de cette relation d'ordre. On souhaite alors construire un nouvel itérateur qui produit une séquence croissante obtenue par fusion de ces k séquences. Par exemple, si on se donne trois itérateurs qui produisent respectivement les séquences « 1, 3 » et « 2, 2, 4 » et « 1, 4, 12 » alors on souhaite que l'itérateur obtenu produise la séquence « 1, 1, 2, 2, 3, 4, 4, 12 ». Implémentez cette idée en Java sous la forme d'une classe `MultiwayMergeIterator<E>` qui implémente l'interface `Iterator<E>` et dont le constructeur attend d'une part la relation d'ordre, représentée par un objet de type `Comparator<E>`, d'autre part les k itérateurs, donnés dans un tableau de type `Iterator<E>[]`. Quelle est la complexité en temps et en espace de votre solution ? On ne compte pas le temps et l'espace consommé par les k itérateurs sous-jacents. ◇

4.3 Flots

Nous avons étudié plus haut (§4.2) des itérateurs dont l'état interne est **modifiable**. Ces itérateurs sont représentés en OCaml par le type `unit -> 'a option`. Un itérateur est donc une fonction qui d'une part produit un élément (s'il en reste), d'autre part modifie son propre état interne de façon à produire, au prochain appel, l'élément suivant.

Nous nous intéressons maintenant à une approche légèrement différente, où l'itérateur est **immuable**. Un itérateur doit alors renvoyer non seulement un élément, mais aussi un nouvel itérateur (immuable lui aussi) qui donne accès aux éléments suivants.

On pourrait modéliser cette idée en OCaml à l'aide de l'abréviation de types suivante :

```
type 'a immutable_iterator =
  unit -> ('a * 'a immutable_iterator) option
```

Toutefois, cette abréviation est récursive, ce que le compilateur OCaml ne permet pas (voir le Détail 4.3). Heureusement, il est possible de la présenter sous une forme équivalente et admise par le compilateur. Pour cela, nous allons nous appuyer sur le fait que, si les abréviations de types récursives sont interdites, les définitions de types algébriques récursifs sont autorisées : voir, par exemple, le type des listes (Exercice 1.4).

La définition ci-dessus mentionne le type `option` (Remarque 1.4), dont les constructeurs sont `None` et `Some`. Ici, `None` signifie qu'il n'y a plus d'éléments, et `Some (x, xs)` signifie que nous avons un élément `x` suivi d'une suite d'éléments donnée par l'itérateur immuable `xs`. Remplaçons le type `option` et ses constructeurs `None` et `Some` par un type algébrique spécialisé, que nous nommons `head`, et dont nous nommons les constructeurs `Nil` et `Cons`. Nous obtenons les définitions suivantes :

```
type 'a immutable_iterator =
  unit -> 'a head

and 'a head =
```

```
| Nil
| Cons of 'a * 'a immutable_iterator
```

Le type algébrique `head` et l'abréviation de types `immutable_iterator` sont mutuellement récursifs, ce que le compilateur OCaml autorise.

En réalité, nous venons de redécouvrir ici une version à peine modifiée du type des listes. Si nous décidions de faire disparaître l'abréviation `'a immutable_iterator` en la remplaçant par sa définition `unit -> 'a head`, nous pourrions écrire directement :

```
type 'a head =
| Nil
| Cons of 'a * (unit -> 'a head)
```

Comparez ceci à la définition du type `'a list` (Exercice 1.4). La seule différence est que, dans le second argument du constructeur `Cons`, on voit apparaître le type de fonction `unit -> ...`. Ceci indique que la suite de la séquence n'est pas (nécessairement) déjà disponible en mémoire, comme dans une liste ordinaire, mais est (peut-être) construite à la demande.

Ceci suggère que les itérateurs immuables sont une notion particulièrement naturelle. Ils ne sont qu'**une variante des listes** simplement chaînées, qui constituent la représentation la plus simple des séquences.

Les itérateurs immuables sont d'ailleurs déjà anciens. On les rencontre par exemple sous le nom de « flots » dans un célèbre article où Landin (1965) propose une sémantique (c'est-à-dire une explication) du langage de programmation Algol 60 sous forme d'une traduction vers le λ-calcul. Landin note (page 95) qu'il est nécessaire, pour expliquer la construction **for** d'Algol 60, de se doter de séquences dont les éléments sont calculés à la demande. Il définit ces séquences comme des fonctions, comme nous l'avons fait ci-dessus lorsque nous avons posé `type 'a immutable_iterator = unit -> 'a head`.

Vis-à-vis d'une liste simplement chaînée, un itérateur immuable offre certains avantages. Ses éléments ne sont calculés qu'à la demande, donc ne sont pas calculés s'ils ne sont pas nécessaires. Il peut donc permettre de travailler en espace $O(1)$ tandis qu'une liste de n éléments exigerait un espace $O(n)$. De plus, dans le cas extrême où n est ∞, un itérateur immuable permet de représenter une **séquence infinie** (Exercice 4.23), qui ne peut pas être représentée par une liste ordinaire.

Vis-à-vis d'un itérateur modifiable, un itérateur immuable présente également certains avantages. Son principal intérêt est la **persistance** : un itérateur immuable, une fois construit, représente pour toujours une séquence fixée. On peut donc parler de « **ses éléments** » sans préciser à quel instant on les considère, de la même manière qu'on parlerait « des éléments » d'une liste immuable. Cela facilite le raisonnement. De plus, contrairement à un itérateur modifiable, qui ne peut être utilisé qu'une fois, un itérateur immuable peut être interrogé autant de fois qu'on le souhaite, par exemple pour itérer plusieurs fois sur « ses éléments ». Il peut donc être **partagé** par plusieurs utilisateurs sans qu'il y ait interférence entre eux : chaque utilisateur aura accès à la même séquence, à savoir « les éléments » de cet itérateur.

En contrepartie, un itérateur immuable a un certain coût. Le temps nécessaire pour obtenir un élément est potentiellement augmenté d'un facteur constant, parce qu'il faut typiquement un appel de fonction et une allocation de mémoire. Ce coût dépend de différents détails de l'implémentation des itérateurs et de la machine utilisée. Suivant les applications, il peut être acceptable ou non.

Détail 4.3 (Abréviations de types récursives) Le compilateur OCaml accepte les abréviations de types récursives si l'on active l'option `-rectypes`. Toutefois, cela est fortement déconseillé, car le compilateur accepte alors également des programmes qui n'ont guère de sens, comme cette liste qui a elle-même pour unique élément :

```
let rec xs = [xs]
```

Il considère que `xs` admet le type récursif `'a list as 'a`, ou en termes plus intuitifs, le type infini « liste de liste de liste de . . . ». Ce type a certes un sens, mais en pratique, mieux vaut rejeter la définition de `xs`, qui est probablement erronée. ◇

Solution page 135. **Exercice 4.23 (Recommandé)** Définissez un itérateur de type `int immutable_iterator` qui produit la séquence infinie $1, -1, 1, -1, \ldots$ ◇

Solution page 135. **Exercice 4.24 (Recommandé)** Écrivez en OCaml une fonction `interval` dont le type est `int -> int -> int immutable_iterator`, de sorte que `interval i j` renvoie un itérateur immuable sur l'intervalle des entiers compris entre `i` au sens large et `j` au sens strict.

On construit un itérateur immuable à l'aide de la définition `let it = interval 0 n`. Quelle est la complexité en temps de cet appel à la fonction `interval` ? Quel est l'espace occupé en mémoire par l'itérateur `it` ? ◇

Solution page 137. **Exercice 4.25 (Recommandé)** Écrivez un réducteur permettant d'itérer sur les éléments d'un itérateur immuable. Il doit prendre la forme d'une fonction `on_immutable_iterator` de type `'a immutable_iterator -> ('a -> 'b -> 'b) -> 'b -> 'b`. On précise, s'il était besoin, que les éléments doivent être présentés au consommateur dans l'ordre de la séquence (et non dans l'ordre inverse) et que l'itération doit se faire en espace $O(1)$.

Comme dans l'Exercice 4.24, on pose `let it = interval 0 n`. Puis, on calcule la somme des éléments de `it` via la définition `let sum = on_immutable_iterator it (+) 0`. Quel est, après ces deux définitions, l'espace occupé en mémoire par `it` et par `sum` ? ◇

Avant de continuer notre étude des itérateurs immuables, nous nous proposons d'apporter une dernière modification à leur définition.

Nous avons remarqué plus haut qu'un itérateur immuable permet d'itérer plusieurs fois sur une même séquence. À première vue, cela semble très utile et confortable. Toutefois, il faut prendre garde au fait que, puisque les éléments de la séquence sont calculés à la demande, s'ils sont demandés une deuxième fois, alors ils seront calculés une deuxième fois. Si par exemple on a construit un itérateur immuable pour la séquence infinie des nombres premiers, alors itérer plusieurs fois sur (un préfixe fini de) cette séquence exigera plusieurs fois le calcul de chaque nombre premier. Si ce calcul est coûteux en temps, cela peut être indésirable.

Une solution naturelle à ce problème est la **mémoisation**. Si l'on est prêt à employer plus d'espace afin de gagner du temps, alors on peut effectuer le calcul de chaque nombre premier seulement lorsqu'il est exigé pour la première fois. Ensuite, on mémorise ce nombre, de façon à ne pas devoir répéter ce calcul si ce même nombre est exigé une nouvelle fois.

Pour mettre en œuvre cette idée, il suffit de modifier très légèrement la définition des itérateurs immuables. On remplace simplement la notion de fonction, notée `unit -> 'a` en OCaml, par la notion de **suspension** (§1.2.4), notée `'a Lazy.t` en OCaml. Dans les deux cas, le calcul est retardé : aucun calcul n'a lieu tant que l'élément n'est pas exigé. Dans le cas où on emploie une fonction ordinaire, un calcul est potentiellement répété à chaque fois que l'élément est exigé, tandis que si on emploie une suspension, le calcul n'a lieu que la première fois que l'élément est exigé ; son résultat est ensuite conservé en mémoire.

Voici donc une définition en OCaml des itérateurs immuables avec mémoisation, ou **flots** :

```
type 'a stream =
  'a head Lazy.t

and 'a head =
| Nil
| Cons of 'a * 'a stream
```

En Java, si l'on dispose d'une implémentation des suspensions, par exemple la classe `Thunk<T>` de l'Exercice 1.12, alors on peut donner une définition analogue des flots :

```
public class Stream<T> {
  public final Thunk<Head<T>> thunk;
  // Constructor omitted.
}
public abstract class Head<T> {}
public class Nil<T> extends Head<T> {}
public class Cons<T> extends Head<T> {
  public final T elem;
  public final Stream<T> rest;
  // Constructor omitted.
}
```

Parce qu'on ne peut pas définir d'abréviation de types en Java, nous ne pouvons pas définir `Stream<T>` comme une simple abréviation pour `Thunk<Head<T>>`. Nous choisissons alors de définir une classe `Stream<T>` dotée d'un unique champ `thunk` de type `Thunk<Head<T>>`. Cela nous coûte un objet et une indirection supplémentaires, mais c'est la solution la plus agréable. Ensuite, comme lors de la définition des listes ordinaires en Java (Exercice 1.4), nous définissons la classe `Head` comme une classe abstraite dotée de deux sous-classes `Nil` et `Cons`. La première n'a aucun champ, tandis que la seconde a deux champs `elem` et `rest`, qui contiennent respectivement le premier élément du flot et le flot des éléments suivants.

Exercice 4.26 (Recommandé) Cet exercice fait suite aux Exercices 4.24 et 4.25. Écrivez en OCaml une fonction `interval` dont le type est `int -> int -> int stream`, de façon à ce que `interval i j` produise un flot des entiers compris entre `i` au sens large et `j` au sens strict. Écrivez ensuite un réducteur permettant d'itérer sur un flot. Il doit prendre la forme d'une fonction `on_stream` de type `'a stream -> ('a -> 'b -> 'b) -> 'b -> 'b`. Solution page 137.

On pose `let s = interval 0 n`. Quel est l'espace occupé en mémoire par le flot `s` ? On pose ensuite `let sum = on_stream s (+) 0`. Quel est alors l'espace occupé en mémoire par le flot `s` ? ◇

Exercice 4.27 (Recommandé) Cet exercice fait suite à l'Exercice 4.26. On calcule la somme des entiers de `0` à `n` exclus en écrivant directement `let sum = on_stream (interval 0 n) (+) 0`. Quel est l'espace mémoire nécessaire pour effectuer ce calcul ? ◇ Solution page 138.

Les flots sont des objets étonnants. Ils combinent certains aspects des itérateurs immuables et certains aspects des listes simplement chaînées. Comme un itérateur immuable, un flot ne produit ses éléments qu'à la demande du consommateur. On évite ainsi les inconvénients liés à l'utilisation naïve des listes, décrits au début de ce chapitre. Comme une liste, cependant, un flot **une fois évalué** est représenté en mémoire par une liste chaînée de suspensions, donc (**s'il est accessible**) occupe en mémoire un espace important.

Les Exercices 4.26 et 4.27 ci-dessus montrent qu'un flot construit par l'appel de fonction `interval 0 n` occupe un espace $O(1)$ tant qu'il n'est pas évalué ; occupe un espace $O(n)$ s'il a été évalué et si on a conservé un pointeur vers le début de ce flot ; mais occupe seulement un espace $O(1)$ si l'on a évalué ses i premiers éléments et si l'on conserve uniquement un pointeur vers le flot (encore non évalué) des éléments suivants.

Les flots sont une structure de données intéressante, qui mérite d'être connue, et qui est parfois préférable aux itérateurs (modifiables ou immuables) sans mémoisation. Toutefois, parce que les éléments d'un flot sont produits à la demande, il est parfois difficile de comprendre dans quel ordre les calculs s'enchaînent ; et parce que la mémoire occupée par un flot peut être difficile à analyser, il peut être difficile de prédire combien d'espace est nécessaire pour effectuer un calcul.

Exercice 4.28 (Recommandé) Écrivez en OCaml une fonction `append` telle que `append xs ys` renvoie en temps $O(1)$ un flot contenant les éléments du flot `xs` suivis des éléments du flot `ys`. Le type de la fonction `append` doit être `'a stream -> 'a stream -> 'a stream`. ◇ Solution page 138.

Solution page 140. **Exercice 4.29** Ajoutez à la classe `Stream` une méthode `Stream<T> append (Stream<T> that)` qui construit en temps $O(1)$ la concaténation des flots `this` et `that`. On donne les conseils suivants. Traitez si possible d'abord l'Exercice 4.28, et inspirez-vous de votre solution à cet exercice. Définissez dans la classe `Head` une méthode abstraite `Head<T> append (Stream<T> that) throws Exception`, et implémentez cette méthode dans les sous-classes `Nil` et `Cons`. Les méthodes `append` des classes `Stream` et `Nil`/`Cons` seront alors mutuellement récursives. ◇

Solution page 141. **Exercice 4.30** Écrivez en OCaml une fonction `interleave` de type `'a stream -> 'a stream -> 'a stream` telle que le flot `interleave xs ys` est constitué alternativement d'un élément du flot `xs`, un élément du flot `ys`, etc. ◇

Solutions des exercices

Solution de l'exercice 1.1 Une solution en Java est donnée dans la Figure 4.1. Sans grande surprise, on représente les entiers étendus comme un type somme à trois branches : un entier étendu est soit un entier ordinaire, soit $+\infty$, soit $-\infty$. Il faut donc une classe abstraite `XInt` et trois sous-classes `XIntMachine`, `XIntPosInfinity`, et `XIntNegInfinity`. La première est dotée d'un champ `value` de type `int`. Ce champ est immuable : on ne manipule ici que des objets immuables.

Afin de permettre la construction d'entiers relatifs de toutes sortes, il faut que chacune des trois sous-classes soit dotée d'un constructeur. Pour `XIntMachine`, on écrit explicitement ce constructeur. Pour `XIntPosInfinity`, `XIntNegInfinity`, le constructeur par défaut (défini automatiquement par Java, et qui ne fait rien) convient.

Afin de permettre la conversion d'un entier relatif en chaîne de caractères, on redéfinit la méthode `toString` (qui est héritée de la classe `Object`) dans chacune des trois sous-classes. On pourrait ajouter en tête le mot-clef `@Override` afin de souligner qu'il s'agit ici d'une redéfinition de méthode.

L'opération `leq`, qui doit permettre de comparer deux entiers relatifs, prend la forme d'une méthode `boolean leq (XInt that)`, chargée de comparer les entiers relatifs `this` et `that`. Cette méthode est déclarée dans la classe abstraite `XInt` et définie dans chacune des trois sous-classes. Dans la sous-classe `XIntNegInfinity`, c'est immédiat. Dans la sous-classe `XIntPosInfinity`, il faut déterminer si l'argument `that` est lui-même $+\infty$ ou non. Pour effectuer cette analyse de cas, on choisit ici d'utiliser l'opérateur `instanceof` de Java. Enfin, dans la sous-classe `XIntMachine`, il faut analyser l'objet `that` et distinguer trois cas. Pour effectuer cette analyse de cas, on choisit ici d'utiliser une méthode auxiliaire, dont la signature est `boolean geq (int that)`. Cette méthode est chargée de comparer l'entier relatif `this` à l'entier machine `that`. À nouveau, elle doit être déclarée dans la classe parent `XInt` et définie dans chacune des trois sous-classes.

De manière générale, l'analyse de l'étiquette d'un objet en Java peut être effectuée soit implicitement via un appel de méthode, soit explicitement à l'aide de l'opérateur `instanceof` ou à l'aide d'un cast (Remarque 1.1).

Enfin, l'opération `max` peut être implémentée à l'aide de l'opération `leq`, sans analyse de cas. On définit donc cette méthode dans la classe parent `XInt`.

Une solution en OCaml est donnée dans la Figure 4.2. À nouveau, le type `xint` est défini comme un type somme à trois branches. Dans le cas où l'étiquette est `XIntMachine`, l'objet contient un champ de type `int`.

La fonction `toString` s'écrit sous forme d'une analyse de cas.

La fonction `leq` s'écrit également sous forme d'une analyse de cas. Cette fois, il est pratique d'étudier simultanément les étiquettes des deux objets `x` et `y`. La construction `match` d'OCaml permet cela. Le compilateur vérifie pour nous que tous les cas sont bien traités et qu'aucune branche n'est inutile.

Enfin, comme en Java, la fonction `max` s'écrit à l'aide de la fonction `leq`, donc sans analyse de cas. ◇

Solution de l'exercice 1.2 On définit le type `box` comme un type enregistrement à un champ :

```
abstract class XInt {
  abstract boolean leq (XInt that); // this <= that?
  abstract boolean geq (int that);  // this >= that?
  XInt max (XInt that) {            // max(this, that)
    return this.leq(that) ? that : this;
  }
}
class XIntMachine extends XInt {
  final int value;
  XIntMachine (int value) { this.value = value; }
  public String toString () { return Integer.toString(value); }
  boolean leq (XInt that) { return that.geq(value); }
  boolean geq (int that) { return value >= that; }
}
class XIntPosInfinity extends XInt {
  public String toString () { return "+infinity"; }
  boolean leq (XInt that) { return that instanceof XIntPosInfinity; }
  boolean geq (int that) { return true; }
}
class XIntNegInfinity extends XInt {
  public String toString () { return "-infinity"; }
  boolean leq (XInt that) { return true; }
  boolean geq (int that) { return false; }
}
```

FIGURE 4.1 – Entiers étendus en Java

```
type xint =
| XIntMachine of int
| XIntPosInfinity
| XIntNegInfinity

let toString x =
  match x with
  | XIntMachine i ->
      string_of_int i
  | XIntPosInfinity ->
      "+infinity"
  | XIntNegInfinity ->
      "-infinity"

let leq x y =
  match x, y with
  | _, XIntPosInfinity
  | XIntNegInfinity, _ ->
      true
  | XIntMachine i, XIntMachine j ->
      i <= j
  | XIntMachine _, XIntNegInfinity
  | XIntPosInfinity, _ ->
      false

let max x y =
  if leq x y then y else x
```

FIGURE 4.2 – Entiers étendus en OCaml

```
type box =
  { mutable contents: int }
```

Puis on définit les trois fonctions demandées :

```
let create (x : int) : box =
  { contents = x }
let get (b : box) : int =
  b.contents
let set (b : box) (y : int) : unit =
  b.contents <- y
```

Bien que cela ne soit pas nécessaire, on a indiqué explicitement le type de chaque argument et le type du résultat des trois fonctions `create`, `get`, `set`. L'appel `create x` crée un nouveau bloc de type `box` dont le contenu initial est l'entier `x`. L'appel `get b` renvoie le contenu de la boîte `b`. L'appel `set b y` modifie le contenu de la boîte `b` en y écrivant la nouvelle valeur `y`.

Pour que le type du contenu ne soit pas nécessairement `int` mais un type au choix de l'utilisateur, on modifie la définition du type `box`, qui devient un **type paramétré** :

```
type 'a box =
  { mutable contents: 'a }
```

On dispose alors du type `int box` des boîtes à contenu entier, du type `bool box` des boîtes à contenu booléen, etc. Les fonctions `create`, `get`, `set` ne sont pas modifiées, sauf pour ce qui concerne les annotations de types : il faut remplacer `int` par `'a` et remplacer `box` par `'a box`. On peut aussi préférer supprimer entièrement ces annotations, comme ceci :

```
let create x =
  { contents = x }
let get b =
  b.contents
let set b y =
  b.contents <- y
```

Supprimer les annotations rend le code plus léger, mais peut conduire à des messages d'erreur plus difficiles à déchiffrer dans le cas où le programme n'est pas bien typé. Il est recommandé de conserver ces annotations.

Dans la bibliothèque d'OCaml, le type `'a box` s'appelle `'a ref`, et les fonctions `create`, `get`, `set` sont nommées respectivement `ref`, `!` et `:=`. On écrit donc `let r = ref 42 in ...` pour créer une nouvelle référence dont le contenu est 42. Puis on écrit `!r` pour en lire le contenu, et `r := 0` pour en modifier le contenu. Les références sont particulièrement utiles parce qu'OCaml n'a pas de variables locales modifiables. ◇

Solution de l'exercice 1.3 Une solution est donnée dans la Figure 4.3. Naturellement, on dote la classe `Box` d'un champ `content` de type `X`. De plus, afin de pouvoir déterminer si une boîte a été initialisée ou non, on prévoit un champ `initialized` de type `boolean`. Les champs sont marqués `private` afin que seules les méthodes `get` et `set` y aient accès. On peut ainsi garantir le respect du protocole prévu dans l'énoncé.

Il serait tentant de se passer du champ `initialized` en stockant dans le champ `content` la valeur `null` lorsque la boîte n'est pas initialisée. Cependant, cela ne serait pas correct, car il serait alors impossible de distinguer une boîte non initialisée et une boîte initialisée avec la valeur `null`. A priori, d'après l'énoncé, l'utilisateur a le droit d'appeler `set(null)`, et il faut que le code se comporte correctement dans ce cas.

Le champ `initialized` reçoit initialement la valeur `false`, tandis que le champ `content` n'est pas explicitement initialisé. Il est inutile ici d'écrire explicitement un constructeur : le constructeur par défaut (défini automatiquement par Java, et qui ne fait rien) convient.

La méthode `set` vérifie que la boîte n'a pas encore été initialisée, et stocke son argument `x` dans le champ `content`, dont le contenu jusqu'ici était `null`. Si la boîte a déjà été initialisée, une

```
public class Box<X> {
  private X content;
  private boolean initialized = false;
  public void set (X x) {
    if (initialized)
      throw new Error ("Attempt to initialize a box twice");
    content = x;
    initialized = true;
  }
  public X get () {
    if (!initialized)
      throw new Error ("Access to uninitialized box");
    return content;
  }
}
```

FIGURE 4.3 – Une « boîte » à initialisation unique

exception est lancée. On choisit d'utiliser ici l'exception `Error`, mais on pourrait aussi utiliser `Exception`, ou encore définir une nouvelle sous-classe de `Error` ou `Exception`.

La méthode `get` vérifie que la boîte a été initialisée, et renvoie le contenu du champ `content`, qui provient d'un précédent appel à `set`. ◇

Solution de l'exercice 1.4 Une solution en Java est donnée dans la Figure 4.4. Puisque le type de listes est un type somme à deux branches, on le traduit en Java sous forme d'une classe abstraite `IntList` dotée de deux sous-classes `Nil` et `Cons`. La classe `Nil` n'a aucun champ ; la classe `Cons` a deux champs, `head` et `tail`, qui contiennent respectivement le premier élément de la liste et le reste de la liste. Elle est dotée d'un constructeur (trivial) qui initialise ces champs. Les champs sont marqués `final`, donc sont immuables.

Pour permettre la conversion d'une liste en une chaîne de caractères, il suffit de redéfinir dans chaque sous-classe la méthode `toString`, qui est héritée de la classe `Object`. Notons que Java effectue des conversions implicites vers le type `String` : l'expression `head + ":: " + tail` signifie en réalité `Integer.toString(head) + ":: "+ tail.toString()`.

Pour permettre de déterminer si une liste contient ou non un élément pair, il faut une méthode de signature `boolean hasEvenElement ()`. On la déclare dans la classe parent, et on la définit dans les deux sous-classes.

Une solution en OCaml est donnée dans la Figure 4.5. Le type des listes s'exprime sous forme d'un type algébrique `intlist` doté de deux constructeurs, `Nil` et `Cons`. À nouveau, le premier n'a pas de champs, tandis que le second a deux champs (anonymes) dont les types respectifs sont `int` et `intlist`.

Les opérations `toString` et `hasEvenElement` s'écrivent sous forme de fonctions récursives définies par cas.

Le type des listes est défini de la même manière dans la bibliothèque d'OCaml. Toutefois, le type des éléments n'est pas fixé : on paramètre la définition par un type arbitraire `'a`. On obtient donc un type `'a list` des listes dont les éléments ont le type `'a`.

De plus, dans la bibliothèque d'OCaml, les constructeurs du type `'a list` sont nommés non pas `Nil` et `Cons`, mais `[]` et `::`. C'est pourquoi la liste vide s'écrit `[]`. De plus, par convention, l'application du constructeur `::` à deux arguments `x` et `xs` s'écrit non pas `::(x, xs)`, mais sous forme infixe, `x :: xs`. Même si cette notation spéciale peut sembler étrange, il faut garder à l'esprit qu'elle dénote en réalité l'allocation d'un objet dans le tas : elle est analogue à l'expression `new Cons (x, xs)` de Java.

Enfin, vous savez peut-être qu'on peut écrire en OCaml une liste de trois éléments sous la forme `[x; y; z]`. Il s'agit simplement d'une notation pour `x :: y :: z :: []`. L'évaluation

```
abstract class IntList {
  abstract boolean hasEvenElement ();
}
class Nil extends IntList {
  public String toString () { return "[]"; }
  boolean hasEvenElement () { return false; }
}
class Cons extends IntList {
  final int head;
  final IntList tail;
  public Cons (int head, IntList tail) {
    this.head = head;
    this.tail = tail;
  }
  public String toString () {
    return head + " :: " + tail;
  }
  boolean hasEvenElement () {
    return head % 2 == 0 || tail.hasEvenElement();
  }
}
```

FIGURE 4.4 – Listes chaînées à éléments entiers

```
type intlist =
| Nil
| Cons of int * intlist

let rec toString xs =
  match xs with
  | Nil ->
      "[]"
  | Cons (x, xs) ->
      string_of_int x ^ " :: " ^ toString xs

let rec hasEvenElement xs =
  match xs with
  | Nil ->
      false
  | Cons (x, xs) ->
      x mod 2 = 0 || hasEvenElement xs
```

FIGURE 4.5 – Listes chaînées à éléments entiers

de cette expression provoque donc l'allocation de trois objets `Cons` et d'un objet `Nil`. ◇

Solution de l'exercice 1.5 La fonction `elements` s'écrit de manière très naturelle sous forme récursive. Si l'arbre est `Leaf`, la liste de ses éléments est vide. Si l'arbre est de la forme `Node (t1, x, t2)` alors la liste de ses éléments est obtenue en concaténant la liste des éléments de `t1`, la liste singleton `[x]`, et la liste des éléments de `t2`. Le code est donc :

```
let rec elements t =
  match t with
  | Leaf ->
      []
  | Node (t1, x, t2) ->
      elements t1 @ [x] @ elements t2
```

Si `xs` et `ys` sont deux listes de longueurs respectives m et n, alors la complexité de l'appel `xs @ ys` est $O(m)$. On peut s'en convaincre en étudiant le code de la fonction `List.append`, qui parcourt récursivement son premier argument.

Étudions à présent la complexité asymptotique en temps de la fonction `elements`. Notons N la taille (c'est-à-dire le nombre d'éléments) de l'arbre `t`. D'après la remarque précédente, l'appel `elements t1 @ ...` a un coût proportionnel à la taille du sous-arbre `t1`, c'est-à-dire $O(N)$. Nous risquons donc de payer un coût $O(N)$ à chaque nœud de l'arbre `t`. Il s'ensuit que le coût total de la traversée de l'arbre est $O(N^2)$. Ce code n'est pas efficace et ne doit pas être utilisé !

Écrivons à présent la fonction `elements_append` exigée par l'énoncé. Au lieu de donner directement sa définition, nous procédons de façon plus graduelle, pour expliquer comment on peut la construire. D'après l'énoncé, il faut que l'appel `elements_append t tail` produise le même résultat que l'appel `(elements t) @ tail`. Nous pouvons donc donner une première définition de `elements_append` en reproduisant la définition de `elements` et en y ajoutant, dans chaque branche, la concaténation `... @ tail` :

```
let elements_append t tail =
  match t with
  | Leaf ->
      [] @ tail
  | Node (t1, x, t2) ->
      elements t1 @ [x] @ elements t2 @ tail
```

Nous savons, par construction, que cette définition (non récursive) est correcte. Nous allons maintenant la simplifier et faire en sorte qu'elle ne fasse plus appel à la fonction inefficace `elements`.

Dans la première branche, on peut simplifier l'expression `[] @ tail`, qui vaut simplement `tail`. Dans la seconde branche, on peut remplacer l'expression `[x] @ ...` par `x :: ...`. Ces simplifications ne sont pas essentielles : elles ne modifient pas la complexité asymptotique de ce code.

Il reste à éliminer les appels à `elements`. C'est facile. Nous savons que, par construction, `elements_append t2 tail` est égale à `elements t2 @ tail`. Nous pouvons donc remplacer ci-dessus la seconde expression par la première. Nous pouvons, de la même manière, remplacer l'appel `elements t1 @ ...` par `elements_append t1 (...)`. Nous obtenons alors :

```
let rec elements_append t tail =
  match t with
  | Leaf ->
      tail
  | Node (t1, x, t2) ->
      elements_append t1 (x :: elements_append t2 tail)
```

Cette fonction (récursive) est de complexité $O(N)$. En effet, le coût des opérations élémentaires effectuées à chaque nœud (analyse de l'étiquette de `t` ; lecture des champs `t1`, `x`, `t2` ; appels de

```
type expr =
| V                  (* variable *)
| C of int           (* constant *)
| S of expr * expr   (* sum *)
| P of expr * expr   (* product *)

let rec eval v e =
  match e with
  | V ->
      v
  | C c ->
      c
  | S (e1, e2) ->
      eval v e1 + eval v e2
  | P (e1, e2) ->
      eval v e1 * eval v e2

let rec derivative e =
  match e with
  | V ->
      C 1
  | C _ ->
      C 0
  | S (e1, e2) ->
      S (derivative e1, derivative e2)
  | P (e1, e2) ->
      S (P (derivative e1, e2), P (e1, derivative e2))
```

FIGURE 4.6 – Expressions symboliques

fonction ; allocation d'un bloc de mémoire via `::`) est $O(1)$. Le coût total de la traversée d'un arbre de taille N est donc $O(N)$.

Enfin, il est facile de donner une nouvelle définition de `elements` :

```
let elements t =
  elements_append t []
```

À nouveau, grâce à l'équation `elements_append t tail = elements t @ tail`, la correction de cette définition est évidente.

Dans la définition de `elements_append`, le paramètre `tail` est traditionnellement appelé **accumulateur**. En effet, il sert à accumuler les éléments `x` rencontrés pendant la traversée de l'arbre. Cette accumulation est efficace parce qu'elle utilise uniquement l'ajout d'un élément en tête de liste `::` et non la concaténation `@`. ◇

Solution de l'exercice 1.6 Une solution est donnée dans la Figure 4.6. Le type somme de l'énoncé se transcrit aisément sous forme d'un type algébrique `expr`. La fonction `eval v e`, qui calcule la valeur numérique de l'expression `e` sous l'hypothèse que la variable a la valeur `v`, s'écrit sous forme récursive, avec analyse de la forme de l'expression `e`. La fonction `derivative e`, qui calcule (sous forme symbolique) la dérivée de l'expression `e`, s'écrit sous forme analogue. Notons que la complexité de cette fonction n'est pas linéaire : cela provient de la branche `P`, où `e1` et `e2` apparaissent chacun deux fois dans le membre droit. Sans résoudre ce problème, on pourrait améliorer légèrement la situation en utilisant dans les membres droits, à la place des constructeurs `S` et `P`, des fonctions auxiliaires `sum` et `product` qui appliqueraient au vol certaines règles de simplification : $e + 0 = e$, $e \times 0 = 0$, $e \times 1 = e$, etc. On peut définir une fonction `derivative` de complexité linéaire, mais il faut une réflexion plus poussée ; c'est ce que l'on appelle la différentiation automatique. ◇

Solution de l'exercice 1.7 Il est clair qu'un objet de classe `Counter` doit avoir deux champs, à savoir d'une part l'entier `step`, d'autre part la valeur actuelle du compteur, que nous pouvons appeler `value`. Les trois opérations requises sont implémentées par un constructeur et par deux méthodes `increment` et `get`. Le code est le suivant :

```
public class Counter {
  private final int step;
  private int value;
  public Counter (int step) { this.step = step; this.value = 0; }
  public void increment ()  { value += step; }
  public int get ()         { return value; }
}
```

Dans les méthodes `increment` et `get`, il faut comprendre que `value` et `step` désignent les champs `this.value` et `this.step`. Ces méthodes contiennent donc des instructions d'accès aux champs, en lecture et en écriture.

Les champs `step` et `value` sont privés. Cela permet de garantir que seuls le constructeur et les méthodes de cette classe y ont accès. On peut en déduire, sans même connaître le reste du programme, que certaines propriétés sont vraies : par exemple, le contenu actuel du champ `value` est toujours un multiple de `step`. (On ignore ici les problèmes d'« *overflow* ».)

Un objet de classe `Counter` est représenté en mémoire sous forme d'un bloc, alloué dans le tas, étiqueté par sa classe, `Counter`, et doté de deux champs, qui contiennent des entiers. ◇

Solution de l'exercice 1.8 La fonction `new_counter` doit produire un compteur, c'est-à-dire une paire de deux fonctions `increment` et `get`. Elle doit donc construire d'abord ces deux fonctions, puis construire un bloc de type `counter`, dans lequel on stocke (des pointeurs vers) ces deux fonctions.

Naturellement, il faut que ces deux fonctions aient accès à l'état interne du compteur, c'est-à-dire à sa valeur actuelle. Comme dans le cas de Java, il faut que cette valeur actuelle soit modifiable, et il faut qu'elle soit stockée dans un bloc de mémoire alloué dans le tas.

Le type des références, défini dans la bibliothèque d'OCaml et dont la définition est rappelée dans la solution de l'Exercice 1.2, convient pour cela. En bref, une référence est un bloc contenant un champ modifiable.

La fonction `new_counter` alloue donc d'abord une nouvelle référence, qui stocke l'état interne du compteur. L'adresse de cette référence est stockée dans une variable locale de la fonction `new_counter`, nommée ici `value`.

Cela fait, la fonction `new_counter` construit les deux fonctions `increment` et `get`. Leurs définitions sont imbriquées dans le corps de la fonction `new_counter`. Ceci fait que, tout naturellement, ces deux fonctions ont accès à la variable locale `value`, qui contient l'adresse du bloc de mémoire où est stockée la valeur actuelle du compteur. Elles peuvent donc modifier ou consulter cette valeur. Ces fonctions **ne sont pas closes**, au sens de la Remarque 1.15.

```
type counter =
  { increment: unit -> unit; get: unit -> int }

let new_counter (step : int) : counter =
  let value = ref 0 in
  let increment () =
    value := !value + step
  and get () =
    !value
  in
  { increment = increment; get = get }
```

Une fois ces deux fonctions construites, il reste à construire une paire de ces deux fonctions, c'est-à-dire un bloc dont le premier champ contient (un pointeur vers) la fonction `increment` et dont le deuxième champ contient (un pointeur vers) la fonction `get`.

Comme dans la version Java, l'état interne du compteur est encapsulé, c'est-à-dire inaccessible depuis l'extérieur, excepté via les fonctions `increment` et `get`. En effet, la valeur actuelle du compteur est stocké dans une référence, dont l'adresse est contenue dans la variable `value`. Cette variable est locale à la fonction `new_counter`, donc inaccessible de l'extérieur.

La variable `step`, qui est également locale à `new_counter`, est naturellement accessible aux fonctions `increment` et `get`, qui sont imbriquées dans `new_counter`. Ainsi, nous pouvons utiliser la variable `step`, sans même avoir besoin de la copier explicitement vers un champ immuable du même nom, comme c'est le cas en Java (Exercice 1.7).

Ce fragment de code s'appuie de façon cruciale sur le fait qu'**une fonction d'OCaml est une valeur** (ainsi, une fonction peut être stockée dans un champ d'une paire) et sur le fait qu'**une fonction d'OCaml a accès aux variables locales des fonctions englobantes**. Ces mécanismes, qui peuvent sembler mystérieux à première vue, sont expliqués dans les deux exercices qui suivent. ◇

Solution de l'exercice 1.9 La méthode statique `newCell` est extrêmement simple : elle crée un nouvel objet de classe `Cell`. Cette classe, que l'on définit séparément, implémente l'interface `Function<Integer, Integer>`.

```
public class CellFactory {
  public static Function<Integer, Integer> newCell () {
    return new Cell ();
  }
}
class Cell implements Function<Integer, Integer> {
  private int value = 0;
  public Integer apply (Integer newValue) {
    int oldValue = value;
    value = newValue;
    return oldValue;
  }
}
```

Un objet de classe `Cell` est doté d'un champ modifiable `value`, qui représente le contenu actuel de la cellule. La méthode `apply` définit le comportement de la cellule. Comme la fonction `exchange` de l'énoncé, elle stocke son argument `newValue` dans la cellule, et renvoie pour résultat l'ancienne valeur `oldValue` de la cellule.

On notera que, en Java, les conversions entre le type primitif `int` et le type d'objet `Integer` sont automatiques et implicites.

On pourrait souhaiter rendre la classe `Cell` privée, de façon à être certain que seule la méthode `newCell` peut créer des objets de cette classe. Pour cela, il faut déplacer la définition de la classe `Cell` dans la classe `CellFactory`, et marquer `Cell` à l'aide des mots-clef **`static`** et **`private`**.

Une autre façon d'obtenir cet effet consiste à remplacer la classe `Cell` par une **classe anonyme**. On écrit dans ce cas :

```
public class CellFactory2 {
  public static Function<Integer, Integer> newCell () {
    return new Function<Integer, Integer> () {
      private int value = 0;
      public Integer apply (Integer newValue) {
        int oldValue = value;
        value = newValue;
        return oldValue;
      }
    };
  }
}
```

```
import java.util.function.Function;
public class Counter implements Function<Unit, Integer> {
  private int value = 0;
  private final int step;
  public Counter (int step) { this.step = step; }
  @Override public Integer apply (Unit u) { // getAndIncrement
    int c = value;
    value = c + step;
    return c;
  }
}
```

FIGURE 4.7 – Un compteur doté d'une seule méthode

L'expression `new Function<...> () { ... }` est légale, et ce bien que `Function` soit une interface et non une classe, parce qu'on fournit immédiatement une définition de la méthode `apply`. Cette version du code est équivalente à la précédente : lorsqu'on soumet ce code au compilateur Java, il le traduit vers la forme précédente, en choisissant un nom arbitraire, par exemple `CellFactory2$1`, pour la classe que nous avions nommée `Cell` dans la version précédente du code.

Quelle que soit l'écriture du code, un objet « cellule » a un champ, à savoir le champ `value`. Il est donc représenté en mémoire comme un bloc doté d'une étiquette (sa classe, ici `Cell` ou `CellFactory2$1`) et d'un champ qui contient un entier. L'objet allie données et comportement : le champ `value` contient une donnée, et l'étiquette indique la classe à laquelle appartient l'objet, donc détermine son comportement. ◇

Solution de l'exercice 1.10 La solution est donnée dans la Figure 4.7. Un objet de classe `Counter` a deux champs `value` et `step`, et est tout à fait analogue à la clôture que le compilateur OCaml alloue pour représenter la fonction `get_and_increment` de l'énoncé. On voit aussi que l'expression `new Counter (step)` en Java est analogue à l'appel de fonction `new_counter step` en OCaml. ◇

Solution de l'exercice 1.11 Remarquons tout d'abord que la fonction identité admet le type souhaité, à savoir `(unit -> 'a) -> 'a thunk`, mais n'est bien sûr pas une solution correcte, puisque ce qui est demandé ici est de construire un mécanisme de mémoisation.

L'idée est de construire une nouvelle fonction, laquelle a accès à un état interne modifiable. Soit cet état indique que le calcul n'a pas encore lieu ; soit il indique que le calcul a déjà eu lieu, et en stocke le résultat. Un état interne de type `'a option ref` convient : c'est un état modifiable et qui contient ou non une valeur de type `'a`. On écrit donc :

```
let thunk (f : unit -> 'a) : 'a thunk =
  let state = ref None in
  fun () ->
    match !state with
    | Some a ->
        a
    | None ->
        let a = f() in
        state := Some a;
        a
```

Lorsque la fonction `thunk f` est dans le but de construire une suspension, elle alloue une nouvelle référence modifiable, `state`, initialisée à la valeur `None`. Puis, elle construit et renvoie une fonction anonyme `fun () -> ...` qui a accès à `state`. Cette fonction anonyme est une suspension. À chaque fois qu'elle est appelée, elle consulte le contenu de la référence `state`. Si son contenu est `Some a`, alors le résultat `a` du calcul est déjà disponible, et il suffit de le

renvoyer. Si au contraire son contenu est `None`, alors il faut effectuer le calcul en appelant la fonction `f`, puis stocker son résultat `a` dans la référence `state`, de façon à pouvoir le réutiliser à l'avenir.

On peut se convaincre expérimentalement que ce code semble fonctionner, par exemple en définissant une fonction « factorielle bavarde », puis en construisant une suspension et en l'évaluant plusieurs fois :

```
let rec fact n =
  Printf.printf "Je calcule fact(%d)...\n%!" n;
  if n = 0 then 1 else n * fact (n-1)
;;
let t = thunk (fun () -> fact 5);;
t();;
t();;
```

On peut coller ce code dans l'évaluateur interactif `ocaml` et observer que le calcul de 5! n'est effectué qu'une fois.

Toutefois, **la fonction `thunk` ci-dessus est fausse**, pour de multiples raisons. Ce code ne garantit pas, comme on le souhaiterait, que le calcul est effectué au plus une fois. En d'autres termes, si l'on a défini un thunk `t` en posant `let t = thunk f`, et si l'on appelle plusieurs fois la fonction `t`, il se peut, dans certaines circonstances, que la fonction `f` soit appelée plusieurs fois. Pouvez-vous imaginer quelles sont ces circonstances ? Essayez de prouver que `f` ne peut être appelée qu'une fois, et peut-être verrez-vous où la preuve échoue. Pouvez-vous imaginer comment corriger le code pour éviter ces problèmes ?

On peut citer au moins trois types de problèmes, mais tous ont une racine commune, à savoir le fait qu'il s'écoule un certain temps entre le moment où on constate que la référence `state` contient `None` et le moment où on y écrit la valeur `Some a`. Pendant ce temps, d'autres appels à la suspension `t` peuvent avoir lieu, et comme ils trouvent la valeur `None` dans la référence, ils provoquent à nouveau un appel à `f`.

Voici un premier exemple, certes alambiqué, où on joue sur le fait que la fonction `f` peut très bien posséder un pointeur sur le thunk `t` (et ce, malgré le fait que `t` a été construit après `f` !) et provoquer un deuxième appel à `t` alors que le premier appel à `t` n'est pas terminé. Cet exemple est une variante du « nœud de Landin ».

```
let self =
  ref (fun () -> ());;
let t = thunk (fun () ->
  Printf.printf "I must be called at most once.\n%!";
  !self()
);;
self := t;;
t();;
```

En principe, si l'implémentation des suspensions est correcte, le message « `I must be called at most once.` » ne devrait être affiché qu'une fois. Or, si on colle ce code dans l'évaluateur interactif `ocaml`, on observe que le message est affiché un grand nombre de fois, puis l'exécution s'arrête par manque d'espace sur la pile. L'évaluateur affiche « `Stack overflow during evaluation (looping recursion?).` »

Pour corriger ce problème, on peut utiliser un état interne à trois valeurs possibles, et non deux : soit le calcul n'a pas encore été demandé, soit le calcul est en cours, soit le calcul est terminé. Si un appel à la suspension a lieu alors que le calcul est déjà en cours, il faut échouer. L'écriture de cette version est laissée en exercice au lecteur.

Un second problème provient du fait que la fonction `f` peut très bien lancer une exception. Dans ce cas, l'instruction `state := Some a` n'est pas atteinte, et la suspension reste dans son état initial, bien que la fonction `f` ait été appelée. De ce fait, rien n'empêche `f` d'être appelée une seconde fois. Pour corriger ce problème, il faut rattraper une éventuelle exception issue de l'appel à `f`.

Un troisième problème provient de la concurrence, qui existe en Java et (avec certaines restrictions) en OCaml. Si deux threads ont accès à la suspension `t`, alors ils peuvent en demander simultanément la valeur. Il a donc une « *race condition* » sur le bloc de mémoire `state`, et à nouveau, la fonction `f` risque d'être exécutée deux fois, simultanément, par les deux threads. Pour corriger ce problème, il faut employer un verrou.

Comme on le voit, une implémentation correcte des suspensions n'est pas entièrement triviale. C'est pour cette raison (et également pour des raisons d'efficacité) qu'OCaml propose une implémentation primitive des suspensions. La bibliothèque de Java ne définit pas de classe `Thunk` ; nous en donnons une (à ma connaissance correcte) dans la solution de l'exercice suivant. ◇

Solution de l'exercice 1.12 Une solution est donnée dans la Figure 4.8. Le principe de cette solution est le même que dans l'exercice précédent.

Nous évitons le premier des trois problèmes mentionnés précédemment en utilisant trois états `UNEVALUATED`, `EVALUATING` et `EVALUATED` au lieu de deux. Si une suspension exige sa propre valeur, une exception est lancée ; cette situation est considérée comme anormale.

Nous évitons le second problème en rattrapant une éventuelle exception lancée par le calcul `f`. Nous utilisons deux champs, `error` et `result`, pour mémoriser le résultat du calcul `f`. Si `error` est non-`null`, c'est que `f` a lancé une exception ; sinon, c'est que `f` a renvoyé la valeur `result`. Nous adoptons ici la convention que, si `f` a lancé une exception, alors à chaque fois que la valeur de cette suspension sera demandée, nous lançons à nouveau cette exception.

Nous évitons le troisième problème en introduisant un verrou, dont la présence est indiquée par le mot-clef `synchronized`. Ce verrou est implicitement acquis au début de l'exécution de la méthode `call`, et relâché lorsque cette méthode rend la main. De ce fait, deux threads ne peuvent pas interférer : ils doivent accéder à la suspension l'un après l'autre. ◇

Solution de l'exercice 2.1 La bibliothèque de Java ne fournit pas de classe représentant une liste chaînée immuable. Il est très facile d'en définir une, spécialisée ici au cas où les éléments sont des entiers :

```
public class IntList {
  public final int  element;
  public final IntList next;
  public IntList (int element, IntList next) {
    this.element = element;
    this.next = next;
  }
}
```

On adopte la convention que la liste vide est représentée par le pointeur `null`. Ce raccourci nous permet de représenter les listes à l'aide d'une seule classe, à savoir `IntList`, alors que sans cela, puisque le type des listes est un type somme à deux branches, nous aurions dû introduire une classe abstraite et deux sous-classes, dans le style de §1.1.

La classe `IntList` n'a pas de méthodes. Ses champs sont publics. Nous construisons des objets de classe `IntList`, et accédons à leurs champs, depuis les méthodes de la classe `IntStack`, dont voici la définition.

```
public class IntStack {
  protected IntList top = null;
  public void push (int x)
  {
    top = new IntList (x, top);
  }
  public int pop () throws NoSuchElementException
  {
    if (top == null)
      throw new NoSuchElementException ();
```

```
import java.util.concurrent.Callable;

enum ThunkStatus {
  UNEVALUATED,
  EVALUATING,
  EVALUATED
}

public class Thunk<T> implements Callable<T> {

  // The computation that we are supposed to perform (at most once).
  private final Callable<T> f;
  // Has the computation been performed already?
  private ThunkStatus status;
  // If so, what is its outcome (an exception or a value)?
  private Exception error;
  private T result;

  public Thunk (Callable<T> f) {
    this.f = f;
    this.status = ThunkStatus.UNEVALUATED;
  }

  public synchronized T call () throws Exception {
    switch (status) {
    case EVALUATING:
      throw new Error ("Thunk demands its own result!");
    case UNEVALUATED:
      status = ThunkStatus.EVALUATING;
      try {
        result = f.call();
      } catch (Exception e) {
        error = e;
      }
      status = ThunkStatus.EVALUATED;
      // Fall-through to the next case.
    default:
      if (error != null) throw error; else return result;
    }
  }

}
```

FIGURE 4.8 – Une implémentation des suspensions

```
      else {
        int x = top.element;
        top = top.next;
        return x;
      }
    }
  }
```

Un objet de classe `IntStack` contient un champ modifiable `top` de type `IntList`. Ce champ contient un pointeur vers la tête de la liste chaînée, c'est-à-dire vers le sommet de la pile. Il doit en principe être privé, ce qui garantit que les méthodes `push` et `pop` sont les seules à pouvoir le consulter ou le modifier. Nous préférons ici le déclarer **`protected`**, en prévision de l'Exercice 2.2, où nous définirons une sous-classe de la classe `IntStack`. Le code situé dans la sous-classe ne pourrait pas accéder au champ `top` s'il était déclaré **`private`**.

En comparaison avec l'Exercice 2.5, notons que la représentation des piles en mémoire est la même dans les deux langages : dans les deux cas, une pile est représentée par une référence (c'est-à-dire un objet à un champ modifiable) vers une liste chaînée immuable.

La méthode `push` crée une nouvelle cellule et l'insère en tête de liste.

La méthode `pop` renvoie l'élément `x` contenu dans la cellule située en tête de liste. Elle supprime cette cellule de la liste, en faisant pointer `top` vers la cellule suivante.

La méthode `pop` ne peut pas extraire un élément de la pile lorsque la pile est vide. Il faut décider de quelle façon cette situation doit être signalée au client. En OCaml, la fonction `push` renvoyait un résultat de type `option int`. En Java, on pourrait, de même, produire un résultat de type `Option<Integer>`, pour une classe `Option<X>` définie de façon appropriée. On pourrait aussi produire un résultat de type `Integer`, et utiliser le pointeur **`null`** pour signaler que la pile est vide. (Rappelons que la classe `Integer`, définie dans la bibliothèque de Java, décrit un objet à seul champ immuable de type **`int`**. Le pointeur **`null`** n'a pas le type **`int`** car ce n'est pas un entier, mais il a le type `Integer`, car le pointeur **`null`** appartient à tous les types d'objets.) Nous choisissons ici une troisième solution, qui consiste à produire un résultat du type primitif **`int`**, mais à lancer une exception lorsque la pile est vide. Nous utilisons l'exception `NoSuchElementException` définie dans la bibliothèque de Java. ◇

Solution de l'exercice 2.2 D'un point de vue algorithmique, l'exercice ne pose aucune difficulté. Tout au plus devons-nous introduire un champ supplémentaire pour garder trace du nombre d'éléments contenus dans la pile, ce qui nous permet d'implémenter la méthode `size` en temps $O(1)$.

Commençons par la solution la plus naïve, où l'on recopie la code de la classe `IntStack` et où on lui apporte des modifications. Cette solution apparaît dans la Figure 4.9. Le champ `size`, incrémenté par `push` et décrémenté par `pop`, permet d'implémenter la méthode `size`. L'ajout de la méthode `peek` ne pose pas de difficulté. En général, **dupliquer le code** d'une classe existante pour y apporter quelques améliorations **n'est pas une bonne idée** ! Cela augmente la charge de travail de celui qui doit maintenir les classes `IntStack` et `RichIntStack`.

L'**héritage** est censé justement permettre à l'auteur de la classe `RichIntStack` de s'appuyer sur la classe `IntStack`, sans devoir modifier ni recopier le code de cette dernière. Cette solution apparaît dans la Figure 4.10. La clause **`extends`** `IntStack` indique que l'on définit ici une sous-classe de `IntStack`, donc que l'on souhaite hériter de tous les éléments (champs, méthodes) définis dans cette dernière. On ajoute ensuite un champ `size`, et on redéfinit les méthodes `push` et `pop` de façon à mettre à jour ce champ. L'annotation `@Override` souligne le fait qu'il s'agit bien de redéfinitions de méthodes héritées, et non pas de définitions de nouvelles méthodes. La nouvelle définition de la méthode `push` fait appel à l'ancienne via la syntaxe **`super`**`.push(x)`. Un phénomène similaire a lieu dans `pop`, où on prend soin de décrémenter le champ `size` après l'appel à **`super`**`.pop()`, et non pas avant. Ainsi, si la pile est vide, l'appel **`super`**`.pop()` lance une exception, et le champ `size` reste inchangé. Enfin, la méthode `peek` est implémentée en consultant directement le champ `top`, qui est hérité de la classe `RichInt`. L'accès à ce champ est permis car il a été déclaré **`protected`** et non pas **`private`**.

```
public class RichIntStack {
  private IntList top = null;
  private int size = 0;
  public void push (int x)
  {
    top = new IntList (x, top);
    size++;
  }
  public int pop () throws NoSuchElementException
  {
    if (top == null)
      throw new NoSuchElementException ();
    else {
      int x = top.element;
      top = top.next;
      size--;
      return x;
    }
  }
  public int peek () throws NoSuchElementException
  {
    if (top == null)
      throw new NoSuchElementException ();
    else
      return top.element;
  }
  public int size ()
  {
    return size;
  }
}
```

FIGURE 4.9 – Implémentation directe de `RichIntStack`

```
public class RichIntStackInheritance extends IntStack {
  private int size = 0;
  @Override public void push (int x)
  {
    super.push(x);
    size++;
  }
  @Override public int pop () throws NoSuchElementException
  {
    int x = super.pop(); // may throw an exception
    size--;
    return x;
  }
  public int peek () throws NoSuchElementException
  {
    if (top == null)
      throw new NoSuchElementException ();
    else
      return top.element;
  }
  public int size ()
  {
    return size;
  }
}
```

FIGURE 4.10 – Implémentation de `RichIntStack` par héritage

En résumé, l'approche fondée sur l'héritage offre l'intérêt de ne pas provoquer de duplication de code. Toutefois, les classes `IntStack` et `RichIntStack` ne sont pas vraiment deux abstractions indépendantes : si on modifiait profondément `IntStack`, et si cela provoquait par exemple la disparition du champ `top`, alors il faudrait adapter `RichIntStack` en conséquence, car le code de la méthode `peek` n'aurait plus de sens.

La **délégation** consiste à implémenter `RichIntStack` en utilisant un objet de type `IntStack`. Dans cette approche, on n'utilise pas l'héritage : la nouvelle classe est définie indépendamment de la première. L'auteur de la classe `RichIntStack` respecte alors l'abstraction `IntStack` : il n'a pas besoin de savoir comment `IntStack` est implémentée (et il n'a pas accès à ses champs ou méthodes privés). Cette solution apparaît dans la Figure 4.11. Un objet de classe `RichIntStack` possède un champ `stack`, qui contient (un pointeur vers) un objet de classe `IntStack`. De ce fait, les appels à **`super`** de la Figure 4.10 sont remplacés par des appels à `stack`. Dans la méthode `peek`, on rencontre une difficulté. On aimerait (peut-être) remplacer l'accès au champ `top` par un accès à `stack.top`, mais celui-ci est interdit, car il viole l'abstraction : ce champ est privé (ou protégé). Ici, on résoud ce problème en effectuant un appel à `pop` suivi d'un appel à `push`. Cette solution n'est pas très élégante ni très efficace. C'est le signe que, probablement, la méthode `peek` devrait faire partie de la classe `IntStack`, car il n'est pas facile de l'ajouter a posteriori.

En résumé, l'approche fondée sur la délégation ne provoque pas de duplication de code, et de plus respecte l'abstraction `IntStack`. De fait, au de lieu de construire la classe `RichIntStack` au-dessus de `IntStack` spécifiquement, on pourrait la construire au-dessus de n'importe quelle implémentation des piles. Cela sera le sujet de l'Exercice 3.19.

Un avantage potentiel de l'approche fondée sur l'héritage est qu'elle conduit à une relation de **sous-typage** entre `RichIntStack` et `IntStack` : un objet de type `RichIntStack` **est** un objet de type `IntStack`. Un autre avantage est que `RichIntStack` hérite automatiquement de toutes les

```
public class RichIntStackDelegation {
  private IntStack stack = new IntStack ();
  private int size = 0;
  public void push (int x)
  {
    stack.push(x);
    size++;
  }
  public int pop () throws NoSuchElementException
  {
    int x = stack.pop(); // may throw an exception
    size--;
    return x;
  }
  public int peek () throws NoSuchElementException
  {
    int x = stack.pop(); // not great!
    stack.push(x);       // argh!
    return x;
  }
  public int size ()
  {
    return size;
  }
}
```

FIGURE 4.11 – Implémentation de `RichIntStack` par délégation

méthodes de `IntStack`, y compris `equals`, `hashCode`, `toString`, etc. Enfin, parce que l'héritage ne respecte pas l'abstraction (i.e., il donne accès aux éléments marqués **`protected`**), il offre une grande flexibilité. Cependant, Java n'autorise que l'héritage simple : on ne peut hériter que d'une seule classe. Cela limite les situations où cette approche est utilisable.

La délégation ne souffre pas de cette limitation. De plus, parce qu'elle respecte l'abstraction, elle est souvent conceptuellement plus simple. Elle ne donne pas de relation de sous-typage entre `RichIntStack` et `IntStack`, mais cela n'est pas forcément gênant. Pour compenser cela, on peut définir une interface `IIntStack`, implémentée par ces deux classes (§2.2). Dans ce cas, on a deux relations de sous-typage : un objet de type `RichIntStack` **est** un objet de type `IIntStack`, et un objet de type `IntStack` **est** aussi un objet de type `IIntStack`. ◇

Solution de l'exercice 2.3 Comme la classe `ArrayList` nous fournit un tableau extensible, doté de toutes les méthodes dont nous avons besoin, l'exercice est facile. (On pourrait aussi utiliser `Vector`, qui est presque identique à `ArrayList`. La principale différence entre ces deux classes est qu'un objet de classe `Vector` est protégé par un verrou, donc accessible sans danger par plusieurs « *threads* » simultanément.) Voici une solution :

```
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class IntArrayStack {
  private ArrayList<Integer> array = new ArrayList<Integer> ();
  public void push (int x) {
    array.add(x);
  }
  public int pop () throws NoSuchElementException {
    if (array.isEmpty())
      throw new NoSuchElementException ();
```

```
      else
        return array.remove(array.size() - 1);
    }
  }
```

Une approche légèrement différente utilise l'héritage :

```
public class IntArrayStack2 extends ArrayList<Integer> {
  public void push (int x) {
    add(x);
  }
  public int pop () throws NoSuchElementException {
    if (isEmpty())
      throw new NoSuchElementException ();
    else
      return remove(size() - 1);
  }
}
```

Dans la première variante, une pile **contient** (un pointeur vers) un tableau extensible, tandis que la seconde variante, une pile **est** un tableau extensible. On parle parfois de **délégation** dans le premier cas, et d'**héritage** dans le second. Ici, seule la première approche est réellement satisfaisante, au sens où elle définit bien un type abstrait, doté de deux méthodes seulement, à savoir `push` et `pop`. La seconde approche ne définit pas un type abstrait, car le fait que `IntArrayStack2` est une sous-classe de `ArrayList<Integer>` est connu du client, et de ce fait, le client a accès non seulement aux méthodes `push` et `pop`, mais aussi à toutes les méthodes héritées de la classe `ArrayList`. ◇

Solution de l'exercice 2.4 La signature, contenue dans le fichier `Counter.mli`, doit déclarer l'existence d'un type abstrait `counter` ainsi que des trois fonctions nommées dans l'énoncé. La signature est donc :

```
type counter
val make: int -> counter
val increment: counter -> unit
val get: counter -> int
```

Vis-à-vis de l'approche fondée sur les clôtures (§1.2.2), le point de vue a changé. Un compteur **était** une paire de fonctions `increment` et `get`, qui n'attendaient aucun argument, parce qu'elles avaient accès à l'état interne du compteur. Ici, un compteur n'est pas une fonction ni une paire de fonctions. C'est une entité abstraite, dont la nature exacte n'est pas connue, à laquelle on peut **appliquer** les opérations `increment` et `get`. Les fonctions `increment` et `get` sont donc définies une fois pour toutes (il y en a deux en tout, et non pas deux pour chaque compteur), mais elles attendent un argument, à savoir le compteur auquel on souhaite les appliquer.

L'implémentation, contenue dans le fichier `Counter.ml`, ne doit pas poser de difficulté :

```
type counter =
  { step: int; mutable value: int }
let make (step : int) =
  { step = step; value = 0 }
let increment c =
  c.value <- c.value + c.step
let get c =
  c.value
```

Comme dans l'Exercice 1.7, un compteur est un bloc de mémoire doté d'un champ immuable `step` et d'un champ modifiable `value`. ◇

Solution de l'exercice 2.5 Le type `int list` représente en OCaml une liste chaînée, immuable, dont les éléments sont des entiers. Comme une pile doit être modifiable, une pile ne peut pas

être simplement une liste. On pose donc qu'une pile est une référence sur une liste immuable d'entiers. Rappelons qu'une référence est un objet modifiable à un champ (voir l'Exercice 1.2 et sa solution). Le type `stack` est donc défini comme un synonyme pour le type `(int list) ref`, que l'on peut écrire plus brièvement `int list ref`. Le contenu du fichier `Stack.ml` est alors le suivant :

```
type stack =
  int list ref

let create () =
  ref []

let push x s =
  s := x :: !s

let pop s =
  match !s with
  | [] ->
      None
  | x :: xs ->
      s := xs;
      Some x
```

La fonction `create` appelle la fonction `ref`, qui alloue une nouvelle référence, une nouvelle « boîte » modifiable si l'on veut, dont le contenu initial est la liste vide `[]`.

La fonction `push` utilise la fonction de lecture `!` (une fonction à un argument) pour obtenir le contenu actuel de la référence `s` ; l'expression `!s` produit donc une liste d'entiers. On construit alors la liste `x :: !s`, dont on écrit enfin l'adresse dans la référence `s`, à l'aide de la fonction d'écriture `:=` (une fonction à deux arguments).

La fonction `pop` utilise elle aussi la fonction de lecture `!` pour obtenir le contenu actuel de la référence `s`. Elle utilise ensuite une construction **`match`** pour effectuer une analyse de cas. On distingue deux cas, selon que la liste est vide ou non. Dans le premier cas, il est impossible d'extraire un élément : `pop` renvoie donc `None` et ne modifie pas la pile. Dans le second cas, où la liste est de la forme `x :: xs`, on souhaite extraire l'élément `x`. On renvoie donc `Some x`, après avoir modifié la pile à l'aide de l'instruction `s := xs`.

À peu de chose près, c'est ainsi qu'est implémenté le module `Stack`, qui fait partie de la bibliothèque d'OCaml. Je vous encourage à consulter la signature et l'implémentation de ce module, qui sont disponibles en ligne. ◇

Solution de l'exercice 2.6 La Figure 4.12 propose une solution. La bibliothèque d'OCaml n'offre pas de tableaux redimensionnables, aussi il faut implémenter soi-même le mécanisme de redimensionnement. Celui-ci est géré par la fonction `enlarge_if_needed`, qui garantit que l'entier `s.limit` est un indice valide dans le tableau `s.content`. Afin d'obtenir une bonne complexité amortie, on double la taille du tableau à chaque fois qu'on doit l'agrandir ; voir par exemple le polycopié INF411 (Filliâtre, 2014, §3.4). On pourrait également souhaiter faire diminuer la taille du tableau lorsque la pile se vide ; ce n'est pas fait ici.

On crée initialement un tableau de taille `default_initial_length`, dont la valeur a été fixée arbitrairement à 256. On remplit les cases inutilisées à l'aide d'un entier `dummy`, dont la valeur a été fixée arbitrairement à 0.

Notons que l'accès à un tableau en lecture s'écrit `t.(i)` et que l'accès en écriture s'écrit `t.(i) <- x`. On peut écrire cela, de façon équivalente, sous forme d'appels de fonction, `Array.get t i` et `Array.set t i x`. ◇

Solution de l'exercice 2.7 Les implémentations à base de listes chaînées et à base de tableaux redimensionnables n'ont pas exactement les mêmes performances en temps et en espace.

Pour ce qui est du temps, les deux implémentations ont la même complexité **asymptotique** et **amortie**, c'est-à-dire que, à une constante près et lorsque l'on considère une séquence de

```
type stack = {
  mutable limit:   int;
  mutable content: int array
}

let default_initial_length = 256

let dummy = 0

let create () = {
  limit = 0;
  content = Array.make default_initial_length dummy
}

let enlarge_if_needed s =
  let content = s.content in
  let length = Array.length content in
  (* Is the array full? *)
  if s.limit = length then begin
    (* If so, create a new array, twice larger... *)
    let content' = Array.make (2 * length) dummy in
    (* ...copy the existing array into it... *)
    Array.blit content 0 content' 0 length;
    (* ...and make it our new array. *)
    s.content <- content'
  end

let push x s =
  enlarge_if_needed s;
  let i = s.limit in
  s.limit <- i + 1;
  s.content.(i) <- x

let pop s =
  let i = s.limit in
  if i = 0 then
    None
  else begin
    s.limit <- i - 1;
    Some s.content.(i - 1)
  end
```

FIGURE 4.12 – Une pile implémentée à l'aide d'un tableau redimensionnable (Exercice 2.6)

N opérations `push` et `pop`, les deux implémentations ont une complexité N. Cependant, en pratique, les deux implémentations diffèrent. Dans l'implémentation à base de listes, chacune des trois opérations a dans le pire cas un coût très faible, car elle ne demande qu'un nombre constant de lectures, d'écritures, et d'allocations dynamiques de mémoire. (Voyez-vous les allocations dynamiques de mémoire ? Il y en a une dans `create`, une dans `push`, et une dans `pop`.) Dans l'implémentation à base de tableaux, `create` est coûteuse car elle alloue un tableau assez gros (ici, de 256 cases), qu'il faut initialiser ; et `push` est occasionnellement très coûteuse car il arrive qu'elle alloue un nouveau tableau.

Pour ce qui est de l'espace, les deux implémentations ont à nouveau la même complexité asymptotique, à savoir $O(n)$ où n est la taille de la pile. Cependant, la représentation à base de tableau peut sembler préférable, et ce pour plusieurs raisons. D'abord, elle est plus compacte. Elle alloue un seul objet en mémoire (le tableau) au lieu de n objets (les cellules de la liste chaînée). Elle demande donc en général moins de mémoire, et donne moins de travail au « *garbage collector* », d'où un gain de temps. Ensuite, un tableau a une meilleure **localité** qu'une liste. Dans un tableau, les éléments sont stockés les uns à côté des autres en mémoire, tandis que dans une liste chaînée, ils peuvent être stockés à des adresses arbitraires. Or, sur un processeur moderne, la **hiérarchie mémoire**, c'est-à-dire l'organisation de la mémoire en plusieurs niveaux de **caches**, fait que des accès à des éléments consécutifs en mémoire sont beaucoup plus rapides que des accès à des adresses arbitraires.

Pour trouver un compromis entre ces deux approches, une idée naturelle est de représenter une pile en mémoire comme une liste de tableaux de taille fixe, ou « *chunks* ». On obtient ainsi une partie des avantages des tableaux (`push` et `pop` sont rapides, tant qu'on travaille dans un « *chunk* » qui n'est ni vide ni plein ; la compacité et la localité sont relativement bonnes) et une partie des avantages des listes (le coût de `push`, dans le pire cas, est borné par la taille d'un « *chunk* »). On peut faire varier la taille des « *chunks* » pour rechercher le meilleur compromis. On pourrait aussi s'autoriser à créer des « *chunks* » de différentes tailles, mais ce n'est pas fait dans le code que nous proposons ici.

Le code est donné par les Figures 4.13 et 4.14. Dans la Figure 4.13, nous définissons un module `BoundedIntStack`, qui implémente une pile de taille bornée à l'aide de tableau de taille fixe. Nous munissons cette pile d'une fonction `is_full`, et nous adoptons la convention que `push` ne peut être appelée que si la pile n'est pas pleine. Ensuite, dans la Figure 4.14, nous implémentons une pile de taille non bornée comme une référence sur une liste de « *chunks* », où un « *chunk* » est tout simplement une pile de taille bornée, fournie par le module `BoundedIntStack`. Nous obtenons ainsi une implémentation très simple, clairement découpée en deux modules, le second s'appuyant sur le premier. ◇

Solution de l'exercice 2.8 La signature ne doit pas révéler comment la file d'attente est implémentée ; il faut donc que le type `'a queue` soit déclaré comme un type abstrait. Alex a probablement écrit la signature suivante :

```
type 'a queue
val create: unit -> 'a queue
val insert: 'a queue -> 'a -> unit
val extract: 'a queue -> 'a option
```

Le code d'Alex est mauvais car l'insertion en queue de liste, notée ici `!q @ [x]`, a un coût linéaire vis-à-vis de la longueur de la liste. (Le symbole `@` dénote la fonction de concaténation de deux listes, dont le coût est linéaire vis-à-vis de la longueur de la première liste.) Or, de la part d'une file d'attente, on est en droit d'attendre insertion et extraction en temps $O(1)$.

On peut atteindre la complexité souhaitée tout en conservant une structure de liste simplement chaînée. Pour cela, on conserve un pointeur `q.tail` vers la dernière cellule de la liste (lorsque celle-ci est non vide), et de plus, on rend les cellules modifiables. Pour insérer une nouvelle cellule `cell` en queue de liste, il suffit alors d'installer un lien de l'ancienne queue de liste vers la nouvelle, c'est-à-dire de `q.tail` vers `cell`. De plus, ensuite, il faut mettre à jour `q.tail` pour pointer vers `cell`. Cette approche est illustrée dans la Figure 4.15. Elle permet d'implémenter insertion et extraction en temps $O(1)$.

```
let size = 256

let dummy = 0

type stack = {
  (* The logical size of the stack; also, the next available address. *)
  mutable limit:   int;
  (* The array, whose length is [size]. *)
  mutable content: int array
}

let create () = {
  limit = 0;
  content = Array.make size dummy
}

let is_full s =
  s.limit = Array.length s.content

let push x s =
  assert (not (is_full s));
  let i = s.limit in
  s.limit <- i + 1;
  s.content.(i) <- x

let is_empty s =
  s.limit = 0

let pop s =
  assert (not (is_empty s));
  let i = s.limit in
  s.limit <- i - 1;
  Some s.content.(i - 1)
```

FIGURE 4.13 – Une pile bornée implémentée à l'aide d'un tableau (Exercice 2.7)

```
type chunk =
  BoundedIntStack.stack

type stack =
  chunk list ref

let create () : stack =
  ref []

let push_new_chunk (s : stack) : chunk =
  let chunk = BoundedIntStack.create() in
  s := chunk :: !s;
  chunk

let push (x : int) (s : stack) =
  (* Ensure there is a non-full chunk at the head of the list. *)
  let chunk =
    match !s with
    | [] ->
        push_new_chunk s
    | chunk :: _ ->
        if BoundedIntStack.is_full chunk then
          push_new_chunk s
        else
          chunk
  in
  (* Push [x] into it. *)
  BoundedIntStack.push x chunk

let rec pop (s : stack) : int option =
  match !s with
  | [] ->
      None
  | chunk :: chunks ->
      (* If there is an empty chunk at the head of the list,
         get rid of it. *)
      if BoundedIntStack.is_empty chunk then begin
        s := chunks;
        pop s
      end
      else
        BoundedIntStack.pop chunk
```

FIGURE 4.14 – Une pile implémentée à l'aide d'une liste de piles bornées (Exercice 2.7)

```
type 'a cell = {
  content: 'a;
  mutable next: 'a cell option
}

type 'a queue = {
  mutable head: 'a cell option;
  mutable tail: 'a cell option
}

let create () =
  { head = None; tail = None }

let insert q x =
  (* Create a new cell. *)
  let cell = { content = x; next = None } in
  (* Insert it at the end. *)
  match q.tail with
  | None ->
      assert (q.head = None);
      q.head <- Some cell;
      q.tail <- Some cell
  | Some tail ->
      assert (tail.next = None);
      tail.next <- Some cell;
      q.tail <- Some cell

let extract q =
  match q.head with
  | None ->
      None
  | Some head ->
      (* Remove the first cell. *)
      q.head <- head.next;
      (* The tail is unaffected, unless
         there was only one cell. *)
      if q.head = None then
        q.tail <- None;
      Some head.content
```

FIGURE 4.15 – Une file d'attente implémentée à l'aide d'une liste simplement chaînée modifiable

D'autres approches existent. On pourrait penser, en particulier, à employer une file d'attente stockée de façon circulaire dans un tableau redimensionnable. Quoiqu'il en soit, comme le type `'a queue` est abstrait, on peut à tout moment remplacer une implémentation par une autre, sans que les utilisateurs soient affectés. ◇

Solution de l'exercice 2.9 On pourrait implémenter `size` facilement à l'aide de la fonction `List.length`, mais sa complexité serait alors $O(n)$, où n est le nombre d'éléments de la pile. Mieux vaut tenir à jour le nombre d'éléments de la pile, de façon à pouvoir implémenter `size` avec une complexité $O(1)$. Voici une implémentation possible :

```
type stack =
  { mutable content: int list; mutable size: int }

let create () =
  { content = []; size = 0 }

let push x s =
  s.content <- x :: s.content;
  s.size <- 1 + s.size

let pop s =
  match s.content with
  | [] ->
      None
  | x :: xs ->
      s.content <- xs;
      s.size <- s.size - 1;
      Some x

let size s =
  s.size
```

De même, on pourrait implémenter l'opération `member` à l'aide de la fonction `List.mem`, qui effectue une recherche dans une liste simplement chaînée, mais sa complexité serait alors $O(n)$. Mieux vaut maintenir à jour le multi-ensemble des éléments de la pile, en utilisant par exemple une table de hachage, de façon à pouvoir implémenter `member` avec une complexité $O(1)$. De ce fait, notre structure de données n'est plus stricto sensu une pile, mais une structure hybride qui combine pile et multi-ensemble. Voici une implémentation possible :

```
let find table x =
  try Hashtbl.find table x with Not_found -> 0

let update table x m =
  if m = 0 then Hashtbl.remove table x
  else Hashtbl.replace table x m

type stack =
  { mutable content: int list;
    mutable members: (int, int) Hashtbl.t }

let create () =
  { content = []; members = Hashtbl.create 127 }

let push x s =
  s.content <- x :: s.content;
  update s.members x (find s.members x + 1)

let pop s =
  match s.content with
```

```
    | [] ->
        None
    | x :: xs ->
        s.content <- xs;
        update s.members x (find s.members x - 1);
        Some x

let member x s =
  Hashtbl.mem s.members x
```

On utilise une table de hachage `members` qui à chaque élément de la pile associe le nombre de fois où celui-ci apparaît dans la pile. (En effet, un même élément peut apparaître plusieurs fois.) On utilise pour cela le module `Hashtbl` de la bibliothèque d'OCaml. On se donne deux fonctions auxiliaires `find` et `update` pour consulter ou modifier une entrée de la table, tout en maintenant la convention qu'un élément apparaît dans la table si et seulement si il apparaît dans la pile. (La table n'associe donc jamais la valeur 0 à un élément.) Le reste du code s'écrit alors très facilement. La fonction `push` incrémente la multiplicité de `x`, tandis que `pop` la décrémente. La fonction `member` teste simplement si `x` apparaît dans la table. On peut se convaincre qu'il faut utiliser un multi-ensemble, c'est-à-dire une table de type `(int, int) Hashtbl.t`, car un simple ensemble, c'est-à-dire une table de type `(int, unit) Hashtbl.t`, ne suffirait pas : dans `pop`, lorsqu'on supprime un élément `x`, il nous faut un moyen de déterminer si la pile contient encore d'autres exemplaires du même élément `x`.

Le dernier point concerne un cas dégénéré. Voici une implémentation possible :

```
type stack = unit
let create () = ()
let push x () = ()
```

Si on supprime l'opération `pop`, les seules opérations restantes sont `create` et `push`. Dans ce cas, il n'existe aucun moyen pour l'utilisateur d'observer le contenu d'une pile. L'implémentation la plus efficace, dégénérée, consiste alors à ne stocker aucune information. ◇

Solution de l'exercice 2.10 On transcrit l'idée de l'Exercice 1.8 en JavaScript. L'objet `Counter` contient un champ modifiable `value`, accessible aux méthodes `get` et `increment`, et inaccessible à l'utilisateur, excepté via ces méthodes.

```
function Counter(step) {
  var value = 0
  this.get = function() { return value; }
  this.increment = function() { value += step; }
}
```

L'expression `new Counter (2)` produit un nouveau compteur pour lequel `step` est fixé à la valeur 2. Si `c` est un compteur, les appels de méthode s'écrivent `c.get()` et `c.increment()`. En revanche, le champ `value` est inaccessible ; une tentative d'évaluer `c.value` produit la valeur `undefined`. ◇

Solution de l'exercice 2.11 La solution complète est donnée dans la Figure 4.16.

Le constructeur se contente de stocker la fonction `successors` dans un champ privé du même nom, afin que la méthode `dfs` y aie accès.

La méthode `newStack` crée une nouvelle pile vide. On s'appuie ici sur la classe `IntStack` de l'Exercice 2.1, dont on suppose qu'elle implémente l'interface `IIntStack`. Seule la méthode `newStack` mentionne la classe `IntStack` : le reste du code ne s'appuie que sur l'interface `IIntStack`, et ignore donc quelle implémentation des piles a été choisie. De plus, la méthode `newStack` est déclarée `protected`, ce qui signifie qu'elle peut être redéfinie dans une sous-classe. Ainsi, sans modifier le code de la classe `DFS`, il est possible de choisir une autre implémentation des piles : il suffit pour cela de définir une sous-classe de `DFS`. La méthode `newStack` est parfois qualifiée de **factory method**, car elle sert uniquement à construire un objet.

```
public abstract class DFS {

  // The graph is described by a function that maps a node to
  // a sequence of its successors. (Nodes are integers.)

  private final Function<Integer,Iterable<Integer>> successors;

  public DFS (Function<Integer,Iterable<Integer>> successors) {
    this.successors = successors;
  }

  // Default choices of data structures for stacks and sets. These
  // could be overridden in a sub-class. We do not care which data
  // structures are used, since we rely on their interfaces.

  protected IIntStack newStack () {
    return new IntStack ();
  }

  protected Set<Integer> newSet () {
    return new HashSet<Integer> ();
  }

  // The algorithm.

  public void dfs (Iterable<Integer> roots) {
    // Initialize the stack and the set of visited nodes.
    IIntStack    stack = newStack();
    Set<Integer> visited = newSet();
    // Insert the roots into the stack.
    for (int root : roots)
      stack.push(root);
    // As long the stack is non-empty, process it.
    try {
      while (true) {
        int node = stack.pop();
        if (!visited.contains(node)) {
          visited.add(node);
          process(node);
          for (int successor : successors.apply(node))
            stack.push(successor);
        }
      }
    } catch (NoSuchElementException e) {
    }
  }

  // This method is invoked when a node is newly discovered.

  abstract protected void process (int node);

}
```

FIGURE 4.16 – Parcours en profondeur d'abord

La méthode `newSet` est construite sur le même schéma. Elle utilise la classe `HashSet`, qui implémente l'interface `Set`. Ces classe et interface font partie de la bibliothèque de Java.

Parce que l'on ne sait pas a priori quelle opération on souhaite effectuer à chaque fois qu'un sommet est découvert, on déclare la méthode `process` sans la définir : on la marque **`abstract`**. Elle devra donc être redéfinie dans une sous-classe. Parce qu'elle contient une méthode abstraite, la classe `DFS` doit elle-même être marquée **`abstract`**.

Tout ceci n'était qu'architecture et emballage. Il reste à écrire l'algorithme lui-même, qui prend place dans la méthode `dfs`. Cette méthode est déclarée **`public`**, car elle est conçue pour être appelée par l'utilisateur final. Elle utilise `newStack` et `newStack` pour construire la pile (qui contient les sommets en attente) ainsi que l'ensemble des sommets déjà traités. Elle utilise `successors.apply(node)` pour obtenir les successeurs d'un sommet. Cet objet a le type `Iterable<Integer>`, donc une boucle **`for (int successor : ...)`** permet d'en énumérer les éléments. ◇

Solution de l'exercice 3.1 La fonction `append` s'écrit sous forme récursive, avec analyse de la forme du premier argument :

```
let rec append xs ys =
  match xs with
  | []      -> ys
  | x :: xs -> x :: append xs ys
```

Les listes `xs` et `ys` doivent contenir des éléments de même type, sans quoi la liste `append xs ys` serait hétérogène (elle contiendrait des éléments de plusieurs types différents), ce qui n'est pas permis par la définition du type `'a list`. Le compilateur OCaml infère cette contrainte. Il n'existe pas d'autre contrainte : le type commun des éléments de `xs` et `ys` est arbitraire. La fonction `append` est donc polymorphe. Son type est `'a list -> 'a list -> 'a list`.

On notera que la définition ci-dessus alloue un espace $O(n)$ sur la pile, où n est la longueur de la liste, à cause des appels récursifs imbriqués. Parce que les systèmes d'exploitation actuels imposent une limite assez restrictive à la taille de la pile, on peut préférer utiliser la fonction `rev_append`, qui s'écrit sous forme **récursive terminale**, et qui de ce fait occupe un espace $O(1)$ sur la pile. La fonction `rev_append` est définie de façon à ce que `rev_append xs ys` produise le même résultat que `append (rev xs) ys`, où `rev` est la fonction qui renverse une liste. Une fois que l'on a défini `rev_append`, on peut reconstruire `rev` et `append` ainsi :

```
let rev xs =
  rev_append xs []
let append xs ys =
  rev_append (rev xs) ys
```

La définition de `rev_append` est laissée en exercice au lecteur curieux. ◇

Solution de l'exercice 3.2 La fonction `combine` s'écrit aisément sous forme récursive, avec analyse simultanée des deux listes :

```
let rec combine xs ys =
  match xs, ys with
  | [], [] ->
      []
  | x :: xs, y :: ys ->
      (x, y) :: combine xs ys
  | [], _ :: _
  | _ :: _, [] ->
      assert false (* this cannot happen *)
```

Dans les deux dernières branches, on a découvert que les deux listes ne sont pas de même longueur, ce qui contredit la **précondition** de la fonction `combine`. On signale que cela n'est pas censé se produire en écrivant **`assert false`**. Cette expression, si elle est exécutée, lance une

exception `Assert_failure`. On aurait pu choisir de lancer une autre exception, par exemple `Invalid_argument`. Comme rien n'oblige les éléments des listes `xs` et `ys` a être de même type, la fonction `combine` est polymorphe vis-à-vis de deux variables de types, à savoir `'a`, le type des éléments de `xs`, et `'b`, le type des éléments de `ys`. Le type de `combine` est `'a list -> 'b list -> ('a * 'b) list`. ◇

Solution de l'exercice 3.3 Ces deux fonctions s'écrivent naturellement sous forme récursive, avec analyse simultanée de `n` et de `xs` :

```
let rec take n xs =
  match n, xs with
  | 0, _
  | _, [] ->
      (* n is zero or the list is empty *)
      []
  | _, x :: xs ->
      (* n is non-zero and the list is non-empty *)
      x :: take (n - 1) xs

let rec drop n xs =
  match n, xs with
  | 0, _ ->
      xs
  | _, [] ->
      []
  | _, x :: xs ->
      drop (n - 1) xs
```

Notons que la troisième et dernière branche de `take` introduit une nouvelle variable locale `xs`, qui vient cacher la variable locale existante du même nom. Il en va de même dans la troisième branche de `drop`. Ce style peut sembler surprenant, mais est assez courant, et permet d'éviter une prolifération de noms de la forme `xs1`, `xs2`, etc.

Les fonctions `take` et `drop` ont le même type, à savoir `int -> 'a list -> 'a list`. ◇

Solution de l'exercice 3.4 Les types des fonctions `Array.length`, `Array.get` et `Array.set` sont nécessairement les suivants :

```
val length: 'a array -> int
val get: 'a array -> int -> 'a
val set: 'a array -> int -> 'a -> unit
```

On peut vérifier cela en consultant le fichier array.mli de la bibliothèque. En effet, `length` attend un tableau et renvoie sa longueur, donc renvoie un entier. `get` attend un tableau et un indice (donc un entier), et renvoie un élément. `set` attend un tableau, un indice, un élément, et ne renvoie rien, ou plutôt renvoie la valeur `()`, dont le type est `unit`. Ces trois fonctions sont polymorphes : le type `'a` des éléments du tableau n'a pas d'importance.

La fonction `fill` peut s'écrire à l'aide d'une boucle :

```
let fill x a =
  for i = 0 to Array.length a - 1 do
    a.(i) <- x
  done
```

Elle aussi est polymorphe : son type est `'a -> 'a array -> unit`. Si on ne souhaite pas employer une boucle `for`, on peut définir `fill` à l'aide d'une fonction récursive auxiliaire. ◇

Solution de l'exercice 3.5 La solution est donnée dans la Figure 4.17. Notons que la classe `Nil` n'a pas besoin de constructeur, car elle n'a pas de champs. Le classe `Cons` a un constructeur, qui initialise ses deux champs `head` et `tail`. Pour construire une liste de longueur 1 contenant l'élément 42, on écrit `new Cons<Integer> (42, new Nil<Integer> ())`. Malheureusement, bien que les annotations `<Integer>` puissent sembler lourdes, on ne peut pas les supprimer. ◇

```
import java.util.NoSuchElementException;

public class Nil<E> extends List<E> {
  public E head () throws NoSuchElementException {
    throw new NoSuchElementException ();
  }
  public List<E> tail () throws NoSuchElementException {
    throw new NoSuchElementException ();
  }
  public int length () {
    return 0;
  }
  public List<E> append (List<E> that) {
    return that;
  }
}

public class Cons<E> extends List<E> {
  private final E head;
  private final List<E> tail;
  public Cons (E head, List<E> tail) {
    this.head = head;
    this.tail = tail;
  }
  public E head () {
    return head;
  }
  public List<E> tail () {
    return tail;
  }
  public int length () {
    return 1 + tail.length();
  }
  public List<E> append (List<E> that) {
    return new Cons<E> (head, tail.append(that));
  }
}
```

FIGURE 4.17 – Les classes `Nil` et `Cons` (Exercice 3.5)

Solution de l'exercice 3.6 La solution est donnée par la Figure 4.18. La méthode `convert` est polymorphe vis-à-vis du type `E` des éléments du tableau. Le code est simple : on crée d'abord une nouvelle table de hachage vide, puis on y ajoute (une par une) toutes les associations de `i` à `array[i]`. On renvoie enfin la table ainsi peuplée. ◇

Solution de l'exercice 3.7 La solution est donnée par la Figure 4.19. Le type de la méthode `invert` reflète clairement son comportement : étant donnée une table de type `HashMap<K, V>`, elle produit une table de type `HashMap<V, K>`. Elle est polymorphe vis-à-vis du type des clefs, `K`, et vis-à-vis du type des valeurs, `V`.

En réalité, tout ceci n'a de sens que si les types `K` et `V` sont munis de méthodes `equals` et `hashCode`, de façon à ce que « clefs » et « valeurs » puissent être utilisées en tant que clefs dans une table de hachage. Cette hypothèse à propos des types `K` et `V` n'apparaît pas explicitement car Java considère que `K` et `V` sont sous-types de `Object`, donc sont nécessairement dotés de ces méthodes.

Comme dans l'Exercice 3.6, on crée une nouvelle table, puis on la peuple, puis on la renvoie. La seule difficulté potentielle est de comprendre comment on énumère toutes les paires clef-valeur contenues dans la table `map`. Une étude de la documentation de la classe `HashMap` montre que la méthode `entrySet` produit un objet de type `Set<Entry<K, V>>`, c'est-à-dire un ensemble de paires clef-valeur. Parce que l'interface `Set` étend l'interface `Iterable`, il est possible d'énumérer les éléments d'un tel ensemble tout simplement à l'aide d'une boucle `for` (§4.2). À chaque itération de cette boucle, on nous donne accès à un objet `entry` de type `Entry<K, V>`. Lorsqu'on consulte la documentation de la classe `Map.Entry`, on constate que cet objet est tout simplement une paire d'une clef et d'une valeur, et que l'on peut obtenir ces deux composantes à l'aide des méthodes `getKey` et `getValue`. Une fois que l'on a obtenu ainsi la clef et la valeur, il ne reste plus qu'à introduire dans la table inverse `inverseMap` une association en sens inverse, de la valeur vers la clef. ◇

Solution de l'exercice 3.8 La fonction `find_zero` s'écrit naturellement de façon récursive :

```
let rec find_zero xs =
  match xs with
  | [] ->
      false
  | x :: xs ->
      x = 0 || find_zero xs
```

On a utilisé ici le « ou » logique, noté `||`, mais on aurait pu employer une construction `if-then-else` explicite. C'est d'ailleurs ce que l'on doit faire dans `find_odd`, car cette fonction renvoie non pas un booléen, mais une option :

```
let rec find_odd xs =
  match xs with
  | [] ->
      None
  | x :: xs ->
      if x mod 2 <> 0 then
        Some x
      else
        find_odd xs
```

On voit que les fonctions `find_zero` et `find_odd` sont identiques, sauf en deux points : (a) la propriété que l'on attend de `x` n'est pas la même ; (b) le résultat de `find_zero` est de type `bool`, tandis celui de `find_odd` est de type `int option`.

Pour traiter le point (a), on écrit une fonction `find` paramétrée par la propriété qui nous intéresse, ou plutôt, paramétrée par une fonction `p` qui, appliquée à `x`, indique si la propriété qui nous intéresse est satisfaite ou non par `x`.

```
import java.util.HashMap;

public class ArrayToHashMap {

  static <E> HashMap<Integer, E> convert (E[] array) {
    HashMap<Integer, E> map = new HashMap<Integer, E> ();
    for (int i = 0; i < array.length; i++)
      map.put(i, array[i]);
    return map;
  }

}
```

FIGURE 4.18 – Conversion d'un tableau en une table de hachage

```
import java.util.HashMap;
import java.util.Map.Entry;

public class InvertHashMap {

  static <K, V> HashMap<V, K> invert (HashMap<K, V> map) {
    HashMap<V, K> inverseMap = new HashMap<V, K> ();
    for (Entry<K, V> entry : map.entrySet())
      inverseMap.put(entry.getValue(), entry.getKey());
    return inverseMap;
  }

}
```

FIGURE 4.19 – Inversion d'une table de hachage

Pour traiter le point (b), on peut noter que le type `int option` est plus riche que le type `bool` : on peut convertir du premier vers le second. On décide donc que la fonction `find` renverra une option. (Nous présentons plus bas une autre façon de procéder.)

La fonction `find` est donc définie ainsi :

```
let rec find p xs =
  match xs with
  | [] ->
      None
  | x :: xs ->
      if p x then
        Some x
      else
        find p xs
```

La fonction `find` attend deux arguments `p` et `xs`, et `p` est elle-même une fonction qui, étant donné un élément, renvoie un booléen. Les conditions `x = 0` et `x mod 2 <> 0` ont été remplacées par la condition `p x`.

Comme les fonctions `find_zero` et `find_odd` ne sont applicables qu'à des listes d'entiers, on pourrait s'attendre à ce que `find` elle aussi ne soit applicable qu'à des listes d'entiers, donc à ce que `find` aie le type `(int -> bool) -> int list -> int option`. C'est vrai : la fonction `find` admet ce type. Cependant, elle admet un type plus général, qui est inféré par OCaml, à savoir `('a -> bool) -> 'a list -> 'a option`. En effet, plus rien dans le code de `find` n'est spécifique des entiers. Les seuls aspects spécifiques des entiers se trouvaient dans les conditions `x = 0` et `x mod 2 <> 0` ; or celles-ci ont été remplacées par la condition `p x`.

On peut maintenant redéfinir `find_zero` et `find_odd` à l'aide de `find` :

```
let find_zero xs =
  match find (fun x -> x = 0) xs with
  | None ->
      false
  | Some _ ->
      true

let find_odd xs =
  find (fun x -> x mod 2 <> 0) xs
```

Dans le code de `find_zero`, l'appel `find (fun x -> x = 0) xs` produit une option, que l'on convertit a posteriori en un booléen.

Pour traiter le point (b) plus haut, au lieu de décider que `find` renvoie une option, on aurait pu ajouter à la fonction `find` des paramètres supplémentaires, de façon à abstraire toutes les différences entre `find_zero` et `find_odd`. Une fois les conditions `x = 0` et `x mod 2 <> 0` remplacées par la condition `p x`, il reste deux différences : d'un côté, on renvoie `false` ou `true`, tandis que de l'autre, on renvoie `None` ou `Some x`. Pour abstraire ces différences, on ajoute à `find` deux paramètres `none` et `some`, le premier indiquant ce qu'il faut renvoyer en cas d'échec, le second indiquant ce qu'il faut renvoyer en cas de succès.

```
let rec find none some p xs =
  match xs with
  | [] ->
      none
  | x :: xs ->
      if p x then
        some x
      else
        find none some p xs
```

Le type de cette fonction `find`, inféré par le compilateur OCaml, est plus complexe. On peut l'écrire

```
'r -> ('a -> 'r) -> ('a -> bool) -> 'a list -> 'r
```

Ici `'a` est le type des éléments de la liste, comme précédemment, et `'r` est le type du résultat de `find`, par exemple `bool` ou `int option`. Le paramètre `none` a le type `'r`, car c'est un résultat, et le paramètre `some` a le type `'a -> 'r`, car c'est une fonction qui, appliquée à un élément `x`, produit un résultat.

On peut à nouveau redéfinir `find_zero` et `find_odd` à l'aide de `find` :

```
let find_zero xs =
  find false (fun x -> true) (fun x -> x = 0) xs

let find_odd xs =
  find None (fun x -> Some x) (fun x -> x mod 2 <> 0) xs
```

On constate que `find_zero` est maintenant obtenue par un simple appel à `find`. On n'a plus besoin de convertir a posteriori une option en un booléen, car on s'est donné une forme suffisamment générale de la fonction `find`. ◇

Solution de l'exercice 3.9 Notons tout d'abord qu'il est permis en Java de définir deux méthodes de même nom dans une même classe (ici, deux méthodes nommées `mergeSort`), à condition que leurs listes de paramètres ne soient pas les mêmes, ce qui permet de les différencier. On appelle cela la **surcharge**, ou « *overloading* ».

La solution est donnée dans la Figure 4.20. La méthode principale, nommée `mergeSort`, effectue simplement un appel à la méthode récursive, également nommée `mergeSort`. Elle donne les valeurs `0` et `n` aux paramètres `lo` et `hi`, de façon à trier tout le tableau. Le paramètre `T`, qui représente l'espace de travail auxiliaire, est instancié par un tableau fraîchement alloué, de taille `n`.

La méthode récursive `mergeSort` traite d'abord deux cas de base, puis le cas général. Les cas de base sont ceux où le segment `[lo, hi[` est de longueur 0 ou 1. Dans le premier cas, il n'y a rien à faire, et dans le second cas, il suffit de copier un élément du tableau `A` vers le tableau `B`. Dans le cas général, le segment `[lo, hi[` est de longueur 2 au moins. On peut donc le diviser en deux sous-segments de longueur égale (à 1 près), à savoir `[lo, limit[` et `[limit, hi[`, avec la certitude que ces deux sous-segments sont de longueur strictement inférieure à celle du segment `[lo, hi[`, ce qui garantit la terminaison de l'algorithme. On effectue deux appels récursifs pour trier ces deux sous-segments, en prenant soin d'écrire les résultats dans le tableau auxiliaire `T`. (Pendant ces deux appels récursifs, le segment `[lo, hi[` du tableau `B` ne contient pas de données utiles, et peut donc lui-même servir d'espace de travail auxiliaire.) Enfin, il reste à fusionner les segments `[lo, limit[` et `[limit, hi[` du tableau `T`, qui sont à présent triés, et à écrire le résultat dans `B`. On utilise pour cela la méthode auxiliaire `merge`. Cette dernière méthode fait progresser deux indices `src1` et `src2` dans les intervalles `[lo, limit[` et `[limit, hi[` respectivement. À chaque étape de la première boucle, on compare les éléments `T[src1]` et `T[src2]`. Cette comparaison se fait à l'aide de la méthode `compare` offerte par l'objet `c`. l'élément `T[src1]` ou bien l'élément `T[src2]` est copié vers le tableau `B`. Les deux boucles qui suivent traitent les situations où l'un des indices `src1` et `src2` a atteint sa limite.

Ce code est un peu complexe, du point de vue algorithmique comme du point de vue de son écriture en Java idiomatique, et mérite d'être soigneusement écrit et testé. Cependant, du point de vue qui nous intéresse principalement ici, à savoir paramétrer l'algorithme de tri vis-à-vis d'une relation d'ordre, la chose est très simple : il suffit de transporter le paramètre `c` jusqu'au point où on doit effectuer une comparaison, où on utilise alors la méthode `c.compare`.

Il reste à décrire comment faire appel à l'algorithme lorsque l'on souhaite trier un tableau par ordre décroissant. Supposons que l'on dispose de deux tableaux `A` et `B`, source et destination. Évidemment, il faut appeler `mergeSort(c, A, B)`, où `c` est un objet de type `Comparator<Integer>`. Il nous revient de construire l'objet `c` de façon à ce qu'il représente l'ordre décroissant sur les entiers. On peut utiliser pour cela une **classe anonyme** :

```
import java.util.Comparator;

public class MergeSortInteger {

  // Effect: B = sort(A)
  static void mergeSort (Comparator<Integer> c, int[] B, int[] A)
  {
    int n = A.length;
    int[] T = new int [n];
    mergeSort(c, B, A, 0, n, T);
  }

  // Effect: B[lo..hi) = sort(A[lo..hi))
  // T[lo..hi) can be used as temporary storage
  static void mergeSort (Comparator<Integer> c, int[] B, int[] A,
                                          int lo, int hi, int[] T)
  {
    if (hi - lo == 0)
      return;
    if (hi - lo == 1)
      B[lo] = A[lo];
    else {
      int limit = lo + (hi - lo) / 2;
      mergeSort(c, T, A, lo, limit, B);
      mergeSort(c, T, A, limit, hi, B);
      merge(c, B, T, lo, limit, hi);
    }
  }

  // Requires: T[lo..limit) and T[limit..hi) are sorted
  // Effect  : B[lo..hi) = sort(T[lo..hi))
  static void merge (Comparator<Integer> c, int[] B, int[] T,
                                    int lo, int limit, int hi)
  {
    int dst = lo;
    int src1 = lo;
    int src2 = limit;
    while (src1 < limit && src2 < hi)
      B[dst++] =
        c.compare(T[src1], T[src2]) < 0 ? T[src1++] : T[src2++];
    while (src1 < limit)
      B[dst++] = T[src1++];
    while (src2 < hi)
      B[dst++] = T[src2++];
  }

}
```

FIGURE 4.20 – L'algorithme MergeSort pour des tableaux d'entiers

```
Comparator<Integer> c =
  new Comparator<Integer> () {
    public int compare (Integer n1, Integer n2) {
      return n1 < n2 ? 1 : n1 == n2 ? 0 : -1;
    }
  };
```

Depuis Java 8, on peut également écrire cela de façon plus concise sous forme d'une **clôture**. La différence est purement syntaxique ; ces deux versions du code produisent, après compilation, les mêmes instructions pour la machine.

```
Comparator<Integer> c =
  (n1, n2) -> {
    return n1 < n2 ? 1 : n1 == n2 ? 0 : -1;
  };
```

À ceux qui auraient du mal à lire l'expression `n1 < n2 ? -1 : n1 == n2 ? 0 : 1`, rappelons que, en Java comme en C, l'expression `b ? e1 : e2` est une expression conditionnelle : « si `b` alors `e1` sinon `e2` ». On aurait pu employer `if-then-else`, qui n'est pas une expression mais une instruction. Le corps de la méthode `compare` aurait alors été `if (n1 < n2) return 1; else if (n1 == n2) return 0; else return -1;`. ◇

Solution de l'exercice 3.10 La fonction `maximum` s'écrit naturellement sous forme récursive :

```
let rec maximum xs =
  match xs with
  | [] ->
      min_int
  | x :: xs ->
      max x (maximum xs)
```

Cette version du code n'est pas récursive terminale. On peut aussi en donner une version récursive terminale, qui utilisera un espace $O(1)$, tandis que la version ci-dessus exige un espace $O(n)$. Cela n'est pas le sujet de cet exercice ; nous ne le faisons pas ici.

Si la liste est vide, on renvoie `min_int`, le plus petit des entiers représentables en machine, qui est l'élément neutre pour l'opération « maximum ». Si la liste s'écrit `x :: xs`, alors on calcule le maximum des deux entiers `x` et `maximum xs`. On utilise pour cela la fonction `max`. Celle-ci fait partie de la bibliothèque d'OCaml, mais nous pourrions aussi bien la définir nous-mêmes :

```
let max (x : int) (y : int) : int =
  if x < y then y else x
```

Le type de `max` est ici `int -> int -> int`. Le type de `maximum` est `int list -> int`.

À présent, voyons comment abstraire notre fonction `maximum` pour la rendre aussi générale que possible, c'est-à-dire aussi réutilisable que possible. On souhaiterait pouvoir appliquer cette fonction non pas seulement à des listes d'entiers, mais à des listes d'éléments de type arbitraire `'a`. Seuls deux points font que la fonction `maximum` actuelle n'est applicable qu'aux entiers : d'une part, `min_int` est un entier ; d'autre part, `max` est une fonction des entiers dans les entiers. Par conséquent, si nous paramétrons la fonction `maximum` par une fonction `join` de type `'a -> 'a -> 'a` et par son élément neutre `bottom` de type `'a`, alors `maximum` sera applicable à une liste de type `'a list`, et produira un résultat de type `'a`. On écrit donc :

```
let rec maximum join bottom xs =
  match xs with
  | [] ->
      bottom
  | x :: xs ->
      join x (maximum join bottom xs)
```

Si le type `'a`, muni de la constante `bottom` et de l'opération `join`, est un sup-demi-treillis, alors on voit que `maximum` calcule la borne supérieure (au sens du sup-demi-treillis) de la liste d'éléments `xs`. Pour spécialiser à nouveau la fonction `maximum` aux entiers, il suffit de l'appliquer aux arguments `max` et `min_int`.

On s'attend à ce que la fonction `maximum` admette le type suivant :

```
val maximum: ('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

En effet, `maximum` admet bien ce type. Toutefois, si on demande au compilateur OCaml d'inférer le type de `maximum`, il affiche un type plus général encore que le précédent :

```
val maximum: ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

En effet, quelques instants de réflexion montrent que rien dans le code de la fonction `maximum` n'oblige les deux arguments de `join` à être de même type. Si on fait l'hypothèse que `join` attend deux arguments de type `'a` et `'b` respectivement, alors pour que le code ait un sens (i.e., soit bien typé) il suffit que `bottom` soit de type `'b` et que `join` produise un résultat de type `'b`. La fonction `maximum` produit alors elle-même un résultat de type `'b`.

En réalité, la fonction `maximum` proposée plus haut est essentiellement `List.fold_right` de la bibliothèque d'OCaml. (Il suffirait d'intervertir les paramètres `bottom` et `xs` dans la définition ci-dessus pour obtenir exactement `List.fold_right`.) En effet, le code de `maximum` impose d'itérer sur la liste, mais ne fixe pas ce qu'il faut faire dans le cas de base (le paramètre `bottom` abstrait cela) ni ce qu'il faut faire pour combiner l'élément `x` et le résultat de l'appel récursif sur `xs` (le paramètre `join` abstrait cela). Nous étudierons ces fonction d'itération très générales, que nous appelons des **réducteurs**, dans le dernier chapitre (§4.1.2).

Pour clore cet exercice, notons qu'on aurait pu proposer des solutions différentes, car il n'existe pas une manière unique de « généraliser » un fragment de code. Au début, nous aurions pu écrire la fonction `maximum` spécialisée aux entiers sous la forme suivante, où `max` a été « inlinée » dans `maximum` :

```
let rec maximum xs =
  match xs with
  | [] ->
      min_int
  | x :: xs ->
      let y = maximum xs in
      if x < y then y else x
```

À partir de là, pour rendre cette fonction polymorphe, il nous aurait semblé naturel de l'abstraire vis-à-vis de la relation d'ordre `(<)` et vis-à-vis de son élément mininum. Nous aurions donc pu écrire :

```
let rec maximum less bottom xs =
  match xs with
  | [] ->
      bottom
  | x :: xs ->
      let y = maximum less bottom xs in
      if less x y then y else x
```

La fonction ainsi obtenue admet le type suivant :

```
val maximum: ('a -> 'a -> bool) -> 'a -> 'a list -> 'a
```

La version précédente de `maximum` exigeait comme premier argument une opération `join`, tandis que cette version exige seulement une relation d'ordre `less`. Cette version peut donc sembler plus facile d'emploi, mais est moins souvent utilisable. Ceci illustre (à petite échelle) la tension qui existe entre définir un fragment de code très général, mais qui laisse beaucoup de travail à son utilisateur, et définir un fragment de code moins général, mais plus immédiatement prêt à l'emploi. ◇

Solution de l'exercice 3.11 La fonction `filter` spécialisée s'écrit ainsi :

```
let rec filter xs =
  match xs with
  | [] ->
      []
  | x :: xs ->
      if x mod 2 = 0 then x :: filter xs else filter xs
```

Son type est `int list -> int list`. Pour la rendre applicable à une liste de type `'a list`, où le type `'a` des éléments est arbitraire, il faut abstraire le code vis-à-vis de la propriété « être pair », qui est spécifique au type `int`. On suppose donc donnée une propriété `p`, c'est-à-dire une fonction `p` de type `'a -> bool`, et on paramètre `filter` vis-à-vis de `p`, comme ceci :

```
let rec filter p xs =
  match xs with
  | [] ->
      []
  | x :: xs ->
      if p x then x :: filter p xs else filter p xs
```

La fonction `filter` ainsi obtenue, appliquée à `p` et à `xs`, renvoie la sous-liste des éléments de `xs` qui satisfont la propriété `p`. Par exemple, `filter (fun x -> x > 0) xs` renvoie la sous-liste des éléments strictement positifs de `xs`. Le type de `filter` est `('a -> bool) -> 'a list -> 'a list`.

Pour ce qui concerne la fonction `partition`, le processus est le même. Nous donnons directement la version polymorphe de cette fonction, elle aussi paramétrée par un prédicat `p` :

```
let rec partition p xs =
  match xs with
  | [] ->
      [], []
  | x :: xs ->
      let ys, zs = partition p xs in
      if p x then x :: ys, zs else ys, x :: zs
```

(Par convention, le constructeur `::` des listes a priorité sur le constructeur `,` des paires.) Le type de `partition` est `('a -> bool) -> 'a list -> 'a list * 'a list`. En effet, son résultat est une paire de listes. ◇

Solution de l'exercice 3.12 La fonction `sort` doit avoir pour paramètre un tableau `a` dont les éléments ont un type quelconque `'a`. Le type de `a` est donc `'a array`. Par ailleurs, la fonction `sort` doit également avoir pour paramètre la relation d'ordre souhaitée, ou plus précisément, une fonction `compare` à qui on soumet deux éléments et qui permet de les comparer. Ici, plusieurs possibilités se présentent. L'idée la plus simple est que `compare` décide la relation d'ordre strict : `compare x y` renvoie alors **`true`** si `x` est strictement plus petit que `y`, au sens de la relation d'ordre considérée, et **`false`** dans le cas contraire. Le type de `compare` doit alors être `'a -> 'a -> bool`. Cependant, dans ce cas, deux appels à `compare` sont nécessaires pour déterminer si `x` est inférieur, égal, ou supérieur à `y`. On préfère donc traditionnellement exiger que `compare` produise directement cette information. Pour cela, on pourrait définir un type somme à trois branches :

```
type outcome = Lt | Eq | Gt
```

Le type de de `compare` devrait alors être `'a -> 'a -> outcome`. Pour des raisons historiques, dans la bibliothèque d'OCaml, comme dans les interfaces `Comparator` et `Comparable` de Java, on demande plutôt que `compare` produise un résultat entier, dont le signe (négatif, nul, positif) détermine la signification. En OCaml, le type de `compare` est donc `'a -> 'a -> int`. La fonction `sort`, qui est paramétrée par `compare` et par `a`, admet donc le type :

```
val sort: ('a -> 'a -> int) -> 'a array -> unit
```

Le type de son résultat est `unit`, car elle ne renvoie « rien », ou plutôt elle renvoie la valeur `()`. Le tableau `a` est modifié en place.

L'implémentation de la fonction `sort` est donnée dans la Figure 4.21. Comme le veut la structure de l'algorithme QuickSort, la fonction `sort` est définie en termes d'une fonction récursive `sort_segment`, paramétrée par deux indices `lo` et `hi` qui délimitent le segment `[lo, hi[` à trier. Si ce segment est de longueur 0 ou 1, il n'y a rien à faire. Sinon, on choisit aléatoirement parmi les éléments de ce segment une valeur pivot, puis on appelle `partition` pour découper ce segment en trois sous-segments `[lo, lt[`, `[lt, gt[`, et `[gt, hi[`, qui contiennent respectivement les valeurs inférieures au pivot, égales au pivot, et supérieures au pivot. Il reste à effectuer deux appels récursifs pour trier le premier et le dernier de ces sous-segments. Le second sous-segment est nécessairement non vide, puisqu'il contient la valeur pivot : ceci garantit la terminaison de l'algorithme.

Il reste à implémenter la fonction `partition`. On pourrait l'écrire sous forme d'une boucle ; c'est ce que l'on ferait en Java. En OCaml, `partition` peut aussi s'écrire très naturellement sous forme récursive. Les appels récursifs étant terminaux (Remarque 1.20), le code machine produit par le compilateur OCaml sera identique à celui d'une boucle.

Le cas le plus simple est celui où `eq` et `gt` sont égaux. Dans ce cas, la zone contenant les éléments encore indéterminés est vide. Il n'y a donc plus rien à faire : il suffit de renvoyer la paire `(lt, gt)`. Dans le cas contraire, cette zone est non vide. On s'intéresse alors à son premier élément, à savoir `a.(eq)`, que l'on compare au pivot. Suivant le résultat `c` de cette comparaison, on déplace si besoin cet élément vers une position appropriée, et on fait évoluer les indices `lt`, `eq` et `gt` de façon appropriée. (Dans chaque cas, un dessin peut aider à comprendre la situation.) On ne modifie pas les variables `lt`, `eq` et `gt`, qui d'ailleurs ne sont pas modifiables. Au lieu de cela, il suffit d'appeler récursivement `partition`, en lui passant les nouvelles valeurs de `lt`, `eq` et `gt`, afin qu'elle termine le travail.

On pourrait s'attendre à ce que le type de `partition` soit :

```
('a -> 'a -> int) -> 'a -> 'a array ->
int -> int -> int -> int * int
```

Ce type est correct, en effet. La fonction `partition` est polymorphe vis-à-vis du type `'a` des éléments du tableau. Toutefois, si on demande au compilateur OCaml d'inférer le type de `partition`, il affiche un type plus général encore que le précédent :

```
('a -> 'b -> int) -> 'b -> 'a array ->
int -> int -> int -> int * int
```

Le code de `partition` contient un seul appel à la fonction `compare`. Le premier argument fourni à `compare` est toujours issu du tableau, tandis que le second est toujours `pivot`. Il n'y a a priori aucune contrainte qui impose que la valeur `pivot` soit du même type que les éléments du tableau : en général, `pivot` peut avoir un type `'b` quelconque, à condition que la fonction `compare` soit de type `'a -> 'b -> int`. L'**inférence de types** permet parfois de s'apercevoir qu'on a écrit du code plus général que ce que l'on pensait ! ◇

Solution de l'exercice 3.13 La solution est donnée dans la Figure 4.22. Les différences vis-à-vis de la version précédente, qui était spécialisée pour les tableaux d'entiers (Figure 4.20), sont minimes. On a remplacé toutes les occurrences des types `int` et `Integer` par une variable de types `E`. De plus, on a explicitement écrit `<E>` dans la signature de chacune des méthodes : cette quantification universelle signifie que ces méthodes fonctionnent pour toute valeur de `E`. Enfin, comme le compilateur Java n'accepte pas l'expression `new E [n]` (voir le Détail 3.3), on a remplacé cette expression par l'expression `java.util.Arrays.copyOf(A, n)`, qui elle aussi alloue un nouveau tableau de `n` éléments et de type `E[]`.

Notre algorithme de tri est maintenant en principe applicable à un tableau de type `E[]`, pour n'importe quelle valeur de `E`. Nous pouvons donc l'appliquer à des tableaux de type

```
let swap a i j =
  let v = a.(i) in
  a.(i) <- a.(j);
  a.(j) <- v

let rec partition compare pivot a lt eq gt =
  assert (lt <= eq && eq <= gt);
  if eq = gt then
    lt, gt
  else begin
    let v = a.(eq) in
    let c = compare v pivot in
    if c < 0 then begin
      swap a lt eq;
      partition compare pivot a (lt + 1) (eq + 1) gt
    end
    else if c > 0 then begin
      swap a eq (gt - 1);
      partition compare pivot a lt eq (gt - 1)
    end
    else
      partition compare pivot a lt (eq + 1) gt
  end

let rec sort_segment compare a lo hi =
  assert (0 <= lo && lo <= hi && hi <= Array.length a);
  let length = hi - lo in
  if length > 1 then begin
    let index = lo + Random.int length in
    let pivot = a.(index) in
    let lt, gt = partition compare pivot a lo lo hi in
    assert (lt < gt);
    sort_segment compare a lo lt;
    sort_segment compare a gt hi
  end

let sort compare a =
  sort_segment compare a 0 (Array.length a)
```

FIGURE 4.21 – L'algorithme QuickSort, sous forme polymorphe, avec partition en trois

```
import java.util.Comparator;

public class MergeSort {

  // Effect: B = sort(A)
  static <E> void mergeSort (Comparator<E> c, E[] B, E[] A)
  {
    int n = A.length;
    E[] T = java.util.Arrays.copyOf(A, n);
    mergeSort(c, B, A, 0, n, T);
  }

  // Effect: B[lo..hi) = sort(A[lo..hi))
  // T[lo..hi) can be used as temporary storage
  static <E> void mergeSort (Comparator<E> c, E[] B, E[] A,
                                     int lo, int hi, E[] T)
  {
    if (hi - lo == 0)
      return;
    if (hi - lo == 1)
      B[lo] = A[lo];
    else {
      int limit = lo + (hi - lo) / 2;
      mergeSort(c, T, A, lo, limit, B);
      mergeSort(c, T, A, limit, hi, B);
      merge(c, B, T, lo, limit, hi);
    }
  }

  // Requires: T[lo..limit) and T[limit..hi) are sorted
  // Effect  : B[lo..hi) = sort(T[lo..hi))
  static <E> void merge (Comparator<E> c, E[] B, E[] T,
                              int lo, int limit, int hi)
  {
    int dst = lo;
    int src1 = lo;
    int src2 = limit;
    while (src1 < limit && src2 < hi)
      B[dst++] =
        c.compare(T[src1], T[src2]) < 0 ? T[src1++] : T[src2++];
    while (src1 < limit)
      B[dst++] = T[src1++];
    while (src2 < hi)
      B[dst++] = T[src2++];
  }

}
```

FIGURE 4.22 – L'algorithme MergeSort sous forme polymorphe

```
let compare (s1 : string) (s2 : string) : int =
  let n1 = String.length s1
  and n2 = String.length s2 in
  let rec loop i : int =
    (* We assume that the comparison has succeeded up to index i. *)
    match i = n1, i = n2 with
    | true, true ->
        (* The strings have the same length. We are at the end. *)
        0
    | false, true ->
        (* We are at the end of [s2], and [s1] is longer. *)
        1
    | true, false ->
        (* We are at the end of [s1], and [s2] is longer. *)
        -1
    | false, false ->
        (* We have one character on each side. *)
        let c1 = s1.[i]
        and c2 = s2.[i] in
        let cmp = Char.compare c1 c2 in
        if cmp <> 0 then cmp
        else loop (i + 1)
  in
  loop 0
```

FIGURE 4.23 – Comparaison lexicographique de deux chaînes de caractères

`Boolean[]`, `Integer[]`, `int[][]`, etc. Malheureusement, nous ne pouvons pas l'appliquer à un tableau de type `boolean[]`, `int[]`, etc., parce que Java n'autorise pas une variable de types `E` à être instanciée par un type primitif (Détail 3.1). ◇

Solution de l'exercice 3.14 Il faut utiliser soit une boucle `while` soit une fonction récursive pour comparer les caractères `s1[i]` et `s2.[i]`, où `i` prend successivement les valeurs 0, 1, etc. La Figure 4.23 propose une solution basée sur une fonction récursive locale, `loop`. Cette fonction attend l'indice `i` comme argument. Elle suppose (elle exige) que `i` est inférieur ou égal à `n1` et à `n2`, et que la comparaison a réussi pour les indices inférieurs à `i`. Les comparaisons `i = n1` et `i = n2` donnent lieu à quatre cas de figure. Dans le dernier cas, on a `i < n1` et `i < n2`, donc les caractères `s1.[i]` et `s2.[i]` sont bien définis. On les compare et, s'ils sont égaux, on effectue l'appel récursif `loop (i + 1)` pour continuer la comparaison. Cette fonction **récursive terminale** s'exécute en temps $O(\min(n_1, n_2))$ et en espace $O(1)$. ◇

Solution de l'exercice 3.15 Le code a déjà été donné plus haut. Il suffit d'en faire une fonction au lieu d'un foncteur :

```
let compare compare1 compare2 (x1, x2) (y1, y2) =
  let c1 = compare1 x1 y1 in
  if c1 <> 0 then c1
  else compare2 x2 y2
```

On voit ici que l'on n'est pas obligé d'écrire un foncteur, qui attend deux structures et produit une structure, mais que l'on peut préférer écrire une fonction, qui attend deux fonctions de comparaison et produit une fonction de comparaison. C'est, dans une certaine mesure, une question de goût. L'utilisation des structures et des foncteurs devient essentielle lorsqu'on manipule des structures dotées de nombreuses composantes. ◇

Solution de l'exercice 3.16 Il faut écrire un foncteur, disons `LexicoList`, dont l'argument doit satisfaire la signature `OrderedType` et dont le résultat satisfait cette même signature. La Figure 4.24 propose une solution.

```
(* .ml file: *)
module LexicoList (E : OrderedType) = struct
  type t = E.t list
  let rec compare xs ys =
    match xs, ys with
    | [], [] ->
        0
    |  _, [] ->
        1
    | [],  _ ->
        -1
    | x :: xs, y :: ys ->
        let cmp = E.compare x y in
        if cmp <> 0 then cmp
        else compare xs ys
end

(* .mli file: *)
module LexicoList (E : OrderedType) :
  OrderedType with type t = E.t list
```

FIGURE 4.24 – Comparaison lexicographique de deux listes

Le code de la fonction de comparaison rappelle l'Exercice 3.14. Il est plus simple, en réalité, parce que l'analyse et la décomposition d'une liste se font facilement à l'aide d'un `match`.

Dans le fichier `.mli`, on donne le type du foncteur `LexicoList`. On indique donc que son argument `E` est une structure de signature `OrderedType` et que son résultat est une structure de signature `OrderedType with type t = E.t list`.

On pourrait déclarer seulement que le résultat du foncteur a pour signature `OrderedType`, sans donner l'équation `t = E.t list`. Cependant, cela rendrait la nouvelle fonction `compare` inutilisable. Il faut publier cette équation pour que l'on sache que la fonction `compare` s'applique à des listes. ◇

Solution de l'exercice 3.17 La signature `Queue` doit au minimum déclarer un type `queue` des files d'attente ainsi que les trois opérations `create`, `insert`, `extract` mentionnées dans l'énoncé. Deux questions au moins se posent alors.

D'une part, comment écrire le type des éléments de la file ? Il pourrait être lui aussi déclaré dans la signature sous forme d'un type fixé `element` ; ou bien, si les opérations sur les files sont polymorphes, il pourrait être représenté par une variable de types `'a`.

D'autre part, les files d'attente sont-elles modifiables ou non ? Si oui, alors (par exemple) l'insertion d'un nouvel élément modifie la file en place et ne renvoie aucun résultat ; dans le cas contraire, l'insertion renvoie une nouvelle file.

Si on choisit de spécifier par exemple des files monomorphes et modifiables, alors on définit la signature `Queue` ainsi :

```
module type Queue = sig
  type queue
  type element
  val create: unit -> queue
  val insert: element -> queue -> unit
  val extract: queue -> element option
end
```

Une structure de file d'attente dont les éléments sont des entiers aurait alors pour signature `Queue with type element = int`.

Si on choisit au contraire des files polymorphes et non modifiables, alors on définit la signature `Queue` ainsi :

```
module type Queue = sig
  type 'a queue
  val empty: 'a queue
  val insert: 'a -> 'a queue -> 'a queue
  val extract: 'a queue -> ('a * 'a queue) option
end
```

On voit que le type `element` a disparu, remplacé par `'a`. De plus, `create` n'est plus une fonction mais une constante `empty` : c'est **la** file vide. La fonction `insert` renvoie maintenant une nouvelle file, tandis que `extract` renvoie (en cas de succès) une paire d'un élément et d'une nouvelle file.

Les deux autres combinaisons – files monomorphes non modifiables et files polymorphes modifiables – ont également un sens.

Une troisième question pouvait se poser : en cas d'échec, la fonction `extract` doit-elle renvoyer une valeur particulière ou bien lancer une exception ? On a choisi ci-dessus la première solution (et utilisé à cet effet le type `option`), mais la seconde est viable également. ◇

Solution de l'exercice 3.18 Si l'on consulte le fichier map.ml, on constate que le type `'a t` des tables d'association est défini comme un type algébrique qui représente des arbres binaires de recherche équilibrés :

```
module Make (Ord : OrderedType) = struct

  (* Our trees are binary search trees with respect to the ordering
     [Ord.compare]. Furthermore, they are height-balanced. *)
  type 'a t =
      Empty
    | Node of 'a t * key * 'a * 'a t * int

  ...

end
```

Donc, la représentation en mémoire d'une table d'association est un arbre binaire. C'est vrai aussi bien pour les tables de type `'a M1.t` que pour les tables de type `'a M2.t`.

On pourrait donc croire, a première vue, qu'il n'y a pas de danger à considérer ces deux types comme égaux : ces deux types semblent décrire les mêmes arbres binaires.

Toutefois, une table de type `'a t` n'est pas un arbre binaire quelconque, mais un arbre binaire **de recherche**, ce qui signifie que les clefs sont rangées d'une manière qui respecte la relation d'ordre `Ord.compare`. C'est ce que souligne le commentaire qui précède la définition du type `'a t`. En bref, parce que le type des tables est abstrait, l'auteur du module `Map` a pu imposer **l'invariant** comme quoi l'ordre des clefs respecte `Ord.compare`.

Or, cet invariant dépend de la relation d'ordre choisie par l'utilisateur. Lorsqu'il construit le module `M1`, l'utilisateur choisit de ranger les clefs par ordre croissant ; lorsqu'il construit le module `M2`, il fait le choix opposé. Les types `'a M1.t` et `'a M2.t` sont donc incompatibles à juste titre, puisque l'organisation des clefs n'y est pas la même. Si on considérait que les types `'a M1.t` et `'a M2.t` sont égaux, alors on pourrait (par exemple) utiliser la fonction `M1.find` pour rechercher une clef dans un arbre construit à l'aide des fonctions du module `M2`. En d'autres termes, on utiliserait l'ordre croissant pour effectuer une recherche dans un arbre où les clefs sont rangées par ordre décroissant, ce qui n'aurait aucun sens.

En bref, pour obtenir les bénéfices habituellement liés à l'abstraction, il faut considérer que chaque application de `Map.Make` produit une abstraction distincte. ◇

Solution de l'exercice 3.19 Voyons d'abord pourquoi l'énoncé ne demande pas de construire un objet de type `IRichStack<E>` à partir d'un objet de type `IStack<E>`. Cela signifierait que,

```
public static <E>
Supplier<IRichStack<E>> enrich (Supplier<IStack<E>> supplier) {
  return () ->
    new IRichStack<E> () {
      private IStack<E> stack = supplier.get();
      private int size = 0;
      public void push (E x)
      {
        stack.push(x);
        size++;
      }
      public E pop () throws NoSuchElementException
      {
        E x = stack.pop(); // may throw an exception
        size--;
        return x;
      }
      public int size ()
      {
        return size;
      }
    };
```

FIGURE 4.25 – Décoration de `IStack<E>` vers `IRichStack<E>`

à partir d'un objet « pile » déjà créé et qui contient peut-être déjà des éléments, on cherche à construire un nouvel objet « pile », qui « décore » l'ancien en lui ajoutant une méthode `size`. Or, nous nous intéressons à un scénario plus restrictif : c'est seulement **au moment de la création d'une nouvelle pile** vide que nous souhaitons effectuer cette décoration. C'est pourquoi il faut construire un objet de type `Supplier<IRichStack<E>>` à partir d'un objet de type `Supplier<IStack<E>>`. En bref, il ne s'agit pas de changer une pile en une pile enrichie, mais de changer une usine à piles en une usine à piles enrichies.

Une solution est donnée dans la Figure 4.25. La méthode `enrich` est statique et polymorphe vis-à-vis de `E`. Elle doit renvoyer un objet de type `Supplier<IRichStack<E>`. Pour construire cet objet, nous employons la syntaxe des clôtures `() -> ...` puis la syntaxe des classes anonymes `new IRichStack<E> () { ... }`. Il reste à définir les champs et les méthodes de notre objet « pile » enrichi. Comme dans la troisième partie de l'Exercice 2.2, nous utilisons la **délégation**. Nous prévoyons donc un champ `stack`, qui contient (un pointeur vers) un objet de type `IStack<E>`, produit par un appel initial à `supplier.get()`. Nous y ajoutons un champ `size`, initialisé à 0, qui doit contenir à tout instant le nombre d'éléments de la pile. Les méthodes `push`, `pop` et `size` s'écrivent alors aisément.

Nous avons **emballé** un objet (ici, la pile ordinaire) dans un autre (ici, la pile enrichie) et **délégué** certains appels de méthodes du second vers le premier. Ce motif est parfois appelé **decorator pattern**. On trouve aussi les mots **adapter**, **wrapper**, **proxy**, **bridge**, **facade**, et d'autres encore, pour décrire des motifs très similaires. ◇

Solution de l'exercice 3.20 L'interface `Hashable` doit exiger la présence des méthodes `equals` et `hashCode`, et doit donner les types de leurs arguments (en dehors de l'argument implicite `this`) et de leur résultat. On la définit ainsi :

```
public interface Hashable<T> {
  boolean equals (T that);
  int hashCode ();
}
```

L'interface `Hashable` doit être paramétrée par un type `T`, qui est le type de l'argument de `equals`.

Sans ce paramètre `T`, nous serions obligés de déclarer que `equals` attend un argument de type `Object`, ce qui serait trop exigeant. Nous n'avons pas besoin qu'une clef soit comparable à une pomme ou à une orange ; il nous suffit (pour implémenter nos tables de hachage) qu'elle soit comparable à d'autres clefs.

Ceci posé, nous pouvons donner l'en-tête de la définition de la classe `HashMap`. Comme la véritable classe `HashMap`, elle est paramétrée par les types `K` et `V` des clefs et des valeurs. Cependant, elle doit être restreinte aux clefs que l'on peut hacher et comparer à d'autres clefs. Il faut donc la restreindre aux types `K` qui sont sous-types de `Hashable<K>`. On écrit :

```
public class HashMap<K extends Hashable<K>, V> {
  ...
}
```

Comme nous l'avons noté dans le Détail 3.8, il serait préférable (car un peu moins restrictif) d'exiger seulement `K extends Hashable<? super K>` plutôt que `K extends Hashable<K>`. ◇

Solution de l'exercice 3.21 La méthode `addAll` souffre du fait que son argument `c` est de type `Collection<E>`, ce qui implique que l'on ne sait pas comment `c` est représenté en mémoire. L'objet `c` peut appartenir à n'importe quelle classe qui implémente l'interface `Collection`. Ce peut donc être une liste, une table de hachage, etc. Certes, on sait que `c` possède les méthodes exigées par l'interface `Collection`, mais rien de plus.

Par conséquent, la seule façon d'implémenter `addAll` consiste à énumérer les éléments de `c` pour les ajouter à `this`. On peut écrire cela sous la forme suivante :

```
public boolean addAll (MyCollection<E> c) {
  Iterator<E> it = c.iterator();
  boolean changed = false;
  while (it.hasNext()) {
    E e = it.next();
    changed |= this.add(e);
  }
  return changed;
}
```

En Java 8, on pourrait fournir cette implémentation sous la forme d'une méthode par défaut au sein même de l'interface `Collection`. Il suffirait alors d'implémenter les autres méthodes pour obtenir automatiquement une implémentation de `addAll`.

Supposons que la complexité de `this.add` est $O(f(m))$, où m est le nombre d'éléments de la collection `this`. Par exemple, si `this` est une table de hachage, on aura $f(m) = 1$; si `this` est un arbre binaire de recherche, on aura $f(m) = \log m$. Supposons (c'est une hypothèse optimiste) que `iterator`, `hasNext` et `next` ont une complexité $O(1)$. Dans ce cas, la complexité de `addAll` est $O(n.f(m+n))$, où m et n sont les nombres d'éléments des collections `this` et `c`. Ce n'est pas déraisonnable, mais si `this` et `c` sont en réalité des structures conçues pour permettre une union efficace, alors c'est décevant. On peut penser aux « *mergeable maps* », aux « *mergeable heaps* », aux « *catenable deques* », etc.

Pour éviter ce problème, définissons une variante de l'interface `Collection`, où l'on contrôle précisément le type de l'argument `c` de la méthode `addAll` :

```
public interface PreciseCollection<E, Self> {
  boolean add (E e);
  boolean contains (E e);
  boolean addAll (Self c);
}
```

On donne à `c` le type `Self`, et comme on ne sait pas ici ce que sera `Self`, on en fait un paramètre de l'interface `PreciseCollection`. Notre algorithme peut alors être écrit sous la forme suivante :

```
class MyAlgorithm<E, C extends PreciseCollection<E, C>> {

  private final Supplier<C> makeCollection;

  public MyAlgorithm (Supplier<C> makeCollection) {
    this.makeCollection = makeCollection;
  }

  public void run () {
    ...
    C c1 = makeCollection.get();
    C c2 = makeCollection.get();
    ...
    c1.addAll(c2);
    ...
  }

}
```

L'algorithme est utilisable quel que soit le type `E` des éléments et quel que soit le type `C` choisi pour représenter les ensembles d'éléments, à condition que `C` implémente l'interface `PreciseCollection<E, C>`. Nous exigeons ainsi que, si `c1` et `c2` sont deux ensembles de type `C`, alors l'appel `c1.addAll(c2)` soit permis. Cependant, nous n'exigeons pas cela dans le cas où `c2` serait une collection arbitraire de type `Collection<E>`. Nous laissons donc la porte ouverte à une implémentation efficace de `addAll`, qui s'appuie sur le fait que les deux ensembles considérés sont de type `C`.

Afin de permettre à notre algorithme de créer autant d'objets de type `C` que nécessaire, nous le paramétrons par une **usine** ou **factory** de type `Supplier<C>`. Ainsi, chaque appel à `makeCollection.get()` fournit un nouvel ensemble vide de type `C`. La méthode `run` ci-dessus illustre la manière dont l'algorithme peut utiliser les ensembles. ◇

Solution de l'exercice 3.22 La documentation du mystérieux constructeur `TreeMap ()` indique que « les clefs insérées dans la table doivent implémenter l'interface `Comparable` ». Elle indique de plus que, pour comparer deux clefs `k1` et `k2`, un appel à `k1.compareTo(k2)` est utilisé. C'est donc une méthode `compareTo` qui détermine la relation d'ordre.

Ce qui est étrange, toutefois, c'est que rien n'oblige le type `K` à posséder une méthode `compareTo`, et même s'il en possède une, rien n'oblige le type de son argument à être `K`.

Dans ces conditions, le code de la classe `TreeMap` doit vérifier, pendant l'exécution, que chaque clef `key` qui est insérée dans la table possède bien une méthode `compareTo` et que cette méthode est applicable à des clefs de type `K`. Ces vérifications sont effectuées dans la méthode `put`. On y trouve par exemple l'instruction suivante :

```
@SuppressWarnings("unchecked")
Comparable<? super K> k = (Comparable<? super K>) key;
```

qui a pour effet de vérifier que `key` appartient à une classe qui implémente bien `Comparable<U>` pour un certain type `U`. Si ce n'est pas le cas, l'exception `ClassCastException` est lancée. De plus, pour des raisons techniques (Naftalin et Wadler, 2006) et contrairement à ce que l'on pourrait croire, ce test ne vérifie pas que `U` est super-type de `K`. Il faut donc de plus vérifier que la méthode `key.compareTo` est applicable à un argument de type `K`. On trouve dans le code de la méthode `put` l'instruction suivante :

```
key.compareTo(key); // type (and possibly null) check
```

Cette instruction lance `NullPointerException` si `key` est `null`, et lance `ClassCastException` si `key.compareTo` n'est pas applicable à une clef.

En conclusion, l'écriture de la classe `TreeMap` est discutable. Si une classe `A` n'implémente pas l'interface `Comparable`, alors l'expression `new TreeMap<A> ()` est acceptée. L'erreur n'est

détectée que pendant l'exécution du programme : la première tentative d'insérer une clef dans cette table provoque le lancement d'une `ClassCastException`.

Il aurait été préférable, quoiqu'un peu plus lourd, de ne proposer qu'un seul constructeur. Deux choix étaient possibles. Soit on acceptait un type `K` arbitraire, et on exigeait lors de la construction de la table un comparateur de type `Comparator<K>`. Soit on imposait la contrainte `K extends Comparable<? super K>>` et on n'exigeait rien lors de la construction de la table. Les auteurs de la bibliothèque ont voulu allier les avantages de ces deux approches, mais cela n'a pas vraiment de sens. ◇

Solution de l'exercice 4.1 L'écriture de `iter` est aisée. On conserve la traversée récursive de l'arbre, et on appelle `f` une fois pour chaque élément.

```
let rec iter t f =
  match t with
  | Leaf ->
      ()
  | Node (t1, x, t2) ->
      iter t1 f;
      f x;
      iter t2 f
```

L'ordre dans lequel on effectue les trois appels est important. Ici, on appelle `f x` entre les deux appels récursifs (d'où un ordre « infixe »), et on effectue le parcours du sous-arbre `t1` avant celui du sous-arbre `t2` (d'où un ordre « de gauche à droite »).

Pour ré-implémenter `elements` à l'aide de `iter`, il faut nécessairement stocker, pendant l'itération, une liste partiellement construite. Pour cela, on utilise une référence, dont on modifie le contenu à chaque itération. On peut écrire :

```
let elements t =
  let xs = ref [] in
  iter t (fun x -> xs := x :: !xs);
  List.rev !xs
```

Le dernier élément rencontré étant ajouté en tête de liste, la liste est naturellement construite à l'envers ; aussi on utilise `List.rev` pour la renverser une fois construite. Si l'on juge que cela n'est pas très satisfaisant, alors il faut employer une variante symétrique de `iter`, qui présente les éléments dans l'ordre infixe de droite à gauche. L'appel à `List.rev` ne sera alors plus nécessaire. ◇

Solution de l'exercice 4.2 Nous présentons d'abord une solution en OCaml (Figure 4.26). La fonction principale, `for_every_permutation`, alloue deux tableaux, à savoir un tableau d'entiers `pi` qui contient une permutation partiellement construite et un tableau de Booléens `busy` qui mémorise quels entiers apparaissent dans l'image de la permutation partielle `pi`. La fonction `for_every_permutation` fait ensuite appel à `enumerate 0`, où `enumerate` est une fonction récursive locale, qui, de par sa situation, a accès aux tableaux `pi` et `busy`.

Au moment où `enumerate i` est appelée, la permutation `pi` a été construite (i.e., le tableau `pi` a été initialisé) pour les indices strictement inférieurs à `i`. le rôle de `enumerate i` est d'énumérer toutes les extensions de cette permutation partielle et de les présenter au consommateur.

Si `i` vaut `n`, alors la permutation est complète et peut être présentée au consommateur `f`. Dans le cas contraire, il faut choisir une valeur pour `pi.(i)`. On énumère donc (à l'aide d'une boucle) toutes les valeurs `j` qui ne sont pas déjà utilisées par cette permutation partielle. Pour chaque choix de `j`, on étend `pi`, puis on effectue un appel récursif `enumerate (i + 1)` pour énumérer toutes les extensions de `pi` ainsi étendue.

Notre solution en Java est analogue (Figure 4.27). Comme Java n'offre pas de syntaxe concise pour définir une fonction récursive locale, nous utilisons une méthode récursive au sein d'une classe auxiliaire (privée) `Enumerate`. Le code est verbeux car il faut déclarer et initialiser les quatre champs de cette classe. Cependant, si l'on fait abstraction de ces détails syntaxiques, ce code est essentiellement identique au code OCaml. ◇

```
let for_every_permutation n f =
  let pi = Array.make n 0 in
  let busy = Array.make n false in
  let rec enumerate i =
    if i = n then
      f pi
    else
      for j = 0 to n - 1 do
        if not busy.(j) then begin
          pi.(i) <- j;
          busy.(j) <- true;
          enumerate (i + 1);
          busy.(j) <- false
        end
      done
  in
  enumerate 0
```

FIGURE 4.26 – Un réducteur pour énumérer les permutations (OCaml)

Solution de l'exercice 4.3 On peut construire ce type mentalement, ou bien demander au compilateur OCaml de l'inférer :

```
val on_interval_accu : int -> int -> (int -> 'a -> 'a) -> 'a -> 'a
```

En bref, `on_interval_accu` est une fonction à 4 arguments, et son troisième argument `f` est lui-même une fonction à deux arguments. Ici `'a` est le type de l'accumulateur ; nous l'avons noté `'state` dans le texte qui précède cet exercice. ◇

Solution de l'exercice 4.4 En principe, l'énoncé doit être clair. Il s'agit d'écrire un producteur qui, étant donné un tableau `a`, énumère les éléments de ce tableau, et les soumet à un consommateur `f`, tout en maintenant un accumulateur `accu`.

A priori, le type `E` des éléments du tableau peut être quelconque. Le type `S` de l'état (ou accumulateur) peut être quelconque également. Notre réducteur sera donc polymorphe vis-à-vis de ces deux types.

Le consommateur reçoit un élément, un état, et renvoie un nouvel état. Son type, en syntaxe OCaml, serait `E -> S -> S`. En Java, nous devons employer une **interface fonctionnelle** pour indiquer que le consommateur doit avoir une méthode à deux paramètres de types `E` et `S` et un résultat de type `S`. Nous pouvons employer pour cela l'interface `BiFunction` de Java 8. Le type du consommateur s'écrit alors `BiFunction<E,S,S>`.

Le producteur ou réducteur reçoit un tableau, un consommateur, et un état initial :

```
import java.util.function.BiFunction;
public class ArrayReduce {
  public static<E,S> S reduce
  (E[] a, BiFunction<E,S,S> f, S accu) {
    for (int i = 0; i < a.length; i++)
      accu = f.apply(a[i], accu);
    return accu;
  }
}
```

Une simple boucle est utilisée pour énumérer les éléments du tableau et les soumettre au consommateur. La variable locale `accu`, qui est modifiable, est utilisée pour stocker l'état courant.

Pour calculer la somme des éléments d'un tableau d'entiers `a`, on écrit :

```
import java.util.function.Consumer;

public class Permutation {

  public static void forEveryPermutation (int n, Consumer<int[]> f) {
    Enumerate e = new Enumerate (n, f);
    e.enumerate(0);
  }

  private static class Enumerate {

    final int n;
    final Consumer<int[]> f;
    final int[] pi;
    final boolean[] busy;

    Enumerate (int n, Consumer<int[]> f) {
      this.n = n;
      this.f = f;
      pi = new int [n];
      busy = new boolean [n];
    }

    void enumerate (int i) {
      if (i == n)
        f.accept(pi);
      else
        for (int j = 0; j < n; j++)
          if (!busy[j]) {
            pi[i] = j;
            busy[j] = true;
            enumerate(i + 1);
            busy[j] = false;
          }
    }

  }

}
```

FIGURE 4.27 – Un réducteur pour énumérer les permutations (Java)

```
int sum = ArrayReduce.reduce(a, ((x, y) -> x + y), 0);
```

Notons toutefois que le tableau `a` doit avoir le type `Integer[]`, et non pas `int[]`, sans quoi notre code polymorphe n'est pas applicable (Détail 3.2). ◇

Solution de l'exercice 4.5 La fonction `fold` a la même structure que la fonction `iter` de l'Exercice 4.1. La seule modification à apporter est l'ajout de l'accumulateur `accu`, qui est argument et résultat de `f` et argument et résultat de `fold`.

```
let rec fold t f accu =
  match t with
  | Leaf ->
      accu
  | Node (t1, x, t2) ->
      let accu = fold t1 f accu in
      let accu = f x accu in
      let accu = fold t2 f accu in
      accu
```

Pour calculer le plus grand élément d'un arbre `t`, on écrit :

```
fold t max min_int
```

On exploite ici la fonction `max`, définie dans la bibliothèque d'OCaml, dont l'élément neutre est le plus petit entier représentable en machine, nommé `min_int`. ◇

Solution de l'exercice 4.6 Les fonctions `fold_left` et `fold_right` diffèrent en ce que la première appelle d'abord le consommateur `f`, puis effectue un appel à elle-même pour continuer l'itération ; tandis que la seconde effectue ces appels dans l'ordre inverse.

```
let rec fold_left f accu xs =
  match xs with
  | [] ->
      accu
  | x :: xs ->
      let accu = f accu x in
      let accu = fold_left f accu xs in
      accu

let rec fold_right f xs accu =
  match xs with
  | [] ->
      accu
  | x :: xs ->
      let accu = fold_right f xs accu in
      let accu = f x accu in
      accu
```

On a suivi la convention de la bibliothèque `List` selon laquelle dans `fold_left` l'accumulateur est l'avant-dernier argument, tandis que dans `fold_right` il est le dernier argument. Cette convention ne semble pas avoir de profonde raison d'être. Elle est la même en Haskell : voir `foldl` versus `foldr`.

La complexité en temps de ces deux fonctions est la même, à savoir $O(n)$, où n est la longueur de la liste. Il faut ajouter à cela le temps exigé par les appels au consommateur `f`.

La complexité en espace n'est pas la même pour ces deux fonctions. Parce que `fold_left` effectue un appel **récursif terminal**, elle est traduite en langage machine sous la forme d'une boucle, et exige un espace $O(1)$ sur la pile. Au contraire, `fold_right` effectue un appel récursif non terminal, puisque, après cet appel, il reste à effectuer un appel à `f`. Chaque appel à `fold_right` consomme un espace $O(1)$ sur la pile, et, ces appels étant imbriqués les uns dans les autres, l'espace total nécessaire sur la pile est $O(n)$.

Ce résultat ne doit pas sembler surprenant. Une liste **simplement chaînée** ne peut être parcourue facilement que dans le sens des pointeurs, de gauche à droite ; c'est ce que fait `fold_left`. Si on veut la parcourir en sens inverse, de droite à gauche, alors il faut d'abord la renverser. C'est ce que fait (implicitement) `fold_right` : elle parcourt d'abord la liste jusqu'au bout tout en empilant les n éléments de la liste ; puis elle revient sur ses pas et dépile ces éléments, tout en les soumettant à `f` au fur et à mesure qu'elle les dépile. La pile sert donc d'espace temporaire pour stocker une version renversée de la liste.

Pour ré-implémenter `fold_right` en utilisant un espace $O(1)$, on peut composer `fold_left` avec la fonction `rev`, qui renverse une liste :

```
let rev xs =
  fold_left (fun accu x -> x :: accu) [] xs
let fold_right f xs accu =
  fold_left (fun accu x -> f x accu) accu (rev xs)
```

La fonction `rev` est elle-même implémentée à l'aide de `fold_left` ! Il en découle que ses complexités en temps et en espace sont bien $O(n)$ et $O(1)$ respectivement. Par conséquent, cette nouvelle version de `fold_right`, qui effectue deux appels successifs à `fold_left`, a elle aussi pour complexités en temps et en espace $O(n)$ et $O(1)$ respectivement. ◇

Solution de l'exercice 4.7 La fonction `eval` s'écrit très aisément sous forme récursive :

```
let rec eval e =
  match e with
  | Const i ->
      i
  | Sum (e1, e2) ->
      eval e1 + eval e2
  | Prod (e1, e2) ->
      eval e1 * eval e2
```

La fonction d'évaluation ci-dessus fixe l'interprétation des constantes, de la somme et du produit. L'interprétation des constantes est l'identité : l'entier `i` est renvoyé directement, sans être soumis à aucun calcul. L'interprétation de la somme et du produit symboliques (représentés par les constructeurs `Sum` et `Prod`) sont la somme et le produit des entiers (représentés par les opérations `+` et `*` d'OCaml).

Pour généraliser `eval` et obtenir `fold`, il suffit d'abstraire (c'est-à-dire de paramétrer) le code vis-à-vis de ces trois interprétations. Supposons que l'on nous donne trois fonctions `const`, `sum` et `prod`. À chaque nœud de l'arbre, on utilise alors l'une de ces trois fonctions pour effectuer le calcul approprié :

```
let rec fold const sum prod e =
  match e with
  | Const i ->
      const i
  | Sum (e1, e2) ->
      sum
        (fold const sum prod e1)
        (fold const sum prod e2)
  | Prod (e1, e2) ->
      prod
        (fold const sum prod e1)
        (fold const sum prod e2)
```

Si ce code semble un peu lourd, on peut préférer l'écrire ainsi :

```
let fold const sum prod e =
  let rec eval e =
    match e with
    | Const i ->
```

```
        const i
    | Sum (e1, e2) ->
        sum (eval e1) (eval e2)
    | Prod (e1, e2) ->
        prod (eval e1) (eval e2)
  in
  eval e
```

Le compilateur OCaml nous donne (s'il était besoin) le type de `fold` :

```
val fold:
  (int -> 'a) ->
  ('a -> 'a -> 'a) ->
  ('a -> 'a -> 'a) ->
  expr -> 'a
```

La valeur calculée n'est pas nécessairement un entier ; son type est `'a`. On lit que la fonction `const`, qui représente l'interprétation des constantes, doit avoir le type `int -> 'a` ; tandis que les fonctions `sum` et `prod` doivent être des opérateurs binaires sur le type `'a`. Une fois appliquée à ces trois arguments, la fonction `fold` attend encore une expression et l'évalue, produisant un résultat final de type `'a`.

On retrouve la fonction d'évaluation ordinaire en appliquant `fold` aux interprétations standard des constantes et des deux opérations :

```
let eval : expr -> int =
  fold (fun i -> i) (fun x y -> x + y) (fun x y -> x * y)
```

Une interprétation dans $\mathbb{Z}/n\mathbb{Z}$ est obtenue ainsi :

```
let eval_modulo (n : int) : expr -> int =
  fold
    (fun i -> i mod n)
    (fun x y -> (x + y) mod n)
    (fun x y -> (x * y) mod n)
```

Une interprétation dans $(\mathbb{Z}, \max, +)$ est obtenue ainsi :

```
let eval_maxplus : expr -> int =
  fold (fun i -> i) max (+)
```

Pour conclure, notons que la fonction `fold` expose à l'utilisateur toute la structure de l'arbre. En effet, à chaque nœud de l'arbre, la fonction fournie par l'utilisateur pour interpréter ce type de nœud est appliquée. Si `t` est l'arbre `Sum (Const 3, Prod (Const 2, Const 4))`, par exemple, alors l'expression :

```
fold const sum prod t
```

est équivalente à :

```
sum (const 3) (prod (const 2) (const 4))
```

En ce sens, `fold` est une fonction d'évaluation « *bottom-up* » canonique.

La fonction `List.fold_right` (Exercice 4.6) était en réalité, de la même manière, une fonction d'évaluation « *bottom-up* » canonique pour les listes. Si on considère le type des listes comme un type d'arbres particulier, alors la construction générale de la fonction `fold` que nous avons illustrée ci-dessus donne `List.fold_right`.

La fonction `fold` sur les arbres que nous avons définie lors de l'Exercice 4.5, au contraire, n'était pas une fonction d'évaluation canonique pour les arbres. En effet, elle n'expose à l'utilisateur que la liste des éléments de l'arbre (dans un certain ordre), et non pas la structure de l'arbre. ◇

Solution de l'exercice 4.8 La classe `Expr` est définie initialement comme une classe abstraite :

```
public abstract class Expr {}
```

Ses sous-classes définissent des champs immuables et un constructeur trivial :

```
public class Const extends Expr {
  public final int i;
  public Const (int i) { this.i = i; }
}
public class Sum extends Expr {
  public final Expr e1, e2;
  public Sum (Expr e1, Expr e2) { this.e1 = e1; this.e2 = e2; }
}
```

On omet la classe `Prod`, en tous points analogue à `Sum`.

Pour définir l'interprétation standard des expressions, on déclare dans la classe parent une méthode `eval` abstraite :

```
public abstract class Expr {
  public abstract int eval ();
}
```

Puis, on définit cette méthode dans chacune de sous-classes :

```
public class Const extends Expr {
  ...
  public int eval () { return i; }
}
public class Sum extends Expr {
  ...
  public int eval () { return e1.eval() + e2.eval(); }
}
```

On peut ajouter, si on le souhaite, l'annotation `@Override`. (Ici, ce n'est pas essentiel. Les classes `Const`, `Sum` et `Prod` n'étant pas abstraites, le compilateur Java va de toute façon vérifier que la méthode `eval` est bien définie dans chacune de ces classes.)

Pour abstraire l'évaluation vis-à-vis de l'interprétation des opérations, définissons d'abord une interface `Interpretation<A>`. Cette interface décrit ce que le client doit fournir : une interprétation de chaque construction. Le schéma adopté ici est exactement le même que dans l'Exercice 4.7, à ceci près qu'au lieu de demander au client de fournir trois fonctions `const`, `sum` et `prod`, on lui demande de fournir un objet doté de trois méthodes.

```
public interface Interpretation<A> {
  A constant (int i);
  A sum (A a1, A a2);
  A prod (A a1, A a2);
}
```

Nous pouvons alors déclarer la méthode `fold` dans la classe parent :

```
public abstract class Expr {
  public abstract <A> A fold (Interpretation<A> interpretation);
}
```

et la définir dans chacune des sous-classes :

```
public class Const extends Expr {
  ...
  public <A> A fold (Interpretation<A> interpretation) {
    return interpretation.constant(i);
  }
}
public class Sum extends Expr {
```

```
  ...
  public <A> A fold (Interpretation<A> interpretation) {
    return interpretation.sum(
      e1.fold(interpretation),
      e2.fold(interpretation)
    );
  }
}
```

L'interprétation dans les Booléens décrite dans l'énoncé est définie comme suit :

```
Interpretation<Boolean> i = new Interpretation<Boolean> () {
  public Boolean constant (int i) { return i != 0; }
  public Boolean sum (Boolean b1, Boolean b2) { return b1 || b2; }
  public Boolean prod (Boolean b1, Boolean b2) { return b1 && b2; }
};
```

Pour évaluer une expression `e` suivant cette interprétation, il suffit d'appeler `e.fold(i)`.

Ce motif de programmation est connu dans la littérature sous le nom de **visitor pattern**. Notre objet de type `Interpretation<A>` est traditionnellement appelé **visiteur**. Notre méthode `fold` est traditionnellement nommée `accept`.

Nous avons décrit ici un cas particulier de ce motif ; il en existe d'autres. Par exemple, pour plus de généralité, il serait bon que la méthode `constant` ait accès non pas seulement au champ `i` de l'objet `Const`, mais à cet objet lui-même ; de même, les méthodes `sum` et `prod` gagneraient à avoir accès non pas seulement aux valeurs produites par les appels récursifs, mais aussi à l'objet `Sum` ou `Prod`. On pourrait donc modifier ainsi l'interface `Interpretation<A>` :

```
public interface Interpretation<A> {
  A constant (Const e);
  A sum (Sum e, A a1, A a2);
  A prod (Prod e, A a1, A a2);
}
```

En Java, on peut de plus tirer parti de la **surcharge** (qui autorise plusieurs méthodes à porter le même nom pourvu que leurs signatures soient différentes) et renommer alors les méthodes `constant`, `sum` et `prod` toutes trois en `visit`.

La façon dont il faut alors modifier la méthode `fold` est laissée en exercice au lecteur. ◇

Solution de l'exercice 4.9 Soulignons d'abord ce que demande l'énoncé : il faut définir une fonction `interval` à deux arguments `i` et `j` qui renvoie un itérateur, c'est-à-dire une fonction à un argument `()` qui, à chaque fois qu'elle est appelée, produit le « prochain » élément de l'intervalle $[i, j[$. Cet itérateur doit être **nouveau** au sens où son état interne, modifiable, doit être distinct de l'état interne de tous les itérateurs que l'on a pu créer précédemment.

Quel doit être cet état interne ? Il semble clair qu'il faut (et il suffit de) stocker la valeur du prochain entier qui sera produit par l'itérateur. Lorsque `interval i j` est appelée, on crée donc immédiatement une nouvelle référence, nommée `next`, initialisée à la valeur `i`. Puis on définit une fonction (anonyme), qui constitue l'itérateur lui-même ; et on renvoie cette fonction.

```
let interval i j : int iterator =
  let next = ref i in
  fun () ->
    let c = !next in
    if c < j then begin
      next := c + 1;
      Some c
    end
    else
      None
```

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class IntervalIterator implements Iterator<Integer> {

  public static Iterator<Integer> interval (int i, int j) {
    return new IntervalIterator (i, j);
  }

  private final int j;
  private int next;

  private IntervalIterator (int i, int j) {
    this.next = i;
    this.j = j;
  }

  public Integer next () {
    if (next < j)
      return next++;
    else
      throw new NoSuchElementException ();
  }

  public boolean hasNext () {
    return next < j;
  }

}
```

FIGURE 4.28 – Un itérateur sur un intervalle

La référence `next` peut être indirectement consultée (et éventuellement modifiée) via un appel à l'itérateur : tant que `!next` n'a pas atteint la valeur `j`, chaque appel à l'itérateur incrémente `next`. De plus, cette référence est accessible uniquement via l'itérateur : le code situé à l'extérieur ne peut même pas nommer cette référence, puisque `next` est une variable locale de la fonction `interval`. Nous avons donc la certitude que la valeur de `next` est comprise entre `i` et `j`, et ne peut que croître avec le temps. Ceci est un exemple d'**invariant** garanti par une barrière d'abstraction. ◇

Solution de l'exercice 4.10 Rappelons que `Iterator` est une interface, et non une classe. L'objet renvoyé par la méthode `interval` doit donc appartenir à une classe qui implémente l'interface `Iterator<Integer>`. Nommons cette classe `IntervalIterator` (Figure 4.28).

La méthode statique `interval` peut être située soit elle aussi dans cette classe, soit dans une autre classe ; cela n'a pas d'importance. Quoiqu'il en soit, le code de cette méthode est trivial : elle construit et renvoie un nouvel objet de classe `IntervalIterator`. Cet objet est doté d'un état interne modifiable, représenté par son champ `next`, et de deux méthodes `next` et `hasNext`.

La méthode `interval` de la Figure 4.28 est tout à fait similaire à la fonction `interval` de l'Exercice 4.9, qui construit et renvoie une fonction dotée d'un état interne modifiable. Cette analogie entre clôtures OCaml et objets Java a été étudiée en détail plus tôt ; voir par exemple l'Exercice 1.7. ◇

Solution de l'exercice 4.11 On doit définir une classe `Interval` qui implémente l'interface `Iterable<Integer>` (Figure 4.29). Le code est malheureusement sans grand intérêt. Il faut doter cette classe de deux champs immuables `i` et `j` ainsi que d'un constructeur qui initialise ces champs. La méthode `iterator` est implémentée via un appel à la méthode statique

```
import java.util.Iterator;

public class Interval implements Iterable<Integer> {

  final int i;
  final int j;

  public Interval (int i, int j) {
    this.i = i;
    this.j = j;
  }

  public Iterator<Integer> iterator () {
    return IntervalIterator.interval(i, j);
  }

}
```

FIGURE 4.29 – Une usine à itérateurs sur un intervalle

`IntervalIterator.interval` de l'Exercice 4.10.

Pour afficher les éléments de l'intervalle [0, 10[, on écrit alors :

```
for (int x : new Interval (0, 10))
  System.out.println(x);
```

Ici, bien sûr, on n'a rien gagné vis-à-vis de la boucle primitive `for (int x = 0; x < 10; x++) { ... }`. On a même perdu un facteur constant en efficacité. Néanmoins, ce style devient intéressant si on remplace `Interval` par un producteur plus complexe. ◇

Solution de l'exercice 4.12 La fonction `on_iterator` s'écrit naturellement sous forme récursive terminale :

```
let rec on_iterator (it : 'a iterator)
                    (f : 'a -> 'b -> 'b)
                    (accu : 'b) : 'b =
  match it() with
  | None ->
      accu
  | Some x ->
      let accu = f x accu in
      on_iterator it f accu
```

Elle appelle d'abord l'itérateur `it` pour obtenir un élément, s'il en reste : l'expression `it()` a le type `'a option`. Il faut alors examiner cette option pour déterminer si elle est étiquetée `None` ou `Some`. Dans le premier cas, il n'y a plus d'éléments. L'itération est terminée : on renvoie `accu`, qui est la valeur actuelle (donc finale) de l'accumulateur. Dans le second cas, on a obtenu un élément `x`. On le soumet à la fonction `f`, ce qui produit un nouvel accumulateur. Puis on effectue un appel récursif (terminal) pour continuer l'itération. Ce code est analogue à la boucle `while (it.hasNext()) { ... }` de Java.

Pour afficher les éléments de l'intervalle [0, 10[, on écrit ceci :

```
on_iterator (interval 0 10) (fun x () ->
  Printf.printf "%d\n" x
) ()
```

Ce code est analogue à une « boucle foreach » de Java. Enfin, si on souhaite calculer une somme, on peut écrire :

```
let sum =
  on_iterator (interval 0 10) (fun x accu ->
    x + accu
  ) 0
```

ou bien, de façon plus concise :

```
let sum =
  on_iterator (interval 0 10) (+) 0
```

Ici, bien sûr, on n'a rien gagné vis-à-vis d'un simple appel à `on_interval` (§4.1.1). Le véritable intérêt des itérateurs n'apparaît que dans des scénarios plus complexes où ne consomme pas d'un coup tous les éléments de la séquence. ◇

Solution de l'exercice 4.13 La définition concrète du type `'a array_iterator` doit décrire l'état (modifiable) de l'itérateur. Or, de quoi celui-ci a-t-il besoin ? Il doit mémoriser d'une part un pointeur vers le tableau `a`, car celui-ci est fourni lors de l'appel à `create`, mais n'apparaît pas parmi les arguments de `next`. Il doit avoir accès d'autre part à une référence contenant l'indice du prochain élément.

```
type 'a array_iterator =
  'a array * int ref
```

Ceci posé, les fonctions `create` et `next` sont simples. La première initialise l'état. La seconde consulte l'état et le met à jour si nécessaire.

```
let create (a : 'a array) : 'a array_iterator =
  let state = ref 0 in
  (a, state)

let next ((a, state) : 'a array_iterator) : 'a option =
  let i = !state in
  if i < Array.length a then begin
    state := i + 1;
    Some a.(i)
  end
  else
    None
```

Cet itérateur sur un tableau ressemble fort à l'itérateur sur un intervalle de l'Exercice 4.9. Nous verrons d'ailleurs qu'on peut déduire le premier du second à l'aide d'une fonction `map` sur les itérateurs (Exercice 4.20).

Peut-on appliquer `on_iterator` (Exercice 4.12) à un objet de type `'a array_iterator` ? Non, bien sûr, ou du moins pas directement, puisque `on_iterator` attend un argument de type `'a iterator`, qui est synonyme de `unit -> 'a option`. Un objet de type `'a array_iterator` n'est pas une fonction. Vu de l'extérieur, c'est un objet dont le type est opaque ; on ne peut rien en faire, sauf le passer à la fonction `next`.

Peut-on écrire une conversion du type `'a array_iterator` vers le type `'a iterator` ? Oui, heureusement. C'est même très simple.

```
let convert_array_iterator (it : 'a array_iterator) : 'a iterator =
  fun () -> next it
```

On pourrait penser, de ce fait, qu'il n'est pas trop grave de ne pas pouvoir appliquer la fonction `on_iterator` à un objet `it` de type `'a array_iterator`, car au lieu d'écrire `on_iterator it`, on peut écrire `on_iterator (convert_array_iterator it)`.

Mais cela souligne en réalité un problème réel et profond : le type `'a array_iterator` n'est pas suffisamment abstrait. Certes, il cache la manière dont un itérateur sur un tableau est implémenté ; mais il révèle encore (de par son nom !) que nous avons affaire à un itérateur **sur un tableau**. Si l'on suivait le style de programmation proposé par Arthur, on aboutirait à définir

un type `'a list_iterator` pour les itérateurs sur les listes, un type `'a hashtbl_iterator` pour les itérateurs sur les tables de hachage, etc. Il serait alors impossible d'implémenter un consommateur composable directement avec n'importe lequel de ces producteurs.

Le type `'a iterator`, au contraire, est suffisamment abstrait. Il est **universel**, au sens où tous les itérateurs admettent ce type, quelle que soit la manière dont ils sont implémentés. Une fonction qui attend un argument de type `'a iterator` est directement applicable à n'importe quel producteur.

Cet exemple illustre le fait qu'un **type de fonction est plus abstrait qu'un type abstrait** (!), car un type de fonction est anonyme (**il décrit le service rendu, et rien de plus**) tandis qu'un type abstrait porte un nom (par exemple `array_iterator`, `hashtbl_iterator`, etc.) qui trahit la provenance de l'objet et nous interdit de remplacer un objet par un autre, même si tous deux rendent le même service.

Cette même remarque est valable en Java et dans les langages orientés objets. L'interface `Iterator` est analogue à un type de fonction : elle décrit le service rendu, et rien de plus. Les classes qui implémentent cette interface, par exemple `IntervalIterator`, `ListIterator`, portent un nom qui trahit la provenance de l'itérateur. **Une interface est donc plus abstraite qu'une classe**, ce qui n'est pas surprenant.

Parce que les types de fonctions d'OCaml et les interfaces de Java décrivent un service, et rien de plus, ils permettent de **remplacer facilement un objet par un autre**, à condition que ces deux objets rendent le même service. Selon certains, c'est là l'essence de la programmation orientée objets (Aldrich, 2013). En fait, c'est aussi l'essence de la programmation fonctionnelle, au sens où on l'entend en OCaml, c'est-à-dire « programmation où les fonctions sont des valeurs comme les autres ». Ces deux paradigmes de programmation ne sont pas fondamentalement éloignés l'un de l'autre. ◇

Solution de l'exercice 4.14 Cet exercice illustre une difficulté que l'on rencontre fréquemment lorsque l'on doit écrire un itérateur pour une structure de données arborescente. On aimerait naturellement écrire le parcours de l'arbre sous forme récursive, mais on ne peut pas le faire, car le parcours doit s'interrompre dès qu'un élément est atteint, et doit pouvoir être repris par la suite. Pour cela, il est nécessaire d'utiliser une pile explicite. On peut écrire ce code sous diverses formes. En voici une, où on a choisi de maintenir une liste explicite des éléments et des arbres restant à énumérer. On définit ainsi le type de cette liste :

```
type stack =
  | SEmpty
  | SElem of int * stack
  | STree of tree * stack
```

Il s'agit donc d'une liste (immuable) d'éléments et d'arbres. La fonction `tree_iterator` s'écrit alors ainsi :

```
let iterator (t : tree) : int iterator =
  let stack = ref (STree (t, SEmpty)) in
  let rec next() =
    match !stack with
    | SEmpty ->
        None
    | SElem (x, rest) ->
        stack := rest;
        Some x
    | STree (Leaf, rest) ->
        stack := rest;
        next()
    | STree (Node (t1, x, t2), rest) ->
        stack := STree (t1, SElem (x, STree (t2, rest)));
        next()
  in
```

```
next
```

Lors de la création de l'itérateur, on alloue et on initialise l'état interne modifiable de l'itérateur. Il s'agit ici d'une référence `stack` qui contient la liste des éléments et des sous-arbres qui restent à énumérer. On définit alors une fonction récursive (terminale) `next`, qui a accès à la pile contenue dans la référence `stack`, et qui peut donc consulter et modifier la pile. La fonction `next` s'écrit en quatre cas. Si la pile est vide, on renvoie `None` : l'itération est terminée. S'il y a un élément `x` en tête de pile, c'est lui le prochain élément : on le dépile et on renvoie `Some x`. S'il y a un arbre trivial `Leaf` en tête de pile, cet arbre ne contient aucun élément : on le dépile et on continue aussitôt la recherche du prochain élément. Enfin, s'il y a un arbre non trivial `Node (t1, x, t2)` en tête de pile, on le remplace en tête de pile par trois éléments distincts, à savoir l'arbre `t1`, l'élément `x`, et l'arbre `t2`, dans cet ordre, car c'est celui exigé par l'énoncé ; et on continue aussitôt. ◇

Solution de l'exercice 4.15 Outre l'arbre `t` fourni par l'utilisateur, l'itérateur alloue dans le tas une pile, c'est-à-dire une liste simplement chaînée, dont l'adresse est stockée dans la référence `stack`. Quelle est la taille maximale atteinte par la pile lorsqu'on parcourt un arbre `t` ? Notons-la *max-stack*(`t`). Un moment de réflexion montre que cette fonction satisfait les équations suivantes :

$$\begin{aligned} \textit{max-stack}(\texttt{Leaf}) &= 1 \\ \textit{max-stack}(\texttt{Node(t1, x, t2)}) &= \max(\textit{max-stack}(\texttt{t1}) + 2, \textit{max-stack}(\texttt{t2})) \end{aligned}$$ ◇

En effet, le parcours d'un arbre `Node(t1, x, t2)` se fait en deux temps. Dans un premier temps, on parcourt le sous-arbre `t1`, et pendant ce temps les deux éléments `x` et `t2` sont inertes sur la pile. Ensuite, une fois le parcours de `t1` terminé et une fois `x` dépilé, on parcourt `t2`.

On voit que ces équations définissent une fonction *max-stack* qui (à une constante près) mesure la **hauteur à gauche** de l'arbre, c'est-à-dire la longueur du plus long chemin qui d'un père va toujours vers le fils gauche.

La complexité en espace de l'itérateur, c'est-à-dire l'espace occupé par la pile, est donc (à une constante près) la hauteur à gauche de l'arbre. Si l'arbre est équilibré, elle est logarithmique vis-à-vis du nombre d'éléments de l'arbre.

Pour déterminer la complexité en temps de la fonction `next`, il faut étudier les deux appels récursifs de `next` et se demander combien de fois `next` peut s'appeler elle-même avant de terminer en produisant `None` ou `Some x`.

Le premier appel de `next` à elle-même ne pose en fait aucun problème. Il est facile de démontrer que la pile contient toujours un élément `SElem (x, _)` soit en tête de pile, soit immédiatement après, c'est-à-dire en deuxième position. Par conséquent, lorsque `next` trouve une pile de la forme `STree (Leaf, rest)`, le reste de la pile `rest` est nécessairement de la forme `SElem (x, _)`. Cela signifie que l'appel récursif à `next` dans ce cas va terminer immédiatement.

Le second appel de `next` à elle-même s'accompagne d'une croissance de la pile. Or, nous venons de démontrer que la taille de la pile est bornée (à une constante près) par la hauteur à gauche de l'arbre. Par conséquent, la même borne s'applique à la complexité en temps de la fonction `next`.

Le coût d'un appel à `next` n'est donc pas $O(1)$. Cela mérite d'être souligné.

Néanmoins, lorsque l'utilisateur effectue une série d'appels successifs à `next` pour énumérer tous les éléments de l'arbre, le coût cumulé des appels à `next` est bien $O(n)$, où n est le nombre d'éléments de l'arbre. En effet, en tout chaque sous-arbre est empilé exactement une fois sous forme d'un `STree`, et chaque élément est empilé exactement une fois sous forme d'un `SElem`. Or le coût total des appels à `next` est (à une constante près) le nombre total d'éléments `STree` et `SElem` qui sont empilés.

Solution de l'exercice 4.16 Nous avons effectué la comparaison lexicographique de deux listes d'éléments lors de l'Exercice 3.16. Ici, on ne compare pas deux listes, mais deux itérateurs, qui produisent les éléments à la demande. Toutefois, la structure générale du code est inchangée.

```
let lexico (compare : 'a comparator) : 'a iterator comparator =
  fun it1 it2 ->
    let rec loop () =
      match it1(), it2() with
      | None, None ->
          0
      |  _, None ->
          1
      | None,  _ ->
          -1
      | Some x, Some y ->
          let cmp = compare x y in
          if cmp <> 0 then cmp
          else loop()
    in
    loop()
```

Soulignons le fait la fonction `loop` est récursive terminale (Remarque 1.20), ce qui implique que la comparaison se fait en espace constant (sans compter l'espace requis par les itérateurs eux-mêmes). De plus, la comparaison termine dès que l'on trouve deux éléments distincts, ou dès que l'on constate que l'une des séquences est plus longue que l'autre. Il n'est pas donc nécessaire en général d'énumérer tous les éléments de `it1` et `it2` pour obtenir le résultat de la comparaison. ◇

Solution de l'exercice 4.17 Il suffit bien sûr de construire un itérateur pour chacun des deux arbres et de comparer ensuite les séquences d'éléments produites par ces deux itérateurs.

```
let same_fringe t1 t2 =
  lexico Pervasives.compare (iterator t1) (iterator t2) = 0
```

On a employé ici la fonction `Pervasives.compare`, issue de la bibliothèque d'OCaml, pour comparer deux entiers : elle admet en particulier le type `int comparator`.

Ce problème est connu dans la littérature sous le nom de « *same-fringe problem* », car il s'agit de comparer les franges de deux arbres. Il illustre l'intérêt des itérateurs (modifiables ou immuables). En effet, une fonction « *fold* » seule ne permet pas de résoudre élégamment ce problème. ◇

Solution de l'exercice 4.18 La solution est donnée par la Figure 4.30. ◇

Solution de l'exercice 4.19 La solution est donnée par la Figure 4.31. Il faut définir d'abord la classe `Pair`, qui malheureusement n'existe pas dans la bibliothèque de Java. Ensuite, on définit un **itérateur produit**, dont les champs `iteratorA` et `iteratorB` contiennent des pointeurs vers les deux itérateurs sous-jacents, et dont les méthodes `next` et `hasNext` font appel aux méthodes `next` et `hasNext` des itérateurs sous-jacents. L'itérateur produit s'arrête dès que l'un des deux itérateurs sous-jacents n'est plus capable de produire d'éléments. ◇

Solution de l'exercice 4.20 Le nouvel itérateur construit par la fonction `map` fonctionne de façon très simple. Lorsqu'on lui demande un nouvel élément, il appelle l'itérateur sous-jacent, puis applique la fonction `f` à l'élément obtenu, dont le type est `'a`, pour obtenir un élément de type `'b`.

```
let map (f : 'a -> 'b) (it : 'a iterator) : 'b iterator =
  fun () ->
    match it() with
    | None ->
        None
    | Some a ->
        Some (f a)
```

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class ArrayIterator<E> implements Iterator<E> {

  private final E[] array;
  private int next;

  public ArrayIterator (E[] array) {
    this.array = array;
    this.next = 0;
  }

  public E next () {
    if (next < array.length)
      return array[next++];
    else
      throw new NoSuchElementException ();
  }

  public boolean hasNext () {
    return next < array.length;
  }

}
```

FIGURE 4.30 – Un itérateur sur un tableau

Il est intéressant de noter que, contrairement à la fonction `List.map` dont la définition est récursive, la fonction `map` des itérateurs n'est pas récursive. Cette différence vient du fait que le type des listes est récursif (Exercice 1.4), tandis que le type des itérateurs, `unit -> 'a option`, ne l'est pas.

Une fois définie cette fonction de transformation, il est facile de construire un itérateur sur un tableau à partir d'un itérateur sur les intervalles :

```
let array_iterator (a : 'a array) : (int * 'a) iterator =
  map (fun i -> i, a.(i)) (interval 0 (Array.length a))
```

Plus généralement, la plupart des fonctions courantes de la bibliothèque `List`, par exemple `filter`, `combine`, etc. ont un analogue sur les itérateurs. Nous avons rencontré lors de l'Exercice 4.12 l'analogue de la fonction `List.fold_left`. ◇

Solution de l'exercice 4.21 La principale difficulté est que pour implémenter `hasNext`, il faut appeler la méthode `get` du fournisseur sous-jacent, afin d'observer si elle lance ou non une exception. Cet appel modifie l'état du fournisseur : le prochain appel à `get` renverra non pas le même élément, mais l'élément suivant. Or, la méthode `hasNext` n'est pas censée modifier l'état de l'itérateur. Cela nous oblige à mémoriser l'élément produit par `get`, de façon à pouvoir renvoyer cet élément (sans appeler à nouveau `get`) lors du prochain appel à `next`.

Nous sommes donc amenés à introduire un décalage temporel, de 0 ou 1 éléments, entre l'itérateur que nous construisons et le fournisseur sous-jacent. Si l'utilisateur de notre itérateur effectue des appels à `next` et `hasNext` qui (dans son esprit) exigent la production de n éléments, alors peut-être serons-nous amenés à exiger la production de $n+1$ éléments de la part du fournisseur sous-jacent. Cela n'est pas entièrement satisfaisant, mais est inévitable, étant données les deux interfaces avec lesquelles nous travaillons ici. Il faut le documenter, voilà tout.

Le code est donné par la Figure 4.32. Un champ Booléen `ahead` nous permet de mémoriser si nous avons exigé ou non un élément à l'avance de la part du fournisseur. La méthode `next`

```
public class Pair<A, B> {
  public final A a;
  public final B b;
  public Pair (A a, B b) { this.a = a; this.b = b; }
}

import java.util.Iterator;

public class IteratorProduct<A, B> implements Iterator<Pair<A, B>> {

  private final Iterator<A> iteratorA;
  private final Iterator<B> iteratorB;

  public IteratorProduct (Iterator<A> iteratorA,
                          Iterator<B> iteratorB) {
    this.iteratorA = iteratorA;
    this.iteratorB = iteratorB;
  }

  public Pair<A, B> next () {
    return new Pair<A, B> (iteratorA.next(), iteratorB.next());
  }

  public boolean hasNext () {
    return iteratorA.hasNext() && iteratorB.hasNext();
  }

}
```

FIGURE 4.31 – Le produit de deux itérateurs

```
import java.util.function.Supplier;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class SupplierIterator<T> implements Iterator<T> {

  // The underlying supplier.
  private final Supplier<T> supplier;
  // A Boolean flag that tells whether we have read
  // one element ahead of time.
  private boolean ahead;
  // The element that we have read ahead of time, if any.
  private T element;

  public SupplierIterator (Supplier<T> supplier) {
    this.supplier = supplier;
    this.ahead = false;
  }

  public T next () {
    if (ahead) {
      ahead = false;
      return element;
    }
    else
      return supplier.get();
  }

  public boolean hasNext () {
    if (ahead)
      return true;
    try {
      element = supplier.get();
      ahead = true;
      return true;
    } catch (NoSuchElementException e) {
      return false;
    }
  }

}
```

FIGURE 4.32 – Un adaptateur entre `Supplier` et `Iterator`

consulte ce champ pour savoir s'il faut renvoyer cet élément, que nous avons stocké dans le champ `element`, ou bien appeler la méthode `get` du fournisseur. La méthode `hasNext` consulte ce champ. S'il contient **`true`**, alors oui, il existe un prochain élément ; nous l'avons déjà. S'il contient **`false`**, alors il faut appeler `get`, en prenant soin de ne pas laisser s'échapper une éventuelle exception `NoSuchElementException`. Si cette exception est observée, c'est qu'il ne reste plus d'éléments, et il suffit de renvoyer **`false`**. Si elle n'est pas observée, alors il faut renvoyer **`true`**, et mettre à jour les champs `ahead` et `element` pour mémoriser le fait que nous avons lu un élément à l'avance. ◇

Solution de l'exercice 4.22 Lorsque la méthode `next` de notre itérateur est appelée, nous devons faire en sorte de produire le plus petit élément parmi les k éléments qui apparaissent en tête des k itérateurs sous-jacents, ou « sources ». Afin d'effectuer efficacement cette opération, nous utilisons une file de priorités. Nous la peuplons initialement de k éléments produits chacun par l'une des k sources. Lorsque notre méthode `next` est appelée, nous extrayons de la file de priorités l'élément souhaité. Il faut alors le remplacer par le prochain élément issu de la même source.

Le code est donné par la Figure 4.33. (La classe auxiliaire `Pair` apparaît dans la Figure 4.31). Notre constructeur crée une file de priorités, et spécifie que l'ordre sur les paires élément-indice est l'ordre donné par notre argument `comparator` sur leurs premières composantes. (Nous utilisons ici la syntaxe des **clôtures** de Java 8 pour construire de façon concise un objet de type `Comparator<Pair<E,Integer>>`.) La file est initialisée en y insérant un élément en provenance de chaque source (si possible). Notre méthode `next` extrait un élément `element` de la file de priorités, qui est celui que nous devons renvoyer. L'indice `i` obtenu en même temps identifie la source d'où provenait cet élément. Nous tirons alors un nouvel élément de cette source et l'ajoutons à la file de priorités.

La complexité en temps de notre méthode `next` est $O(1)$ plus le coût des appels `poll` et `add` à la file de priorités. Une implémentation typique des files de priorités, basée sur un tas équilibré, permet d'effectuer `poll` en temps $O(1)$ et `add` en temps $O(k)$.

La complexité en temps de `hasNext` est celle de la méthode `size` des files de priorité, soit typiquement $O(1)$.

L'espace nécessaire à notre itérateur est $O(1)$ plus l'espace occupé par la file de priorités, soit $O(k)$.

Dans une application typique où k est faible et où le nombre d'éléments produits par chaque itérateur est très grand, cet algorithme est intéressant. Il permet par exemple d'effectuer la fusion ordonnée de très gros fichiers, stockés sur le disque car trop gros pour pouvoir être chargés en mémoire vive. ◇

Solution de l'exercice 4.23 On définit une fonction récursive `alternator` paramétrée par un entier `sign`, qui vaut 1 ou -1, et qui représente le premier élément de la séquence :

```
let rec alternator (sign : int) : int immutable_iterator =
  fun () ->
    Cons (sign, alternator (-sign))
```

L'expression `alternator 1` fournit alors l'itérateur immuable demandé. ◇

Solution de l'exercice 4.24 Le code est simple. Il s'écrit presque comme si la fonction `interval` construisait une liste. La seule différence est qu'il faut écrire **`fun () ->`** `...` pour retarder le calcul et obtenir une valeur de type `int immutable_iterator`.

```
let rec interval i j : int immutable_iterator =
  fun () ->
    if i < j then
      Cons (i, interval (i + 1) j)
    else
      Nil
```

```
import java.util.*;

public class MultiwayMergeIterator<E> implements Iterator<E> {

  // The iterators that we are controlling.
  private Iterator<E>[] input;
  // This priority queue contains (at most) one element from each
  // input iterator. If it contains no element from iterator i,
  // then this iterator has no more elements. The queue actually
  // contains not just elements, but pairs of an element and an
  // index i that tells us which iterator produced this element.
  private PriorityQueue<Pair<E,Integer>> queue;

  public MultiwayMergeIterator (Comparator<E> comparator,
                                Iterator<E>[] input) {
    this.input = input;
    // Create the priority queue.
    this.queue = new PriorityQueue<Pair<E,Integer>> (
      input.length,
      (p1, p2) -> comparator.compare(p1.a, p2.a)
    );
    // Populate the queue with one element from each input stream (if
    // there is one; otherwise, not a problem).
    for (int i = 0; i < input.length; i++)
      drawFrom(i);
  }

  private void drawFrom (int i) {
    // Draw an element from iterator i and add it to the queue.
    if (input[i].hasNext())
      queue.add(new Pair<E,Integer> (input[i].next(), i));
  }

  public E next () {
    // Take the pair that contains the least element of the queue.
    Pair<E,Integer> pair = queue.poll();
    if (pair == null)
      throw new NoSuchElementException ();
    // Decompose this pair.
    E element = pair.a;
    int i = pair.b;
    // Replace this element in the priority queue with a new element
    // drawn from the same input iterator (if there is one).
    drawFrom(i);
    // Return the element that we took out of the queue.
    return element;
  }

  public boolean hasNext () {
    return queue.size() > 0;
  }

}
```

FIGURE 4.33 – Fusion ordonnée de k itérateurs

Lorsqu'on appelle `interval 0 n`, la seule opération effectuée par la fonction `interval` est l'allocation en mémoire d'une **clôture** qui représente la fonction `fun () -> ...`. Cette fonction a deux variables libres `i` et `j`, donc cette clôture aura deux champs pour stocker les valeurs de `i` et `j`, qui ici sont `0` et `n`. La création d'une clôture se fait en temps $O(1)$, donc la complexité en temps de l'appel `interval 0 n` est $O(1)$.

Une clôture occupe un espace $O(1)$, donc l'itérateur `it` occupe un espace $O(1)$, et ce, quelle que soit la valeur de l'entier `n`.

En résumé, la séquence des éléments de l'intervalle n'est pas construite à l'avance. Les éléments sont calculés à la demande : l'addition `i + 1` n'est effectuée qu'au moment où l'entier `i` est exigé. (On peut trouver étrange que l'addition `i + 1` soit effectuée « un cran trop tôt ». Ici, ce n'est pas grave, car une addition ne coûte rien. Le lecteur exigeant pourra modifier le code de façon à ce que l'addition `i + 1` soit effectuée au moment où l'entier `i + 1` est exigé.) ◇

Solution de l'exercice 4.25 Cet exercice est analogue à l'Exercice 4.12, mais concerne cette fois les itérateurs immuables. À nouveau, le code s'écrit facilement sous forme récursive terminale :

```
let rec on_immutable_iterator xs f accu =
  match xs() with
  | Nil ->
      accu
  | Cons (x, xs) ->
      let accu = f x accu in
      let accu = on_immutable_iterator xs f accu in
      accu
```

En fait, le code est essentiellement identique à celui de la fonction `fold_left` sur les listes (Exercice 4.6). La seule différence est qu'il faut écrire `xs()` au lieu de `xs` afin d'exiger le calcul de l'élément suivant.

On a vu lors de l'exercice précédent que l'itérateur `it` construit par l'appel `interval 0 n` occupe en mémoire un espace $O(1)$. Quant à l'entier `sum`, il occupe naturellement un espace $O(1)$ également. Les Exercices 4.26 et 4.27 montreront que cette analyse devient plus complexe dans le cas où l'on considère des flots bâtis à l'aide de suspensions. ◇

Solution de l'exercice 4.26 La définition de la fonction `interval` est presque identique à celle donnée dans la solution de l'Exercice 4.24. La seule différence est qu'il faut remplacer `fun () -> ...` par `lazy (...)` afin de construire une suspension au lieu d'une fonction.

```
let rec interval i j : int stream =
  lazy (
    if i < j then
      Cons (i, interval (i + 1) j)
    else
      Nil
  )
```

De même, la définition de la fonction `on_stream` est presque identique à celle donnée dans la solution de l'Exercice 4.25. La seule différence est qu'il faut remplacer `xs()` par `Lazy.force xs` afin d'évaluer une suspension au lieu d'appeler une fonction.

```
let rec on_stream xs f accu =
  match Lazy.force xs with
  | Nil ->
      accu
  | Cons (x, xs) ->
      let accu = f x accu in
      on_stream xs f accu
```

On pose `let s = interval 0 n`. En consultant la définition de la fonction `interval`, on constate que cet appel de fonction termine immédiatement : il construit une suspension et la

renvoie. Il termine donc en temps $O(1)$. La suspension ainsi construite doit stocker les valeurs des variables `i` et `j` (à savoir ici `0` et `n`). Elle occupe donc un espace $O(1)$.

On pose ensuite `let sum = on_stream s (+) 0`. Cet appel de fonction s'exécute bien sûr en temps $O(n)$, car il exige le calcul successif de chacun des éléments de l'intervalle $[0, n[$, ainsi que le calcul de leur somme cumulée. Une fois ce calcul effectué, **le flot `s` est explicitement représenté en mémoire** sous forme d'une liste chaînée de suspensions déjà évaluées. Il occupe donc un espace $O(n)$.

En effet, `s` est un flot, donc une suspension. Cette suspension a été évaluée par l'appel à `on_stream`. Elle contient donc la valeur mémoisée `Cons (0, s')`, où `s'` est un flot qui représente l'intervalle $[1, n[$. Mais le même raisonnement s'applique à `s'` : lui aussi est un flot, donc une suspension, et cette suspension a été évaluée par `on_stream`. Elle contient donc la valeur mémoisée `Cons (1, s'')`, etc.

Cet exemple montre que l'espace occupé en mémoire par un flot peut varier au cours du temps. Le flot `s` occupe initialement un espace $O(1)$, parce que ce flot n'a pas été évalué. Mais, une fois ce flot entièrement évalué, il occupe un espace $O(n)$. On pourrait l'évaluer à nouveau et calculer par exemple le produit de ses éléments : `let prod = on_stream s ( * ) 0`. Ce calcul ne provoquerait pas de nouvelle allocation de mémoire ; l'espace occupé par `s` resterait le même.

En résumé, on voit que, même si un flot est **conceptuellement immuable**, il est en réalité construit à l'aide d'objets modifiables, à savoir les suspensions. Il en résulte que l'espace mémoire occupé par ce flot peut varier au cours du temps. ◇

Solution de l'exercice 4.27 La réponse est $O(1)$. Le calcul peut être effectué en espace constant. En simplifiant très légèrement les choses, on peut affirmer que, à chaque étape de l'itération, ne sont stockés en mémoire que l'élément courant et la somme cumulée courante – exactement comme si, au lieu d'utiliser un flot, on avait écrit manuellement une boucle **for**.

Eu égard à la solution de l'exercice précédent, où l'on expliquait que le flot `interval 0 n`, une fois évalué, occupait un espace $O(n)$, cette affirmation peut paraître surprenante.

Expliquons d'abord pourquoi elle est vraie à la fin du calcul.

Il est vrai que le flot `interval 0 n`, une fois entièrement évalué, est représenté en mémoire sous forme d'une liste chaînée de suspensions, donc occupe un espace $O(n)$. Cependant, une fois l'itération terminée, c'est-à-dire après l'appel `on_stream (interval 0 n) (+) 0`, ce flot est devenu inutile. Plus précisément, il est devenu **inaccessible** : il n'existe aucun pointeur vers lui. La mémoire qu'il occupe peut donc être libérée par le **garbage collector**. Par conséquent, si l'on ne compte que les blocs de mémoire accessibles, on peut affirmer que ce flot n'occupe plus d'espace en mémoire.

Voyons à présent pourquoi notre affirmation est vraie également pendant le calcul.

Plaçons-nous à un instant arbitraire où l'élément i a déjà été exigé mais les éléments suivants n'ont pas encore été exigés par la fonction `on_stream`. Les éléments de l'intervalle $[0, i[$ ont tous été produits. Ce préfixe du flot forme donc en mémoire une liste chaînée de suspensions déjà évaluées. Mais, encore une fois, ce préfixe est inaccessible : il n'existe plus aucun pointeur vers ces suspensions. Ce préfixe n'occupe donc pas (plus) d'espace en mémoire. L'élément `i` et la somme cumulée, représentée par la variable locale `accu` de la fonction `on_stream`, occupent un espace $O(1)$. Enfin, les éléments du flot au-delà de i n'ont pas encore été exigés ; ils sont représentés par une suspension non encore évaluée, qui occupe un espace $O(1)$.

En résumé, grâce au « *garbage collector* », qui libère lorsque cela est nécessaire les blocs de mémoire inutilisés, ce calcul peut être effectué en espace $O(1)$.

Il reste vrai que ce calcul alloue au total $O(n)$ blocs de mémoire. Mais parmi ces blocs, seuls $O(1)$ blocs sont accessibles à un instant donné. ◇

Solution de l'exercice 4.28 La structure générale de la fonction `append` est la même que celle d'une fonction de concaténation de deux listes, comme `List.append`. La seule différence est qu'il faut y ajouter des instructions de création de suspension `lazy (...)` et d'évaluation de

suspension `Lazy.force (...)`. Pour cela, il faut se laisser guider d'une part par la structure des types, d'autre part par la question : ce calcul doit-il être effectué maintenant ou bien suspendu ?

Voici une solution correcte :

```
let rec append (xs : 'a stream) (ys : 'a stream) : 'a stream =
  lazy (
    match Lazy.force xs with
    | Nil ->
        Lazy.force ys
    | Cons (x, xs) ->
        Cons (x, append xs ys)
  )
```

Parce que l'appel `append xs ys` doit terminer en temps $O(1)$, donc sans exiger le premier élément du flot `xs`, on place dès la première ligne une instruction de création de suspension. Le code entre parenthèses dans **`lazy`** `(...)` n'est donc exécuté qu'au moment où le premier élément du flot `append xs ys` est exigé.

Ce code entre parenthèses doit renvoyer une valeur de type `'a head`. Pour construire cette valeur, il faut tester si le flot `xs` est vide ou non. On évalue donc (la tête de) ce flot, à l'aide de l'expression `Lazy.force xs`, qui produit elle-même une valeur de type `'a head`. Une construction **`match`** permet alors de déterminer si cette valeur est étiquetée `Nil` ou `Cons`.

Si c'est `Nil`, alors le flot `xs` est vide. Il faut donc renvoyer la séquence des éléments de `ys`. Toutefois, on ne peut pas écrire simplement `ys`, car cette variable a le type `'a stream`, et on attend ici une expression de type `'a head`. On insère donc un appel à `Lazy.force`. En d'autres termes, si on a exigé le premier élément de `xs` et si on a obtenu `Nil`, alors on exige aussitôt le premier élément de `ys`.

Si c'est `Cons (x, xs)`, alors il faut renvoyer une séquence dont le premier élément est `x` et dont les éléments suivants sont ceux de `xs` suivis de ceux de `ys`. On écrit pour cela l'expression `Cons (x, append xs ys)`. Notons que cette expression alloue deux blocs de mémoire, à savoir un bloc `Cons` et une suspension, et termine en temps $O(1)$.

Soulignons le fait que, si on n'y prend garde, on peut écrire des variantes de la fonction `append` qui sont acceptées par le compilateur OCaml (parce qu'elles ont bien le type souhaité, à savoir `'a stream -> 'a stream -> 'a stream`) mais n'ont pas le comportement souhaité en termes de complexité. Par exemple,

```
let rec wrong_append xs ys =
  match Lazy.force xs with
  | Nil ->
      ys
  | Cons (x, xs) ->
      lazy (Cons (x, wrong_append xs ys))
```

Voyez-vous en quoi ce code est incorrect ? L'appel `wrong_append xs ys` exige immédiatement le premier élément de `xs`, sans attendre que le premier élément du flot `wrong_append xs ys` soit exigé. De ce fait, si l'utilisateur demande les n premiers éléments du flot `wrong_append xs ys`, alors cela provoquera le calcul des $n+1$ premiers éléments du flot `xs`, ce qui n'est pas nécessaire ; c'est un de trop.

En voici un autre exemple :

```
let rec very_wrong_append xs ys =
  match Lazy.force xs with
  | Nil ->
      ys
  | Cons (x, xs) ->
      let tail = very_wrong_append xs ys in
      lazy (Cons (x, tail))
```

Voyez-vous en quoi ce code est encore pire que le précédent ? La seule différence vis-à-vis de `wrong_append` tient au fait que l'appel récursif a lieu avant la construction de la suspension, et non pas au moment où la suspension est évaluée. De ce fait, l'appel `very_wrong_append xs ys` exige immédiatement tous les éléments du flot `xs`, ce qui est totalement anormal.

Ces erreurs peuvent être difficiles à détecter parce que ces variantes erronées de la fonction `append` sont fonctionnellement correctes : elles renvoient bien un flot contenant les éléments du flot `xs` suivis des éléments du flot `ys`. Seul le moment où les calculs sont effectués est incorrect.◇

Solution de l'exercice 4.29 Afin de faciliter l'écriture du code qui suit, on ajoute d'abord à la classe `Stream` un constructeur supplémentaire ainsi qu'une méthode `force`.

```
public class Stream<T> {
  ...
  public Stream (Callable<Head<T>> c) {
    this (new Thunk<Head<T>> (c));
  }
  public Head<T> force () throws Exception {
    return this.thunk.call();
  }
  ...
}
```

Ce nouveau constructeur permet de construire un flot en fournissant non pas une suspension de type `Thunk<Head<T>>` mais une clôture de type `Callable<Head<T>>`. Il construit une suspension puis fait appel au constructeur existant. La méthode `force` exige l'évaluation de la suspension stockée dans le champ `thunk`. Ces deux ajouts ne sont pas essentiels. Ils permettent simplement de gagner un peu de concision dans ce qui suit.

Comme le suggère l'énoncé, on déclare dans la classe `Head` une méthode abstraite :

```
public abstract Head<T> append (Stream<T> that) throws Exception;
```

Il reste alors à implémenter les méthodes `append` des classes `Stream`, `Nil` et `Cons`. Comme dans l'exercice précédent, la première doit terminer en temps $O(1)$, donc construit immédiatement une suspension :

```
public class Stream<T> {
  ...
  public Stream<T> append (Stream<T> that) {
    return new Stream<T> (() -> this.force().append(that));
  }
  ...
}
```

Le jour où cette suspension est évaluée, l'appel `this.force()` a lieu, qui exige le premier élément du flot `this`. Cet appel produit un objet de type `Head<T>`. On appelle alors la méthode `append` de cet objet, ce qui permet d'exécuter un code différent selon que cet objet est étiqueté `Nil` ou `Cons`.

La méthode `append` de la classe `Nil` correspond au premier cas. Comme dans l'exercice précédent, le flot `this` étant vide, il faut alors renvoyer la séquence des éléments du flot `that` :

```
public class Nil<T> extends Head<T> {
  public Head<T> append (Stream<T> that) throws Exception {
    return that.force();
  }
}
```

La méthode `append` de la classe `Cons` correspond au second cas. Le flot `this` étant non vide, il faut renvoyer une séquence constituée d'abord du premier élément du flot `this`, nommé ici `this.elem`; puis des éléments suivants du flot `this`, nommés ici `this.rest`; puis des éléments du flot `that`. Comme dans l'exercice précédent, on produit immédiatement un objet

étiqueté `Cons` qui contient d'un part l'élément `this.elem`, d'autre part la concaténation des flots `this.rest` et `that`, obtenue grâce à un appel récursif.

```
public class Cons<T> extends Head<T> {
  ...
  public Head<T> append (Stream<T> that) {
    return new Cons<T> (elem, rest.append(that));
  }
}
```

En résumé, la structure du code est la même que dans l'exercice précédent. Cependant, le fait de devoir écrire ce code dans trois classes différentes le rend probablement plus difficile à comprendre. De plus, Java est moins concis qu'OCaml. Le code obtenu a le comportement souhaité, mais n'est malheureusement pas très clair. ◇

Solution de l'exercice 4.30 La fonction `interleave` s'écrit facilement sous forme récursive et par analyse de cas sur le premier flot :

```
let rec interleave xs ys =
  lazy (
    match Lazy.force xs with
    | Nil ->
        Lazy.force ys (* point 1 *)
    | Cons (x, xs) ->
        Cons (x, interleave ys xs) (* point 2 *)
  )
```

Deux points méritent d'être soulignés. D'abord, dans le cas où le premier flot est vide, on continue avec le second flot. Ainsi, si l'un des deux flots est plus court que l'autre, on cesse d'intercaler des éléments en provenance des deux flots ; on continue simplement avec ce qui reste du flot le plus long. Ensuite, dans le cas où le premier flot est non vide, on produit son premier élément `x`, puis on continue avec un appel récursif `interleave ys xs` où le rôle des deux flots a été échangé. Cette petite astuce permet d'extraire facilement un élément à tour de rôle de chaque flot, sans qu'il soit nécessaire de dupliquer le code. ◇

Bibliographie

Sylvain Conchon et Jean-Christophe Filliâtre. *Apprendre à programmer avec OCaml: Algorithmes et structures de données*. Eyrolles, 2014.

Barbara Liskov et John V. Guttag. *Program Development in Java – Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.

Jean-Christophe Filliâtre. Les bases de la programmation et de l'algorithmique. INF411, École Polytechnique, 2014.

Jonathan Aldrich. The power of interoperability: why objects are inevitable. *In ACM Symposium on New Ideas in Programming and Reflections on Software (Onward !)*, pages 101–116, 2013.

Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, 1965.

Maurice Naftalin et Philip Wadler. *Java generics and collections*. O'Reilly, 2006.

John C. Reynolds. Types, abstraction and parametric polymorphism. *In Information Processing 83*, pages 513–523. Elsevier Science, 1983.

D. J. Wheeler. The use of sub-routines in programmes. *In Proceedings of the 1952 ACM National Meeting*, pages 235–236, 1952.

Index