Scala par l’exemple

Martin Odersky

Scala par l'exemple

Martin Odersky

Table des matières

Notes du traducteur

Cette traduction est un travail personnel, une contribution pour aider à mieux faire connaitre ce langage.

Si vous trouvez des erreurs (fautes d'orthographe ou de grammaire, termes improprement traduits), merci de m'en faire part à l'adresse *eric.jacoboni at gmail*. J'intégrerai vos remarques dans la prochaine version.

Chapitre 1. Introduction

Scala rassemble dans un même langage les paradigmes de programmation orientée objet et de programmation fonctionnelle. Il a été conçu pour exprimer les motifs de programmation courants de façon concise, élégante, avec un système de typage sûr. Plusieurs nouvelles constructions innovantes sont désormais à la disposition du programmeur :

- Les types abstraits et la composition par mixin unifient les concepts d'objet et de module.
- Le pattern matching peut s'appliquer à des hiérarchies de classes et unifie ainsi l'accès aux données, qu'il s'effectue de façon fonctionnelle ou orientée objet. Il simplifie aussi énormément le traitement des arborescences XML.
- Une syntaxe et un système de typage souples permettent de créer des bibliothèques élaborées et des langages spécifiques au domaine (DSL).

En même temps, Scala est compatible avec Java. Vous pouvez donc utiliser les bibliothèques et les frameworks Java sans code supplémentaire.

Ce document présente Scala de façon informelle, au moyen d'une suite d'exemples.

Les chapitres 2 et 3 présentent quelques-unes des fonctionnalités qui font l'intérêt de Scala. Les chapitres suivants introduisent les constructions du langage de façon plus détaillée – les expressions et les fonctions simples, les objets et les classes, les listes et les flux, la notion d'état mutable et le pattern matching – et montrent quelques techniques de programmation utiles. Cette présentation informelle est prévue pour être le complément du manuel de référence du langage, qui décrit Scala de façon plus détaillée et plus précise.

Remerciements. Nous devons beaucoup au merveilleux ouvrage de Abelson et Sussman, "Structure et Interprétation des programmes informatiques"[ASS96] et nous reprendrons ici un bon nombre de leurs exemples et exercices en remplaçant, bien sûr, Scheme par Scala. En outre, nos exemples utilisent les constructions orientées objet de Scala lorsque cela est approprié.

Chapitre 2. Premier exemple

Notre premier exemple est une implémentation du tri rapide en Scala :

```
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

Cette implémentation ressemble beaucoup à ce que l'on aurait pu écrire en Java ou en C : elle utilise les mêmes opérateurs et les mêmes structures de contrôle que dans ces deux langages. Mais nous pouvons quand même noter quelques différences de syntaxe :

- Les définitions commencent par un mot réservé. Les définitions de fonctions sont introduites par **def**, les définitions de variables par **var** et les définitions de valeurs (des variables en lecture seule) par **val**.
- Le type d'un symbole est placé après le symbole et le signe deux-points. Ce type peut souvent être omis car le compilateur peut le déduire du contexte.
- Les types tableaux s'écrivent `Array[T]` au lieu de `T[]` et l'accès à un élément se note `a(i)` et non `a[i]`.
- Les fonctions peuvent être imbriquées. Une fonction "interne" peut accéder aux paramètres et aux variables locales de sa fonction englobante. Le nom de tableau `xs`, par exemple, est visible dans les fonction `swap` et `sort1` et n'a donc pas besoin de leur être passé en paramètre.

Pour l'instant, Scala ressemble à un langage conventionnel avec quelques spécificités syntaxiques mais, en réalité, vous pouvez écrire vos programmes dans un style impératif classique ou selon le paradigme orienté objet. C'est un point important car c'est l'une des raisons pour lesquelles on peut aisément combiner des composants Scala avec des composants écrits dans en Java, C# ou Visual Basic.

Vous pouvez également utiliser un style de programmation totalement différent. Voici à nouveau la fonction de tri rapide, cette fois-ci écrite de façon fonctionnelle :

```
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(sort(xs filter (pivot >)),
                 xs filter (pivot ==),
                 sort(xs filter (pivot <)))
```

Ce programme fonctionnel capture de façon concise l'essence même de l'algorithme du tri rapide :

- Si le tableau est vide ou n'a qu'un seul élément, c'est qu'il est déjà trié auquel cas on le renvoie immédiatement.
- Si le tableau n'est pas vide, on choisit un élément situé au milieu, qui servira de "pivot".
- On divise le tableau en trois tableaux contenant, respectivement, les éléments plus petits que le pivot, les éléments plus grands que le pivot et les éléments égaux au pivot (ce qui n'est pas exactement ce que fait la version impérative puisque cette dernière partitionne le tableau en deux : les éléments plus petits que le pivot et les éléments supérieurs ou égaux au pivot).
- On trie les deux premiers sous-tableaux en appelant récursivement la fonction de tri.
- Le résultat est obtenu en concaténant ces trois sous-tableaux.

Les implémentations impérative et fonctionnelle ont la même complexité asymptotique – O(N *log*(N)) dans le cas général et O(N^2) dans le pire des cas. Cependant, alors que la première agit sur place en modifiant son paramètre tableau, la seconde renvoie un nouveau tableau trié sans modifier son paramètre – elle nécessite donc plus de mémoire temporaire que la version impérative.

L'implémentation fonctionnelle semble indiquer que Scala est un langage spécialisé dans le traitement fonctionnel des tableaux. Ce n'est pas le cas : toutes les opérations utilisées dans cet exemple sont, en réalité, de simples appels de méthodes de la classe `Seq[T]`, qui décrit les *séquences* et qui fait partie de la bibliothèque standard de Scala, elle-même écrite en Scala. Les tableaux étant des instances de `Seq`, toutes les méthodes sur les séquences peuvent donc leur être appliquées.

La méthode `filter`, par exemple, prend en paramètre une *fonction prédicat* qui associe des valeurs booléennes à chaque élément du tableau. Le résultat est un tableau contenant tous les éléments du tableau initial qui vérifient ce prédicat. La méthode `filter` d'un objet de type `Array[T]` a donc la signature suivante :

```
def filter(p: T => Boolean): Array[T]
```

Ici, `T => Boolean` est le type des fonctions prenant un élément de type `T` et renvoyant un `Boolean`. Les fonctions comme `filter`, qui prennent une autre fonction en paramètre ou qui renvoient une fonction comme résultat, sont appelées fonctions de *premier ordre*.

Scala ne fait pas de différence entre les noms de méthodes et les noms d'opérateurs. Un nom de méthode peut être une suite de lettres et de chiffres commençant par une lettre ou un suite de caractères spéciaux comme "+", "*" ou ":". En Scala, tout nom de méthode peut être utilisé comme opérateur infixe : l'opération binaire `E op E'` sera toujours interprétée comme l'appel de méthode `E.op(E')`. Ceci est également vrai pour les opérateurs binaires infixes dont le nom commence par une lettre : l'expression `xs filter (pivot >)` est équivalente à l'appel de méthode `xs.filter(pivot >)`.

Dans l'exemple du tri rapide, `filter` est appliquée trois fois à un paramètre qui est une fonction anonyme. Lors du premier appel, `pivot >` représente la fonction qui prend un paramètre `x` et renvoie la valeur `pivot > x`. C'est un exemple d'*application partielle d'une fonction*. Une autre façon d'écrire ce paramètre aurait été de rendre `x` explicite, en utilisant la notation `x => pivot > x`. Cette fonction est anonyme car elle n'a pas de nom qui lui est associé. Ici, le type du paramètre `x` a été omis car le compilateur Scala est capable de l'inférer automatiquement à partir du contexte d'utilisation de la fonction. Pour résumer, `xs.filter(pivot >)` renvoie la liste de tous les éléments de la liste xs inférieurs à pivot.

Si vous examinez à nouveau en détails la première implémentation impérative du tri rapide, vous pourrez constater qu'elle utilise également la plupart des constructions de la seconde solution, mais sous forme déguisée.

Les opérateurs "standard" comme `+`, `-` ou `<`, par exemple, ne sont pas traités de façon spéciale : comme `append`, ce sont des méthodes de leur opérande gauche. Par conséquent, l'expression `i + 1` est considérée comme l'appel `i.+(1)` de la méthode `+` de la valeur entière `i`. Un compilateur peut évidemment (s'il est suffisamment astucieux) reconnaître qu'il s'agit d'un cas particulier d'appel de la méthode `+` sur des opérandes entiers et produire un code optimisé pour cette opération.

Pour des raisons d'efficacité et pour améliorer les diagnostics d'erreurs, la boucle `while` est une construction primitive mais, en principe, elle aurait très bien pu être écrite comme une fonction prédéfinie. En voici une possible implémentation :

```
def While (p: => Boolean) (s: => Unit) {
  if (p) { s ; While(p)(s) }
}
```

La fonction `While` prend une fonction de test comme premier paramètre. Celle-ci n'attend pas de paramètre et produit une valeur booléenne. Le second paramètre de `While` est une fonction commande qui ne prend pas non plus de paramètre et qui produit un résultat de type `Unit`. `While` appelle la fonction commande tant que la fonction de test renvoie `true`.

Le type `Unit` correspond en gros au `void` de Java : il est utilisé à chaque fois qu'une fonction ne renvoie pas de résultat intéressant. En fait, Scala étant un langage orienté expressions, chaque fonction renvoie un résultat : si aucune expression explicite n'est renvoyée, c'est la valeur `()` (qui se prononce "unit" et qui est du type `Unit`) qui est renvoyée par défaut. Les fonctions qui renvoient le type `Unit` sont également appelées *procédures*. Voici, par exemple, une version "orientée expression" de la fonction `swap` de la première version du tri rapide qui renvoie explicitement "unit" :

```
def swap(i: Int, j: Int) {
  val t = xs(i); xs(i) = xs(j); xs(j) = t
  ()
}
```

La valeur renvoyée par cette fonction est simplement celle de sa dernière expression – il n'y a pas besoin du mot-clé **return**. Notez que les fonctions qui renvoient une valeur qui n'est pas de type `Unit` doivent toujours avoir un `=` avant leur corps ou l'expression qui les définit.

Chapitre 3. Programmer avec des acteurs et des messages

Voici un exemple montrant un domaine d'application pour lequel Scala est particulièrement bien adapté. Supposons que l'on veuille implémenter un service d'enchères électroniques. Nous utiliserons le modèle de traitement par acteurs d'Erlang pour implémenter les participants à l'enchère. Les acteurs sont des objets auxquels on envoie des messages. Chaque acteur dispose d'une "boîte aux lettres" dans laquelle arrive ses messages et qui est représentée comme une file d'attente. Il peut traiter séquentiellement les messages de sa boîte ou rechercher les messages qui correspondent à certains critères.

Pour chaque article mis aux enchères, il y a un acteur commissaire-priseur qui publie les informations sur cet article, qui reçoit les offres des clients et qui communique avec le vendeur et celui qui a remporté l'enchère afin de clore la transaction. Nous présenterons ici une implémentation simple de cet acteur.

La première étape consiste à définir les messages échangés au cours d'une enchère. Nous utiliserons deux classes de base abstraites, `AuctionMessage` pour représenter les messages des clients adressés au service des enchères et `AuctionReply` pour représenter les réponses du service aux clients. Pour ces deux classes de bases, il existe un certain nombre de cas, définis dans le Listing 3–1.

Ces *case classes* définissent le format des différents messages de la classe. Ces messages pourraient être traduits en documents XML car nous supposons qu'il existe des outils automatiques permettant d'effectuer des conversions entre documents XML et ces représentations internes

Le Listing 3–2 présente une implémentation Scala d'une classe `Auction` permettant de représenter les commissaires-priseurs qui coordonnent les enchères pour un article particulier. Les objets de cette classe sont crées en indiquant :

- Un acteur vendeur qui doit être prévenu lorsque l'enchère est terminée.
- Une enchère minimale.
- La date de fin de l'enchère.

Le comportement de l'acteur est défini par sa méthode `act` qui passe son temps à sélectionner un message (avec `receiveWithin`) et qui y réagit ; ceci jusqu'à la fin de l'enchère, qui est signalée par le message `TIMEOUT`. Avant de se terminer, l'acteur reste actif pendant une période déterminée par la constante `timeToShutdown` et répond aux autres offres que l'enchère est terminée.

Classes des messages pour un service d'enchères.

```
import scala.actors.Actor

abstract class AuctionMessage
case class Offer(bid: Int, client: Actor)          extends AuctionMessage
case class Inquire(client: Actor)                  extends AuctionMessage

abstract class AuctionReply
case class Status(asked: Int, expire: Date)        extends AuctionReply
case object BestOffer                              extends AuctionReply
case class BeatenOffer(maxBid: Int)                extends AuctionReply
case class AuctionConcluded(seller: Actor, client: Actor)
```

```
                                                    extends AuctionReply

case object AuctionFailed                           extends AuctionReply
case object AuctionOver                             extends AuctionReply
```

Voici quelques informations supplémentaires sur les constructions utilisées dans ce programme :

- La méthode `receiveWithin` de la classe `Actor` prend en paramètre une durée en millisecondes et une fonction qui traite les messages de la boîte aux lettres. Cette fonction est décrite par une suite de cas qui précisent, chacun, un motif et une action à réaliser lorsqu'un message correspond à ce motif. La méthode `receiveWithin` sélectionne le premier message de la boîte aux lettres qui correspond à l'un de ces motifs et lui applique l'action correspondante.

- Le dernier cas de `receiveWithin` utilise le motif `TIMEOUT`. Si aucun autre message n'est reçu dans l'intervalle, ce motif est déclenché après l'expiration du délai passé à la méthode `receiveWithin` englobante. `TIMEOUT` est un message spécial qui est déclenché par l'implémentation de la classe `Actor`.

- Les messages de réponse sont envoyés en utilisant la syntaxe `destination ! UnMessage`. Le symbole `!` est utilisé ici comme un opérateur binaire portant sur un acteur et un message : c'est l'équivalent Scala de l'appel `destination.!(UnMessage)`, c'est-à-dire de l'appel de la méthode `!` de l'acteur *destination* avec le *message* en paramètre.

Cette présentation a donné un avant-goût de la programmation distribuée en Scala. Il pourrait sembler que Scala dispose d'un grand nombre de constructions gérant les processus acteurs, l'envoi et la réception des messages, la gestion des délais d'expiration, etc. mais il n'en est rien : toutes ces constructions sont des méthodes de la classe `Actor`. Cette classe est elle-même implémentée en Scala et repose sur le modèle sous-jacent des threads du langage hôte (Java ou .NET). La section 17.11 présentera l'implémentation de toutes les fonctionnalités de la classe `Actor` utilisées ici.

Se reposer sur une bibliothèque permet d'avoir un langage relativement simple et d'offrir beaucoup de souplesse aux concepteurs des bibliothèques. En effet, le langage n'ayant pas besoin de spécifier les détails de la communication entre les processus, il peut rester plus simple et plus général. En outre, le modèle particulier des messages stockés dans une boîte aux lettres étant un module de la bibliothèque, il peut être librement modifié si d'autres applications ont besoin d'autres modèles. Cette approche nécessite toutefois que le langage soit suffisamment expressif pour fournir les abstractions nécessaires de façon simple. La conception de Scala a été faite dans cet esprit : l'un des buts principaux était que le langage devait être suffisamment souple pour servir de langage hôte pour des langages spécifiques au domaine implémentés au moyen de modules de bibliothèque. Les constructions permettant la communication des acteurs, par exemple, peuvent être considérées comme l'un de ces langages spécifiques, qui étend conceptuellement le langage Scala de base.

Implémentation d'un service d'enchères.

```
class Auction(seller: Actor, minBid: Int, closing: Date) extends Actor {
  val   timeToShutdown  = 36000000      // msec
  val bidIncrement = 10
  def act() {
    var maxBid = minBid - bidIncrement
    var maxBidder: Actor = null
    var running = true
    while (running) {
      receiveWithin ((closing.getTime() - new Date().getTime())) {
        case Offer(bid, client) =>
          if (bid >= maxBid + bidIncrement) {
            if (maxBid >= minBid) maxBidder ! BeatenOffer(bid)
            maxBid = bid; maxBidder = client; client ! BestOffer
          } else {
            client ! BeatenOffer(maxBid)
          }
        case Inquire(client) =>
          client ! Status(maxBid, closing)
        case TIMEOUT =>
```

```
          if (maxBid >= minBid) {
            val reply = AuctionConcluded(seller, maxBidder)
            maxBidder ! reply; seller ! reply
          } else {
            seller ! AuctionFailed
          }
          receiveWithin(timeToShutdown) {
            case Offer(_, client) => client ! AuctionOver
            case TIMEOUT => running = false
          }
        }
      }
    }
  }
}
```

Chapitre 4. Expressions et fonctions simples

Les exemples précédents ont donné un avant-goût ce qu'il était possible de faire avec Scala. Nous allons maintenant introduire ses constructions les unes après les autres de façon plus systématique, en commençant par le niveau le plus bas : les expressions et les fonctions.

Expressions et fonctions simples

Un système Scala est fourni avec un interpréteur qui peut être considéré comme une calculatrice améliorée. L'utilisateur interagit avec cette calculatrice en tapant des expressions et la calculatrice renvoie les résultats des évaluations et leurs types. Voici un exemple :

```
scala> 87 + 145
res0: Int = 232

scala> 5 + 2 * 3
res1: Int = 11

scala> "hello" + " world !"
res2: java.lang.String = hello world !
```

On peut également nommer une sous-expression et réutiliser ensuite ce nom à la place de l'expression :

```
scala> def scale = 5
scale: Int

scala> 7 * scale
res3: Int = 35

scala> def pi = 3.141592653589793
pi: Double

scala> def radius = 10
radius: Int

scala> 2 * pi * radius
res4: Double = 62.83185307179586
```

Les définitions sont introduites par le mot réservé **def** ; elles créent un nom qui désigne l'expression qui suit le signe `=`. L'interpréteur répondra en indiquant ce nom suivi de son type.

L'exécution d'une définition comme `def x = e` n'évaluera pas l'expression `e`. Celle-ci sera évaluée à chaque fois que `x` sera utilisée. Scala permet également de définir une valeur avec `val x = e` : cette définition évalue la partie droite `e` lors de son exécution. Toute utilisation de `x` sera ensuite remplacée par la valeur pré-calculée de `e` afin que l'expression n'ai pas besoin d'être réévaluée.

Une expression formée d'opérateurs et d'opérandes est évaluée en appliquant de façon répétée les étapes de simplification suivantes :

- Prendre l'opération la plus à gauche.
- Évaluer ses opérances.

- Appliquer l'opérateur aux valeurs des opérandes.

Un nom défini par **def** est évalué en remplaçant le nom par sa partie droite (non évaluée). Un nom défini par **val** est évalué en remplaçant le nom par la valeur de sa partie droite. Le processus d'évaluation s'arrête lorsque l'on a obtenu une valeur, c'est-à-dire une donnée comme une chaîne, un nombre, un tableau ou une liste.

Évaluation d'une expression aritmétique.

```
(2 * pi) * radius
(2 * 3.141592653589793) * radius
6.283185307179586 * radius
6.283185307179586 * 10
62.83185307179586
```

Ce processus de simplification par étapes des expressions en valeurs est appelé *réduction*.

Paramètres

Avec **def**, nous pouvons également définir des fonctions avec des paramètres :

```
scala> def square(x: Double) = x * x
square: (Double)Double

scala> square(2)
res0: Double = 4.0

scala> square(5 + 3)
res1: Double = 64.0

scala> square(square(4))
res2: Double = 256.0

scala> def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (Double,Double)Double

scala> sumOfSquares(3, 2 + 2)
res3: Double = 25.0
```

Les paramètres des fonctions sont placés après le nom de la fonction et sont toujours placés entre parenthèses. Chaque paramètre est associé à un type qui est indiqué après le nom du paramètre et le signe deux-points. Ici, nous n'avons besoin que de types numériques de base, comme `scala.Double`, qui décrit les nombres flottants en double précision. Scala définit des *alias* pour certains types standard : vous pouvez donc écrire les types numériques comme en Java – `double`, par exemple, est un alias du type `Scala.Double` et `int` est un alias de `Scala.Int`.

Les fonctions avec des paramètres sont évaluées comme les opérateurs dans les expressions. Les paramètres sont d'abord évalués (de gauche à droite), puis l'appel de fonction est remplacé par sa partie droite et, en même temps, tous ses paramètres formels sont remplacés par les paramètres effectifs correspondants.

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
9 + 16
25
```

Cet exemple montre que l'interpréteur réduit les paramètres des fonctions en valeurs avant de réécrire l'appel de la fonction. On aurait pu choisir d'appliquer la fonction aux paramètres non réduits, ce qui aurait donné la suite de réductions suivante :

```
  sumOfSquares(3, 2+2)
  square(3) + square(2+2)
  3 * 3 + square(2+2)
  9 + square(2+2)
  9 + (2+2) * (2+2)
  9 + 4 * (2+2)
  9 + 4 * 4
  9 + 16
  25
```

Cette seconde évaluation est appelée *appel par nom* alors que la première est appelée *appel par valeur*. Pour les expressions qui n'utilisent que des fonctions pures et qui peuvent donc être réduites par le modèle de substitution, les deux approches donnent le même résultat final.

L'appel par valeur a l'avantage d'éviter de répéter l'évaluation des paramètres alors que l'appel par nom permet d'éviter l'évaluation des paramètres lorsque le paramètre n'est pas utilisé par la fonction. Généralement, l'appel par valeur est plus efficace que l'appel par nom, mais son évaluation peut provoquer une boucle sans fin là ou un appel par nom se serait terminé. Considérons, par exemple, ces deux fonctions :

```
scala> def loop: Int = loop
loop: Int

scala> def first(x: Int, y: Int) = x
first: (Int,Int)Int
```

L'appel `first(1, loop)` avec un appel par nom produira `1` alors qu'avec un appel par valeur il se réduira éternellement en lui-même et l'évaluation ne se terminera donc jamais :

```
  first(1, loop)
  first(1, loop)
  first(1, loop)
  ...
```

Scala utilise par défaut l'appel par valeur mais bascule vers l'appel par nom si le type du paramètre est précédé du symbole **=>**.

```
scala> def constOne(x: Int, y: => Int) = 1
constOne: (Int,=> Int)Int

scala> constOne(1, loop)
res0: Int = 1

scala> constOne(loop, 2)        // boucle infinie
^C                              // Stoppe l'exécution avec Ctrl-C
```

Expressions conditionnelles

Le **if–else** de Scala permet de choisir entre deux alternative. Sa syntaxe est identique au **if–else** de Java mais, là ou la structure de Java ne peut être utilisée que comme une alternative entre instructions, celle de Scala permet de choisir entre deux expressions. C'est la raison pour laquelle le **if–else** de Scala permet également de remplacer les expressions conditionnelles **… ? … : …** de Java.

```
scala> def abs(x: Double) = if (x >= 0) x else -x
abs: (Double)Double
```

Les expressions booléennes de Scala sont identiques à celles de Java : elles sont formées à partir des constantes **true** et **false**, des opérateurs de comparaison, de la négation booléenne `!` et des opérateurs booléens `&&` et `||`.

Exemple : racines carrées par la méthode de Newton

Nous allons maintenant illustrer les éléments du langage que nous venons d'apprendre par un programme un peu plus intéressant. Nous allons écrire une fonction

```
def sqrt(x: Double): Double = ...
```

qui calcule la racine carrée de `x`.

Une méthode classique pour calculer les racines carrées est celle de Newton, qui consiste à réaliser des approximations successives. On part d'une approximation initiale `y` (`y = 1`, par exemple) puis on améliore progressivement cette approximation en prenant la moyenne de `y` et `x/y`. Les trois colonnes suivantes, par exemple, indiquent les valeurs de `y`, celles du quotient `x/y` et de leurs moyennes pour les premières approximations de la racine carrée de 2 :

```
1          2/1 = 2              1.5
1.5        2/1.5 = 1.3333       1.4167
1.4167     2/1.4167 = 1.4118    1.4142
1.4142     ...                  ...
y          x/y                  (y + x/y)/2
```

Nous pouvons implémenter cet algorithme en Scala à l'aide d'un ensemble de petites fonctions qui, chacune, représente l'un des éléments de cet algorithme. Commençons par définir une fonction qui itère à partir d'une approximation pour arriver à un résultat :

```
def sqrtIter(guess: Double, x: Double): Double =
  if (isGoodEnough(guess, x)) guess
  else sqrtIter(improve(guess, x), x)
```

Vous remarquerez que `sqrtIter` s'appelle récursivement. Les boucles des programmes impératifs peuvent toujours être modélisées par des appels récursifs dans un programme fonctionnel.

Notez également que la définition de `sqrtIter` indique le type du résultat, après la liste des paramètres. C'est obligatoire pour les fonctions récursives. Pour les autres, il est facultatif : le vérificateur de type le déduira à partir de la partie droite de la fonction. Cependant, même pour les fonctions récursives, il est généralement conseillé de préciser explicitement le type du résultat car il participe à la documentation de cette fonction.

La seconde étape consiste à définir les deux fonctions appelées par `sqrtIter` : une fonction pour améliorer l'approximation et une fonction de test `isGoodEnough` pour terminer l'appel :

```
def improve(guess: Double, x: Double) =
  (guess + x / guess) / 2

def isGoodEnough(guess: Double, x: Double) =
  abs(square(guess) - x) < 0.001
```

Enfin, la fonction `sqrt` elle-même est définie par une application de `sqrIter` :

```
def sqrt(x: Double) = sqrtIter(1.0, x)
```

Exercice 4–4–1 : Le test `isGoodEnough` n'est pas très précis pour les petits nombres et pourrait provoquer une boucle sans fin pour les très grands nombres (pourquoi ?). Proposez une autre version de `isGoodEnough` qui résoud ces problèmes.

Exercice 4–4–2 : Faites la trace de l'exécution de l'expression `sqrt(4)`.

Fonctions imbriquées

Le style de programmation fonctionnelle encourage la construction de nombreuses petites fonctions auxiliaires. Dans le dernier exemple, l'implémentation de `sqrt` utilise les fonctions auxiliaires `sqrIter`, `improve` et `isGoodEnough` mais, les noms de ces fonctions n'ayant un sens que pour l'implémentation de `sqrt`, il est préférable que les utilisateurs ne puissent pas y accéder directement.

Nous pouvons imposer cette restriction (et éviter la pollution de l'espace de noms) en incluant ces fonctions auxiliaires dans la fonction appelante :

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)
  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2
  def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001 sqrtIter(1.0, x)
}
```

Dans ce code, les accolades `{…}` délimitent un *bloc*. En Scala, les blocs sont eux-mêmes des expressions : chaque bloc se termine par une expression résultat qui définit sa valeur. Cette expression résultat peut être précédée de définitions auxiliaires, qui ne seront visibles que dans le bloc.

Dans un bloc, chaque définition doit être suivie d'un point-virgule qui sépare cette définition des définitions suivantes ou de l'expression résultat. Ce point-virgule est inséré explicitement à la fin de chaque ligne, sauf dans les cas suivants :

1. La ligne se termine par un point ou par un opérateur infixe qui ne peut pas apparaître à la fin d'une expression.

2. La ligne suivante commence par un mot qui ne peut pas débuter une expression.

3. Nous sommes entre des parenthèses `(…)` ou des crochets, qui ne peuvent pas contenir plusieurs instructions.

Les exemples suivants sont donc tous corrects :

```
def f(x: Int) = x + 1;
f(1) + f(2)

def g1(x: Int) = x + 1
g(1) + g(2)

def g2(x: Int) = {x + 1}; /* ';' obligatoire */ g2(1) + g2(2)

def h1(x) =
  x +
  y
h1(1) * h1(2)

def h2(x: Int) = (
  x      // Parenthèses obligatoires, sinon un point-virgule
  + y    // serait inséré après le x
)
h2(1) / h2(2)
```

Scala utilise les règles de portée habituelles des blocs. Un nom défini dans un bloc externe est également visible dans un bloc interne à condition qu'il n'y soit pas redéfini. Cette règle permet de simplifier notre exemple `sqrt` car nous n'avons plus besoin de passer `x` en paramètre aux fonctions imbriquées puisqu'il sera toujours visible en tant que paramètre de la fonction externe `sqrt`. Voici le code simplifié :

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else sqrtIter(improve(guess))
  def improve(guess: Double) =
    (guess + x / guess) / 2
  def isGoodEnough(guess: Double) =
    abs(square(guess) - x) < 0.001
  sqrtIter(1.0)
```

Récursivité terminale

Considérons la fonction suivante, qui calcule le plus grand diviseur commun de deux nombres :

```
def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
```

Avec notre modèle de substitution pour l'évaluation des fonctions, l'évaluation de gcd(14, 21) sera :

```
  gcd(14, 21)
  if (21 == 0) 14 else gcd(21, 14 % 21)
  if (false) 14 else gcd(21, 14 % 21)
  gcd(21, 14 % 21)
  gcd(21, 14)
  if (14 == 0) 21 else gcd(14, 21 % 14)
  gcd(14, 21 % 14)
  gcd(14, 7)
  if (7==0) 14 else gcd(7, 14 % 7)
  gcd(7,14 % 7)
  gcd(7, 0)
  if(0 = =0) 7 else gcd(0, 7 % 0)
  7
```

Comparez cette évaluation avec celle d'une autre fonction récursive, factorielle :

```
def factorielle(n: Int): Int = if (n == 0) 1 else n * factorielle(n - 1)
```

L'appel factorielle(5) s'effectue de la façon suivante :

```
        factorielle(5)
        if (5 == 0) 1 else 5 * factorielle(5 - 1)
        5 * factorielle(5 - 1)
        5 * factorielle(4)
  ...   5   * (4 * factorielle(3))
  ...   5   * (4 * (3 * factorielle(2)))
  ...   5 * (4 * (3 * (2 * factorielle(1))))
  ...   5 * (4 * (3 * (2 * (1 * factorielle(0)))))
  ...   5 * (4 * (3 * (2 * (1 * 1))))
  ...   120
```

On note une différence importante entre ces deux séquences de réécriture : les termes de la séquence de réécriture de `gcd` ont toujours la même forme. À mesure que l'évaluation progresse, leur taille est limitée par une constante. Dans l'évaluation de `factorielle`, par contre, nous obtenons des chaînes d'opérandes de plus en plus longues qui sont ensuite multipliées dans la dernière partie de l'évaluation.

Bien que la véritable implémentation de Scala ne fonctionne pas en réécrivant les termes, ils devraient avoir le même comportement en termes d'espace que dans ces séquences de réécriture. Dans l'implémentation de `gcd`, on note que l'appel récursif est la dernière action réalisée dans l'évaluation du corps. On dit aussi que `gcd` est "récursive terminale". Le dernier appel d'une fonction récursive terminale peut être implémenté par un saut revenant au début de cette fonction. En outre, les paramètres de cet appel peuvent écraser les paramètres de l'instance courante de `gcd` puisqu'ici on n'a pas besoin d'un nouvel espace de pile. Les fonctions récursives terminales sont donc des processus itératifs qui peuvent s'exécuter dans un espace constant.

L'appel récursif dans `factorielle`, par contre, est suivi d'une multiplication. Il faut donc allouer un nouveau cadre de pile pour l'instance récursive de `factorielle` et le désallouer lorsque cette instance se sera terminée. Notre formulation actuelle de `factorielle` n'est pas récursive terminale et, pour s'exécuter, a besoin d'un espace proportionnel à son paramètre d'entrée.

Plus généralement, si la dernière action d'une fonction est un appel à une autre fonction (qui peut être elle-même), il suffit d'un seul cadre de pile pour les deux fonctions. Ces appels sont appelés "appels terminaux" et peuvent, en principe, toujours réutiliser le cadre de pile de la fonction appelante. Cependant, certains environnements d'exécution (c'est le cas de la machine virtuelle Java) ne disposent pas des primitives permettant de réutiliser un cadre de pile pour optimiser les appels terminaux. On exige par conséquent uniquement d'une implémentation Scala qu'elle puisse réutiliser le cadre de pile d'une fonction récursive terminale directe, dont la derniere action consiste à s'appeler elle-même. Les autres appels terminaux pourraient également être optimisés, mais cela est propre à l'implémentation et donc non garanti.

Exercice 4–6–1 : Écrivez une version récursive terminale de `factorielle`.

Chapitre 5. Fonctions du premier ordre

En Scala, une fonction est une "valeur de première classe". Comme n'importe quelle autre valeur, elle peut être passée en paramètre ou renvoyée comme résultat. Les fonctions qui prennent en paramètres d'autres fonctions ou qui les renvoient comme résultat sont appelées fonction du *premier ordre*. Ce chapitre présente ces fonctions de premier ordre et montre qu'elles constituent un mécanisme souple et puissant pour la composition des programmes.

À titre d'exemple, considérons ces trois tâches apparentées :

1. Écrire une fonction additionnant tous les entiers compris entre deux bornes `a` et `b` :

```
def sumInts(a: Int, b: Int): Int =
  if (a > b) 0 else a + sumInts(a + 1, b)
```

2. Écrire une fonction additionnant les carrés de tous les entiers compris entre deux bornes `a` et `b` :

```
def square(x: Int): Int = x * x
def sumSquares(a: Int, b: Int): Int =
  if (a > b) 0 else square(a) + sumSquares(a + 1, b)
```

3. Écrire une fonction additionnant les puissances de 2 de tous les entiers compris entre deux bornes `a` et `b` :

```
def powerOfTwo(x: Int): Int = if (x == 0) 1 else 2 * powerOfTwo(x - 1)
def sumPowersOfTwo(a: Int, b: Int): Int =
  if (a > b) 0 else powerOfTwo(a) + sumPowersOfTwo(a + 1, b)
```

Ces fonctions étant toutes des instances de "Somme de `f(n)` pour `n` variant de `a` à `b`", nous pouvons mettre ce motif commun en facteur en définissant une fonction `sum` :

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

Le type `Int => Int` est celui des fonctions qui prennent un paramètre de type Int et qui renvoient un résultat de type Int. `sum` est donc une fonction qui prend en paramètre une autre fonction – en d'autres termes, c'est une *fonction du premier ordre*.

Grâce à elle, nous pouvons reformuler nos trois fonctions précédentes de la façon suivante :

```
def sumInts(a: Int, b: Int): Int = sum(id, a, b)
def sumSquares(a: Int, b: Int): Int = sum(square, a, b)
def sumPowersOfTwo(a: Int, b: Int): Int = sum(powerOfTwo, a, b)
```

avec :

```
def id(x: Int): Int = x
def square(x: Int): Int = x * x
def powerOfTwo(x: Int): Int = if (x == 0) 1 else 2 * powerOfTwo(x - 1)
```

Fonctions anonymes

Le fait de passer des fonctions en paramètre pousse à créer de nombreuses petites fonctions. Dans l'exemple précédent, nous avons défini `id`, `square` et `powerOfTwo` comme des fonctions séparées afin de pouvoir les passer en paramètre à `sum`.

Au lieu d'utiliser des définitions de fonctions nommées pour ces paramètres, nous pouvons les formuler plus simplement à l'aide de *fonctions anonymes*. Une fonction anonyme est une expression dont l'évaluation produit une fonction – cette fonction est définie sans qu'on lui donne un nom. À titre d'exemple, voici une fonction anonyme qui calcule le carré d'un entier :

```
(x: Int) => x * x
```

La partie située avant la flèche `=>` est la liste des paramètres de la fonction, la partie située après est son corps. Voici une fonction anonyme qui multiplie ses paramètres :

```
(x: Int, y: Int) => x * y
```

Grâce aux fonctions anonymes, nous pouvons reformuler les deux premières fonctions de sommation sans utiliser de fonctions auxiliaires nommées :

```
def sumInts(a: Int, b: Int): Int = sum((x: Int) => x, a, b)
def sumSquares(a: Int, b: Int): Int = sum((x: Int) => x * x, a, b)
```

Le plus souvent, le compilateur Scala peut déduire les types des paramètres à partir du contexte de la fonction anonymes, auquel cas ceux-ci peuvent être omis. Pour `sumInts` et `sumSquares`, par exemple, le type de `sum` nous indique que le premier paramètre doit être une fonction de type `Int => Int`. Le type `Int` du paramètre de la fonction anonyme est donc redondant et peut être omis. En outre, lorsqu'il n'y a qu'un seul paramètre et que l'on ne précise pas le type, nous pouvons également supprimer les parenthèses qui l'entourent :

```
def sumInts(a: Int, b: Int): Int = sum(x => x, a, b)
def sumSquares(a: Int, b: Int): Int = sum(x => x * x, a, b)
```

Généralement, le terme Scala `(`x_1`: `T_1`, …, `x_n`: `T_n`) => E` définit une fonction qui fait correspondre à ses paramètres $x_1, \ldots, x_n$ le résultat de l'expression `E` (où `E` peut utiliser $x_1, \ldots, x_n$). Les fonctions anonymes ne sont pas des éléments de langage essentiels de Scala car elles peuvent toujours s'exprimer en termes de fonctions nommées. La fonction anonyme

```
(x1: T1, ..., xn: Tn) => E
```

est équivalente au bloc

```
{ def f (x1: T1, ..., xn: Tn) = E ; f _ }
```

où `f` est un nouveau nom qui n'est utilisé nulle part ailleurs dasn le programme. On dit aussi que les fonctions anonymes sont du "sucre syntaxique".

Curryfication

Notre dernière formulation des fonctions de sommation est déjà très compacte, mais nous pouvons faire mieux. Vous remarquerez que `a` et `b` apparaissent comme paramètres de chaque fonction mais qu'ils ne semblent assez "statiques". Peut-on donc s'en débarrasser ? Réécrivons `sum` pour qu'elle ne prenne pas les bornes `a` et `b` en paramètre :

```
def sum(f: Int => Int): (Int, Int) => Int = {
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)
  sumF
}
```

Ici, `sum` est une fonction qui renvoie une autre fonction, celle qui effectue la somme : `sumF`. C'est cette dernière qui fait tout le travail : elle prend les bornes `a` et `b` en paramètre, elle applique la fonction `f` qui a été passée à `sum` à tous les entiers compris entre ces bornes et additionne les résultats.

Grâce à cette nouvelle formulation de `sum`, nous pouvons maintenant écrire :

```
def sumInts = sum(x => x)
def sumSquares = sum(x => x * x)
def sumPowersOfTwo = sum(powerOfTwo)
```

Ou, avec des définitions de valeurs :

```
val sumInts = sum(x => x)
val sumSquares = sum(x => x * x)
val sumPowersOfTwo = sum(powerOfTwo)
```

`sumInts`, `sumSquares` et `sumPowersOfTwo` peuvent s'appliquer comme n'importe quelle autre fonction :

```
scala> sumSquares(1, 10) + sumPowersOfTwo(10, 20)
res0: Int = 2096513
```

Comment s'appliquent les fonctions qui renvoient des fonctions ? Dans l'expression `sum(x => x * x)(1, 10)`, par exemple, la fonction `sum` peut s'appliquer à la fonction carré (`x => x * x`) et la fonction résultante est ensuite appliquée à la seconde liste de paramètres `(1, 10)`. Cette notation est possible car l'application des fonctions est associative à gauche : si $\texttt{params}_1$ et $\texttt{params}_2$ sont des listes de paramètres, alors `f(params`$_1$`)(params`$_2$`)` est équivalent à `(f(params`$_1$`))(params`$_2$`)`.

Dans notre exemple, `sum(x => x * x)(1, 10)` est donc équivalent à `(sum(x => x * x))(1, 10)`.

Les fonctions qui renvoient des fonctions sont si utiles que Scala dispose d'une syntaxe spéciale pour les exprimer. La définition suivante de `sum`, par exemple, est équivalente à la précédente mais est plus courte :

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```

De façon générale, une définition de fonction curryfiée où `n > 1` :

```
f (params_1) ... (params_n) = E
```

est traduite en :

```
def f (params_1) ... (params_n-1) = { def g (params_n) = E ; g }
```

où *g* est un nouveau identificateur. Avec une fonction anonyme, cette traduction est encore raccourcie :

```
def f (params_1) ... (params_n-1) = ( params_n ) => E
```

Si l'on applique `n` fois cette transformation, on trouve donc que :

```
def f (params_1) ... (params_n) = E
```

est équivalent à :

```
def f = (params_1) => ... => (params_n) => E
```

ou, avec une définition de valeur :

```
val f = (params_1) => ... => (params_n) => E
```

Ce style de définition et d'application des fonctions est appelé *curryfication*, en hommage aux travaux de Haskell B. Curry, mathématicien du 20e siècle – bien que ce concept remonte aux articles de Moses Schönfinkel et Gottlob Frege.

Le type d'une fonction renvoyant une fonction s'exprime comme dans une liste de paramètres. Le type de la dernière formulation de `sum`, par exemple, est `(Int => Int) => (Int, Int) => Int`. Cette écriture est possible car les types des fonctions sont associatifs à droite : T_1 `=>` T_2 `=>` T_3 est donc équivalent à T_1 `=> (`T_2 `=>` T_3`)`.

Exercice 5–2–1 : La fonction `sum` utilise une récursion linéaire. Écrivez une version récursive terminale en remplaçant les `??` :

```
def sum(f: Int => Int)(a: Int, b: Int): Int = {
  def iter(a: Int, result: Int): Int = {
    if (??) ??
    else iter(??, ??)
  }
  iter(??, ??)
}
```

Exercice 5–2–2 : Écrivez une fonction `product` qui calcule le produit des valeurs d'une fonction pour les points appartenant à un certain intervalle.

Exercice 5–2–3 : Écrivez `factorielle` en termes de `product`.

Exercice 5–2–4 : Pouvez-vous écrire une fonction encore plus générale, qui généralise à la fois `sum` et `product` ?

Exemple : Trouver les points fixes des fonctions

Un nombre `x` est appelé *point fixe* d'une fonction `f` si `f(x) = x`.

Pour certaines fonctions `f`, nous pouvons trouver le point fixe en commençant par une valeur supposée et en appliquant `f` jusqu'à ce que cette valeur ne change plus (ou que la modification soit inférieure à une tolérance donnée). Ceci est possible si la suite

```
x, f(x), f(f(x)), f(f(f(x))), ...
```

converge vers le point fixe de `f`. Ce principe est capturé par la fonction suivante :

```
val tolerance = 0.0001
def isCloseEnough(x: Double, y: Double) = abs((x - y) / x) < tolerance
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
 iterate(firstGuess)
}
```

Nous pouvons maintenant appliquer ce principe pour reformuler la fonction racine carrée que nous avions étudiée précédemment. Commençons par spécifier `sqrt` :

```
sqrt(x) =  y tel que y * y = x
        =  y tel que y = x  /y
```

Par conséquent, `sqrt(x)` est un point fixe de la fonction `y => x / y`. Ceci suggère qu'elle peut être calculée à l'aide de `fixedPoint` :

```
def sqrt(x: double) = fixedPoint(y => x / y)(1.0)
```

Cependant, nous constatons alors que ce calcul ne converge pas. Ajoutons à la fonction point fixe une instruction `print` affichant la supposition courante :

```
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    println(next)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

L'appel `sqrt(2)` produit alors :

```
2.0
1.0
2.0
1.0
2.0
...
```

Un moyen de contrôler ces oscillations consiste à empêcher la supposition de trop changer. Ceci peut se faire en calculant la *moyenne* des valeurs successives de la suite initiale :

```
scala> def sqrt(x: Double) = fixedPoint(y => (y + x/y) / 2)(1.0)
sqrt: (Double)Double

scala> sqrt(2.0)
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
1.4142135623746899
```

En fait, la traduction de la fonction `fixedPoint` produirait exactement notre définition précédente du point fixe de la section 4–4.

Les exemples précédents ont montré que la puissance expressive d'un langage est considérablement améliorée lorsque l'on peut passer des fonctions en paramètre. L'exemple suivant montre que les fonctions qui renvoient des fonctions peuvent également être très utiles.

Reprenons à nouveaux nos itérations pour trouver le point fixe. Nous sommes partis de l'observation que `(x)` est un point fixe de la fonction `y => x / y`. Pius, nous avons fait en sorte que l'itération converge en faisant la moyenne des valeurs successives. Cette technique d'amortissement par la moyenne est si générale que nous pouvons l'envelopper dans une autre fonction :

```
def averageDamp(f: Double => Double)(x: Double) = (x + f(x)) / 2
```

Grâce à `averageDamp`, nous pouvons reformuler la fonction racine carrée :

```
def sqrt(x: Double) = fixedPoint(averageDamp(y => x/y))(1.0)
```

Cette définition exprime aussi clairement que possible les éléments de l'algorithme.

Exercice 5–3–1 : Écrivez une fonction calculant les racines cubiques en vous servant de `fixedPoint` et de `averageDamp`.

Résumé

Le chapitre précédent a montré que les fonctions sont des abstractions essentielles car elles permettent d'introduire des méthodes générales de calcul sous la forme d'éléments explicites et nommés dans le langage. Ce chapitre, quant à lui, a montré que ces abstractions pouvaient être combinées en fonctions du premier ordre afin de créer d'autres abstractions. En tant que programmeurs, nous devons saisir les opportunités d'abstraction et de réutilisation. Le niveau d'abstraction le plus haut n'est pas toujours le meilleur, mais il est important de connaître les techniques d'abstraction pour pouvoir les utiliser à bon escient.

Éléments du langage déjà vus

Les chapitres 4 et 5 ont présenté les éléments du langage qui permettent de créer des expressions et des types formés de données primitives et de fonctions. Nous indiquerons ci-dessous la syntaxe sans contexte de ces éléments selon le formalisme Backus-Naur étendu, où `|` représente une alternative, `[…]` un élément facultatif (0 ou 1 occurrence) et `{…}` une répétition (0 ou plusieurs occurrences).

Caractères

Les programmes Scala sont des suites de caractères Unicode. Nous distinguons les jeux de caractères suivants :

- Les espaces, comme `' '`, la tabulation ou les caractères de fin de ligne.
- Les lettres de `'a'` à `'z'` et de `'A'` à `'Z'`.
- Les chiffres de `'0'` à `'9'`.
- Les délimiteurs `. , ; ( ) { } [ ] \ " '`
- Les opérateurs comme `'#' '+' ':'`. Essentiellement, il s'agit des caractères affichables qui n'appartiennent à aucun des ensembles précédents.

Lexèmes

```
ident     = lettre {lettre | chiffre}
          | opérateur { opérateur }
          | ident '_' ident
littéral  = "comme en Java"
```

Les littéraux sont les mêmes qu'en Java. Il définissent des nombres, des caractères, des chaînes ou des valeurs booléennes. Des exemples de littéraux sont `0`, `1.0e10`, `'x'`, `"il a dit "bonjour !""` ou **true**.

Les identificateurs ont deux formes. Ils peuvent commencer par une lettre suivie (éventuellement) d'une suite de lettres ou de symboles, ou commencer par un caractère opérateur, suivi (éventuellement) d'une suite de caractères opérateurs. Les deux formes peuvent contenir des caractères blanc souligné `'_'`. En outre, un blanc souligné peut être suivi par une forme ou l'autre d'identificateur. Les identificateurs suivants sont donc corrects :

```
x         Room10a       +         --      foldl_:         +_vector
```

Cette règle signifie que les identificateurs formés d'opérateurs doivent être séparés par des espaces. L'entrée `x+-y` sera donc analysée comme une séquence de trois lexèmes `x`, `+-` et `y`. Si nous voulions exprimer la somme de `x` avec l'opposé de `y`, nous devrions ajouter un espace pour écrire, par exemple, `x+ -y`.

Le caractère `$` est réservé pour les identificateurs produits par le compilateur ; ne l'utilisez pas dans les sources de vos programmes.

Les mots suivants sont réservés et ne peuvent donc pas servir d'identificateurs :

```
abstract  case      catch       class     def
do        else      extends     false     final
finally   for       if          implicit  import
match     new       null        object    override
package   private   protected   requires  return
sealed    super     this        throw     trait
try       true      type        val       var
while     with      yield
_   :    =    =>    <-   <:    <%   >:   #     @
```

Types

```
Type          = TypeSimple | TypeFonction
TypeFonction  = TypeSimple '=>' Type | '(' [Types] ')' '=>' Type
TypeSimple    = Byte | Short | Char | Int | Long | Float | Double |
                Boolean | Unit | String
Types         = Type {',' Type}
```

Les types peuvent être :

- Des types numériques : `Byte`, `Short`, `Char`, `Int`, `Long`, `Float` et `Double` (ce sont les mêmes qu'en Java).
- Le type `Boolean` avec les valeurs **true** et **false**.
- Le type `Unit` avec la seule valeur `()`.
- Le type `String`.
- Des types fonctions, comme `(Int, Int) => Int` ou `String => Int => String`.

Expressions

```
Expr          = ExprInfixe | ExprFonction | if '(' Expr ')' Expr else Expr
ExprInfixe    = ExpPréfixe | ExprInfixe Opérateur ExprInfixe
Opérateur     = ident
ExprPréfixe   =  ['+' | '-' | '!' | '~' ] ExprSimple
ExprSimple    = ident | littéral | ExprSimple '.' ident | Bloc
ExprFonction  = (Liaisons | Id) '=>' Expr
Liaisons      =  '(' Laison {',' Liaison} ')'
Liaison       = ident [':' Type]
Bloc          = '{' {Def ';'} Expr '}'
```

Les expressions peuvent être :

- Des identificateurs comme `x`, `isGoodEnough`, `*` ou `+-`
- Des littéraux comme `0`, `1.0` ou `"abc"`
- Des sélections de champs ou de méthodes comme `System.out.println`

- Des appels de fonctions comme `sqrt(x)`
- Des applications d'opérateurs comme `-x` ou `y + x`
- Des conditionnelles comme **if** `(x < 0) -x` **else** `x`
- Des blocs comme `{` **val** `x = abs(y) ; x * 2 }`
- Des fonctions anonymes comme `x => x + 1` ou `(x: Int, y: Int) => x + y`.

Définitions

```
Def         = DefFonction | DefValeur
DefFonction = 'def' ident {'(' [Paramètres] ')'} [':' Type] '=' Expr
DefValeur   = 'val' ident [':' Type] '=' Expr
Paramètres  = Paramètre {',' Paramètre}
Paramètre   = ident ':' ['=>'] Type
```

Les définitions peuvent être :

- Des définitions de fonctions comme **def** `square(x: Int): Int = x * x`
- Des définitions de valeurs comme **val** `y = square(2)`.

Chapitre 6. Classes et objets

Scala n'a pas de type prédéfini pour représenter les nombres rationnels, mais il est relativement simple d'en définir un à l'aide d'une classe. Voici une implémentation possible :

```
class Rational(n: Int, d: Int) {
  private def gcd(x: Int, y: Int): Int = {
    if (x == 0) y
    else if (x < 0) gcd(-x, y)
    else if (y < 0) -gcd(x, -y)
    else gcd(y % x, x)
  }
  private val g = gcd(n, d)

  val numer: Int = n/g
  val denom: Int = d/g
  def +(that: Rational) =
    new Rational(numer * that.denom + that.numer * denom,
                 denom * that.denom)
  def -(that: Rational) =
    new Rational(numer * that.denom - that.numer * denom,
                 denom * that.denom)
  def *(that: Rational) =
    new Rational(numer * that.numer, denom * that.denom)
  def /(that: Rational) =
    new Rational(numer * that.denom, denom * that.numer)
}
```

Ce code définit `Rational` comme une classe avec un constructeur prenant deux paramètres `n` et `d` correspondant au numérateur et au dénominateur. Cette classe fournit des champs qui renverront ces composantes ainsi que des méthodes permettant d'effectuer les opérations arithmétiques classiques sur des nombres rationnels. Chacune de ces méthodes prend en paramètre l'opérande droit de l'opération – l'opérande gauche est toujours le nombre rationnel récepteur de l'appel.

Membres privés. L'implémentation des nombres rationnels définit une méthode privée `gcd` qui calcule le plus grand dénominateur commun de deux entiers et un champ privé `g` qui contient le pgcd des paramètres passés au constructeur. Ces membres sont inacessibles à l'extérieur de la classe Rational et sont utilisés par l'implémentation pour éliminer les facteurs communs afin de garantir que le numérateur et le dénominateur soient toujours sous une forme simplifiée.

Création et accès aux objets. Voici un programme qui affiche la somme de tous les nombres `1/i`, avec `i` variant de 1 à 10 :

```
var i = 1
var x = new Rational(0, 1)
while (i <= 10) {
  x += new Rational(1, i)
  i += 1
}
println("" + x.numer + "/" + x.denom)
```

Ici, l'opérateur `+` de la dernière instruction prend une chaîne comme opérande gauche et une valeur de type quelconque comme opérande droit. Il convertit cet opérande droit en chaîne et renvoie le résultat de sa concaténation avec l'opérande gauche.

Héritage et redéfinition. Toute classe Scala étend une superclasse. Si la classe ne mentionne pas de superclasse dans sa définition, Scala considère qu'il s'agit du type racine `scala.AnyRef` (dans les implémentations Java, ce type est un alias de `java.lang.Object`). La classe `Rational` aurait donc également pu être définie de la façon suivante :

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // comme précédemment
}
```

Une classe hérite de tous les membres de sa superclasse. Elle peut également *redéfinir* certains d'entre eux. La classe `java.lang.Object`, par exemple, définit une méthode `toString` qui renvoie une représentation de l'objet sous forme de chaîne :

```
class Object {
  ...
  def toString: String = ...
}
```

L'implémentation de `toString` dans `Object` renvoie une chaîne contenant le nom de la classe de l'objet suivi d'un nombre. Il est donc souhaitable de redéfinir cette méthode pour les nombres rationnels :

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // Comme précédemmment
  override def toString = "" + numer + "/" + denom
}
```

Notez qu'à la différence de Java, les définitions redéfinies doivent être précédées du modificateur **override**.

Si la classe *A* étend la classe *B*, vous pouvez utiliser des objets de type *A* à chaque fois que des objets de type *B* sont attendus – le type *A* est dit *conforme* au type *B*. Ici, `Rational` étant conforme à `AnyRef`, vous pouvez donc affecter une valeur `Rational` à une variable de type `AnyRef` :

```
var x: AnyRef = new Rational(1, 2)
```

Méthodes sans paramètres. À la différence de Java, les méthodes Scala n'ont pas nécessairement une liste de paramètres comme le montre la méthode square ci-dessous. Cette méthode est simplement invoquée en mentionnant son nom.

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // comme précédemment
  def square = new Rational(numer*numer, denom*denom)
}
val r = new Rational(3, 4)
println(r.square)                  // affiche "9/16"
```

L'accès aux méthodes sans paramètres est donc identique à l'accès aux valeurs des champs comme `numer`. La différence se situe au niveau de leurs définitions : la partie droite d'une valeur est évaluée lorsque l'objet est créé et ne change plus après, tandis que la partie droite d'une méthode sans paramètres est évaluée à chaque appel de cette méthode. Cet accès uniforme au champs et aux méthodes sans paramètres donne bien plus de souplesse au développeur de la classe : souvent, un champ dans une version d'une classe devient une valeur calculée dans la version suivante. L'accès uniforme garantit que les clients n'auront pas besoin d'être réécrits à cause de cette modification.

Classes abstraites. Supposons que nous voulions écrire une classe pour représenter des ensembles d'entiers, dotée des deux opérations `incl` et `contains`. `(s incl x)` devra renvoyer un nouvel ensemble contenant l'élément `x` et tous les éléments de `s`. `(s contains x)` renverra **true** si l'ensemble `s` contient l'élément `x` ; **false** dans le cas contraire. L'interface de ces ensemble est décrit par le code suivant :

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

`IntSet` est une classe *abstraite*, ce qui a deux conséquences. La première est qu'une classe abstraite peut déclarer des membres qui n'ont pas d'implémentation (ces membres sont dits *différés*). Ici, `incl` et `contains` sont dans ce cas. La seconde est qu'une classe abstraite pouvant contenir des membres non implémentés, on ne peut pas créer d'objets de cette classe avec **new**. En revanche, une classe abstraite peut servir de classe de base pour une autre classe qui implémentera les membres non encore définis.

Traits. À la place d'une classe abstraite, on peut également souvent utiliser le mot-clé `trait` en Scala. Les traits sont des classes abstraites conçues pour être ajoutées à d'autres classes. Un trait permet d'ajouter certaines méthodes ou certains champs à une classe encore inconnue. Un trait `Bordered`, par exemple, pourrait servir à ajouter un contour à différents composants graphiques. Un autre cas d'utilisation est celui où un trait rassemble les signatures d'une fonctionnalité fournie par différentes classes, comme le font les interfaces Java.

IntSet appartenant à cette catégorie, nous pouvons donc également le définir comme un trait :

```
trait IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

Implémentation des classes abstraites. Supposons que nous voulions implémenter nos ensembles comme des arbres binaires. Il y a deux formes possibles pour les arbres : un arbre pour représenter l'ensemble vide et un arbre formé d'un entier et de deux sous-arbre pour un ensemble non vide. Voici leurs implémentations :

```
class EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, new EmptySet, new EmptySet)
}

class NonEmptySet(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

`EmptySet` et `NonEmptySet` étendent toutes les deux la classe `IntSet`, ce qui implique que les types `EmptySet` et `NonEmptySet` sont conformes au type `IntSet` – on peut donc utiliser une valeur de type `EmptySet` ou `NonEmptySet` partout où l'on attend une valeur de type `IntSet`.

Exercice 6–0–1 : Écrire les méthodes `union` et `intersection` qui renvoient respectivement l'union et l'intersection de deux ensembles.

Exercice 6–0–2 : Ajouter la méthode `def excl(x: Int)` qui renvoie l'ensemble sans l'élément `x`. Pour ce faire, il est préférable d'implémenter la méthode `def isEmpty: Boolean` qui teste si l'ensemble est vide.

Liaison Dynamique. Les langages orientés objets (dont Scala) utilisent une *recherche dynamique* pour trouver les méthodes qui sont appelées. Le code invoqué par un appel de méthode dépend donc du type qu'a l'objet lors de l'exécution. Supposons par exemple que s a été déclaré par IntSet et que l'on utilise l'expression (s contains 7) : le code qui sera exécuté lors de l'appel à `contains` dépendra du type de la valeur de s à l'exécution. Si c'est une valeur EmptySet, c'est l'implémentation de contains dans la classe EmptySet qui s'exécutera (même principe si cette valeur est de type NonEmptySet). Ce comportement est une conséquence directe du modèle de substitution utilisé pour l'évaluation :

```
(new EmptySet).contains(7)
```

renvoie **false** après remplacement de `contains` par son corps dans la classe `EmptySet`.

```
new NonEmptySet(7, new EmptySet, new EmptySet).contains(1)
```

après remplacement de `contains` par son corps dans la classe `NonEmptySet` donne :

```
if (1 < 7) new EmptySet contains 1
else if (1 > 7) new EmptySet contains 1
else true
```

soit, après réécriture de la conditionnelle :

```
new EmptySet contains 1
```

et le résultat final est donc **false**.

La recherche dynamique des méthodes est analogue aux appels de fonctions du premier ordre car, dans les deux cas, l'identité du code qui sera exécuté n'est connu qu'au moment de l'exécution. Cette similitude n'est pas accidentelle : Scala représente chaque valeur de fonction par un objet (voir la section 8–6).

Objets. Dans l'implémentation précédente des ensembles d'entiers, les ensembles vides étaient créés par **new** `EmptySet` : un nouvel objet était donc créé à chaque fois que l'on avait besoin d'une valeur ensemble vide. Nous aurions pu éviter cette création inutile en définissant une fois pour toute une valeur "ensemble vide" et en l'utilisant à la place de chaque occurrence de **new** `EmptySet` :

```
val EmptySetVal = new EmptySet
```

Un problème de cette approche est qu'une définition de valeur comme celle-ci ne peut pas être une définition de premier niveau en Scala : elle doit faire partie d'une autre classe ou d'un objet. En outre, la définition de la classe `EmptySet` semble désormais un peu superflue – pourquoi définir une classe d'objet si l'on n'a besoin que d'un seul objet de cette classe ? Une approche plus directe consiste donc à utiliser une *définition d'objet* :

```
object EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, EmptySet, EmptySet)
}
```

La syntaxe d'une définition d'objet suit la syntaxe d'une définition de classe ; elle peut avoir une clause `extends` et elle a un corps. Comme pour les classes, la clause `extends` définit les membres hérités par l'objet tandis que le corps définit les nouveaux membres ou redéfinit certains membres hérités. Une définition d'objet, par contre, ne définit qu'un seul objet : il est impossible de créer d'autres objets de même structure à l'aide de **new**. Les définitions d'objets n'ont donc pas non plus de paramètres constructeurs, alors qu'une définition de classe peut en utiliser.

Les définitions d'objets peuvent apparaître n'importe où dans un programme Scala, y compris au premier niveau. L'ordre d'exécution des entités de premier niveau n'étant pas fixé par Scala, vous pourriez vous demander quand l'objet ainsi défini est exactement créé et initialisé : la réponse est que cet objet est créé dès que l'un de ses membres est utilisé. Cette stratégie est appelée *évaluation paresseuse*.

Classes standard. Scala est un langage orienté objet pur, ce qui signifie que toute valeur peut être considéré comme un objet. En fait, même les types primitifs comme `int` ou `boolean` ne sont pas traités spécialement : ils sont définis comme des alias de classes Scala dans le module Predef :

```
type boolean = scala.Boolean
type int = scala.Int
type long = scala.Long
...
```

Pour des raisons d'efficacité, le compilateur représente généralement les valeurs de type `scala.Int` par des entiers sur 32 bits, les valeurs `scala.Boolean` par des booléens Java, etc. Mais il convertit ces représentations spéciales en objets lorsque cela est nécessaire – lorsque, par exemple, une valeur primitive entière est passée à une fonction qui attend un paramètre de type `AnyRef`. La représentation des valeurs primitives n'est donc qu'une optimisation : elle ne modifie pas le sens d'un programme.

Voici la spécification de la classe `Boolean` :

```
package scala
abstract class Boolean {
  def && (x: => Boolean): Boolean
  def || (x: => Boolean): Boolean
  def !                 : Boolean

  def == (x: Boolean)   : Boolean
  def != (x: Boolean)   : Boolean
  def < (x: Boolean)    : Boolean
  def > (x: Boolean)    : Boolean
  def <= (x: Boolean)   : Boolean
  def >= (x: Boolean)   : Boolean
}
```

Les booléenns peuvent être définis uniquement à l'aide de classes et d'objets, sans aucune référence à un type prédéfini de booléens ou d'entiers. Nous donnons ci-dessous une implémentation possible, qui n'est pas celle de la bibliothèque standard de Scala car, nous l'avons déjà évoqué, celle-ci utilise des booléens prédéfinis pour des raisons d'efficacité.

```
package scala
abstract class Boolean {
  def ifThenElse(thenpart: => Boolean, elsepart: => Boolean)

  def && (x: => Boolean): Boolean = ifThenElse(x, false)
  def || (x: => Boolean): Boolean = ifThenElse(true, x)
  def !                 : Boolean = ifThenElse(false, true)

  def == (x: Boolean) : Boolean = ifThenElse(x, x.!)
  def != (x: Boolean) : Boolean = ifThenElse(x.!, x)
  def < (x: Boolean)  : Boolean = ifThenElse(false, x)
  def > (x: Boolean)  : Boolean = ifThenElse(x.!, false)
  def <= (x: Boolean) : Boolean = ifThenElse(x, true)
  def >= (x: Boolean) : Boolean = ifThenElse(true, x.!)
}

case object True extends Boolean {
  def ifThenElse(t: => Boolean, e: => Boolean) = t
}
case object False extends Boolean {
  def ifThenElse(t: => Boolean, e: => Boolean) = e
}
```

Voici une spécification partielle de la classe `Int` :

```
package scala
abstract class Int extends AnyVal {
  def toLong: Long
  def toFloat: Float
  def toDouble: Double

  def + (that: Double): Double
  def + (that: Float): Float
  def + (that: Long): Long
  def + (that: Int): Int            // Idem pour -, *, /, %

  def << (cnt: Int): Int            // Idem pour >>, >>>

  def & (that: Long): Long
  def & (that: Int): Int            // Idem pour |, ^

  def == (that: Double): Boolean
  def == (that: Float): Boolean
  def == (that: Long): Boolean      // Idem pour !=, <, >, <=, >=
}
```

En principe, la classe `Int` pourrait aussi être implémentée uniquement à l'aide d'objets et des classes, sans utiliser de type entier prédéfini. Pour voir comment y parvenir, considérons un problème un peu plus simple : l'implémentation d'un type `Nat` pour représenter les entiers naturels (c'est-à-dire positifs ou nuls). Voici la définition d'une classe `Nat` abstraite :

```
abstract class Nat {
  def isZero: Boolean
  def predecessor: Nat
  def successor: Nat
  def + (that: Nat): Nat
  def - (that: Nat): Nat
}
```

Pour implémenter les opérations de la classe `Nat`, nous définirons un sous-objet `Zero` et une sous-classe `Succ` (pour successeur). Chaque nombre *N* est représenté par *N* applications du constructeur `Succ` à `Zero` :

```
new Succ( ... new Succ (Zero) ... )     // N fois
```

L'implémentation de l'objet `Zero` est triviale :

```
object Zero extends Nat {
  def isZero: Boolean = true
  def predecessor: Nat = error("negative number")
  def successor: Nat = new Succ(Zero)
  def + (that: Nat): Nat = that
  def - (that: Nat): Nat = if (that.isZero) Zero
                           else error("negative number")
}
```

L'implémentation des fonctions `predecessor` et de soustraction sur `Zero` lancent une exception `Error` qui met fin au programme et affiche le message indiqué. Voici l'implémentation de la classe `Succ` :

```
class Succ(x: Nat) extends Nat {
  def isZero: Boolean = false
  def predecessor: Nat = x
  def successor: Nat = new Succ(this)
  def + (that: Nat): Nat = x + that.successor
  def - (that: Nat): Nat = if (that.isZero) this
                           else x - that.predecessor
}
```

Notez l'implémentation de la méthode `successor`. Pour créer le successeur d'un nombre, nous devons passer l'objet lui-même (désigné par le mot-clé **this**) comme paramètre du constructeur de `Succ`.

Les implémentations de `+` et `-` contiennent toutes les deux un appel récursif dont le récepteur est le paramètre qui a été passé au constructeur. Cette récursivité se termine lorsque le récepteur est l'objet `Zero` (ce qui arrivera nécessairement à cause de la façon dont les nombres sont formés).

Exercice 6–0–3 : Écrivez une implémentation `Integer` permettant de représenter les nombres entiers. Cette implémentation doit fournir toutes les opérations de la classe `Nat` et lui ajouter les deux méthodes suivantes :

```
def isPositive: Boolean
def negate: Integer
```

La première méthode doit renvoyer **true** si le nombre est positif. La seconde méthode doit renvoyer l'opposé du nombre. Vous n'avez pas le droit d'utiliser les classes numériques standard de Scala dans cette implémentation (il y a deux façons d'implémenter `Integer` : on peut partir de l'implémentation actuelle de `Nat` et représenter un entier comme un nombre naturel avec un signe ou l'on peut généraliser l'implémentation de `Nat` en `Integer`, en utilisant les trois sous-classes `Zero` pour 0, `Succ` pour les nombres positifs et `Pred` pour les nombres négatifs).

Éléments du langage introduits dans ce chapitre

Types :

```
Type    = ... | ident
```

Les types peuvent maintenant être des identificateurs quelconques, qui représentent des classes.

Expressions :

```
Expr    = ... | Expr '.' ident | 'new' Expr | 'this'
```

Une expression peut maintenant être une création d'objet, une sélection $E.m$ du membre m d'une expression E évaluée comme un objet ou le mot-clé **this**.

Définitions et déclarations :

```
Def           = DefFonction | DefValeur | DefClasse | DefTrait | DefObjet
DefClasse     = ['abstract'] 'class' ident ['(' [Paramètres] ')']
                ['extends' Expr] ['{' {DefModèle} '}']
DefTrait      = 'trait' ident ['extends' Expr] ['{' {DefModèle} '}']
DefObjet      = 'object' ident ['extends' Expr] ['{' {DefModèle} '}']
DefModèle     = [Modificateur] (Def | Dcl)
DefObjet      = [Modificateur] Def
Modificateur  =  'private' | 'override'
Dcl           = DclFonction | DclVal
DclFonction   = 'def' ident {'(' [Paramètres] ')'} ':' Type
DclVal        = 'val' ident ':' Type
```

Une définition peut maintenant être une définition de classe, de trait ou d'objet, comme

```
class C(params) extends B { defs }
trait T extends B { defs }
object O extends B { defs }
```

Les définitions **def** dans une classe, un trait ou un objet peuvent être précédées des modificateurs **private** ou **override**.

Les classes abstraites et les traits peuvent également contenir des déclarations. Tous les deux introduisent des fonctions ou des valeurs différées avec leurs types, mais sans fournir d'implémentation. Avant de pouvoir créer des objets d'une classe abstraite ou d'un trait, il faut implémenter leurs membres différés dans des sous-classes.

Chapitre 7. Case Classes et Pattern Matching

Supposons que nous voulions écrire un interpréteur d'expressions arithmétiques. Pour commencer simplement, nous nous limiterons aux nombres et à l'opération d'addition. Ces expressions peuvent être représentées par une hiérarchie de classes dont la racine est la classe `Expr`, avec ses deux sous-classes `Number` et `Sum`. Une expression comme `1 + (3 + 7)` peut alors être représentée par

```
new Sum(new Number(1), new Sum(new Number(3), new Number(7)))
```

Pour évaluer une expression comme celle-ci, il faut connaître sa forme (`Sum` ou `Number`) et accéder à ses composants. L'implémentation suivante fournit les méthodes nécessaires :

```
abstract class Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}

class Number(n: Int) extends Expr {
  def isNumber: Boolean = true
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = error("Number.leftOp")
  def rightOp: Expr = error("Number.rightOp")
}

class Sum(e1: Expr, e2: Expr) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = error("Sum.numValue")
  def leftOp: Expr = e1
  def rightOp: Expr = e2
}
```

Grâce à ces méthodes de classification et d'accès, l'écriture d'une fonction d'évalution est relativement simple :

```
def eval(e: Expr): Int = {
  if (e.isNumber) e.numValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else error("unrecognized expression kind")
}
```

Cependant, définir toutes ces méthodes dans les classes `Sum` et `Number` est assez pénible. En outre, le problème empirera lorsque nous voudrons ajouter d'autres formes d'expressions. Si, par exemple, nous voulons ajouter la multiplication, nous devrons non seulement implémenter une classe `Prod` avec toutes les méthodes de classification et d'accès, mais également introduire une nouvelle méthode abstraite `isProduct` dans la classe `Expr` et l'implémenter dans les sous-classes `Number`, `Sum` et `Product`. Devoir modifier un code existant lorsqu'un système évolue est toujours un problème car cela introduit des soucis de versions et de maintenance.

La promesse de la programmation orientée objet est que ce genre de modifications devraient être inutiles car elles peuvent être évitée en réutilisant par héritage le code existant et non modifié. Évidemment, notre problème se résoud en utilisant une décomposition mieux orientée objet. Le principe est de faire de l'opération `eval` de "haut niveau" une méthode

de chaque classe d'expression au lieu de l'implémenter comme une fonction extérieure à la hiérarchie des classes comme nous l'avons fait plus haut. `eval` étant désormais un membre de chaque expression, toutes les méthodes de classification et d'accès deviennent inutiles et l'implémentation s'en trouve considérablement simplifiée :

```
abstract class Expr {
  def eval: Int
}

class Number(n: Int) extends Expr {
  def eval: Int = n
}

class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

En outre, l'ajout d'une nouvelle classe `Prod` n'implique plus de modifier le code existant :

```
class Prod(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval * e2.eval
}
```

La conclusion que nous pouvons tirer de cet exemple est qu'une décomposition orientée objet convient parfaitement à la construction des systèmes susceptibles d'être étendus par de nouveaux types de données. Mais nous pourrions aussi étendre notre exemple d'une autre façon, en ajoutant de nouvelles opérations sur les expressions. Nous pourrions, par exemple, vouloir ajouter une opération qui affiche proprement un arbre d'expressions sur la sortie standard.

Si nous avons défini toutes les méthodes de classification et d'accès, cette opération peut aisément être implémentée comme une fonction externe :

```
def print(e: Expr) {
  if (e.isNumber) Console.print(e.numValue)
  else if (e.isSum) {
    Console.print("(")
    print(e.leftOp)
    Console.print("+")
    print(e.rightOp)
    Console.print(")")
  }
  else error("unrecognized expression kind")
}
```

Si, par contre, nous avons choisi une décomposition orientée objet des expressions, nous devrons ajouter une nouvelle procédure print à chaque classe :

```
abstract class Expr {
  def eval: Int
  def print
}

class Number(n: Int) extends Expr {
  def eval: Int = n
  def print { Console.print(n) }
}

class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
  def print {
    Console.print("(")
    print(e1)
    Console.print("+")
```

```
    print(e2)
    Console.print(")")
  }
}
```

Une décomposition orientée objet classique nécessite donc la modification de toutes les classes lorsque l'on ajoute de nouvelles opérations à un système.

Comme autre moyen d'étendre l'interpréteur, pensez à la simplification des expressions. Nous pourrions vouloir écrire, par exemple, une fonction qui réécrit les expressions de la forme `a * b + a * c` en `a * (b + c)`. Ce traitement nécessite d'inspecter plusieurs noeuds de l'arbre d'expresson en même temps et ne peut donc pas être implémenté par une méthode dans chaque classe, sauf si cette méthode inspecte également les autres noeuds. Nous sommes donc obligés ici d'avoir des méthodes de classification et d'accès, ce qui semble nous ramener au point de départ, avec tous les problèmes d'extension que nous avons déjà évoqués.

Si l'on examine ce problème de plus près, on observe que le seul but des méthodes de classification et d'accès consiste à *inverser* le processus de construction des données. Elles nous permettent, d'abord, de déterminer quelle est la sous-classe utilisée et, ensuite, de savoir quels étaient les paramètres du constructeur. Cette situation étant relativement fréquente, Scals dispose d'un mécanisme permettant de l'automatiquer : les case classes.

Case classes et case objets

Les *case classes* et les *case objets* sont définis comme des classes ou des objets normaux, sauf que leur définition est préfixée du modificateur **case**. Les définitions suivantes, par exemple :

```
abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

créent les case classes Number et Sum. Le modificateur case devant une définition de classes ou d'objet a les effets suivants :

1. Les case classes fournissent implicitement une méthode constructeur de même nom que la classe. Dans notre exemple, les deux méthodes suivantes sont donc implicitement créées :

   ```
   def Number(n: Int) = new Number(n)
   def Sum(e1: Expr, e2: Expr) = new Sum(e1, e2)
   ```

   Nous pouvons donc désormais construire des arbres d'expressions de façon un peu plus concise :

   ```
   Sum(Sum(Number(1), Number(2)), Number(3))
   ```

2. Les case classes et les case objets fournissent implicitement les méthodes `toString`, `equals` et `hashCode`, qui redéfinissent les méthodes de même nom dans la classe `AnyRef`. L'implémentation de ces méthodes prend en compte la structure d'une instance de la classe. La méthode `toString` représente un arbre d'expressions comme il a été construit :

   ```
   Sum(Sum(Number(1), Number(2)), Number(3))
   ```

   sera donc converti exactement en cette chaîne alors que l'implémentation par défaut de la classe `AnyRef` aurait renvoyé une chaîne formée du nom du constructeur le plus externe, `Sum`, suivi d'un nombre. La méthode `equals` considère que deux instances de la classe sont égales si elles ont été construites avec le même constructeur et les mêmes paramètres égaux deux à deux. La redéfinition de cette méthode affecte également l'implémentation de == et != puisque, en Scala, ces opérateurs sont implémentés en termes d'`equals` :

```
Sum(Number(1), Number(2)) == Sum(Number(1), Number(2))
```

renverra donc **true**. Si `Sum` ou `Number` n'étaient pas des case classes, cette expression aurait renvoyé **false** puisque l'implémentation standard de `equals` dans la classe `AnyRef` considère toujours que des objets créés par des appels différents sont également différents. La méthode `hashCode` suit le même principe que les deux autres. Elle calcule un code de hachage à partir du nom du constructeur de la case classe et des paramètres passés au constructeur au lieu d'utiliser l'adresse de l'objet, qui est le comportement de l'implémentation par défaut de `hashCode`.

3. Les case classes fournissent implicitement des méthodes d'accès permettant de récupérer les paramètres passés au constructeur. Dans notre exemple, `Number` disposerait donc d'une méthode d'accès renvoyant la valeur du paramètre `n` du constructeur :

```
def n: Int
```

alors que `Sum` aurait deux méthodes d'accès :

```
def e1: Expr
def e2: Expr
```

Pour une valeur *s* de type `Sum`, nous pouvons donc écrire *s*`.e1` pour accéder à l'opérande gauche. En revanche, pour une valeur *e* de type `Expr`, le terme *e*`.e1` serait illégal car `e1` est définie dans `Sum` – ce n'est pas un membre de la classe de base `Expr`. Comment alors déterminer le constructeur et accéder aux paramètres du constructeur pour les valeurs dont le type statique est `Expr` ? La réponse est donnée par la quatrième et dernière particularité des case classes.

4. Les case classes permettent de construire des *motifs* permettant de désigner leur constructeur.

Pattern Matching

Le pattern matching est une généralisation de l'instruction `switch` de C ou Java aux hiérarchies de classes. À la place de l'instruction `switch` existe une méthode `match` qui est définie dans la classe racine `Any` de Scala et qui est donc disponible pour tous les objets. Cette méthode prend en paramètre plusieurs cas. Voici, par exemple, une implémentation de `eval` qui utilise le pattern matching :

```
def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(l, r) => eval(l) + eval(r)
}
```

Cet exemple utilise deux cas. Chacun d'eux associe un motif à une expression. Les motifs sont comparés avec les valeurs du sélecteur `e`. Le premier motif ici, `Number(n)` capture toutes les valeurs de la forme `Number(v)`, où `v` est une valeur quelconque. En ce cas, la variable `n` du motif est liée à la valeur `v`. De même, le motif `Sum(l, r)` capture toutes les valeurs du sélecteur de la forme `Sum(v1, v2)` et lie les variables du motif `l` et `r` respectivement à `v1` et à `v2`.

En règle générale, les motifs sont construits à partir de :

- Constructeurs de case classes, comme `Number` et `Sum`, dont les paramètres sont aussi des motifs.
- Variables, comme `n`, `e1`, `e2`.
- Motif "joker" `_`
- Littéraux, comme **true** ou `"abc"`.
- Identificateurs de constantes, comme `MAXINT` ou `EmptySet`.

Les variables de motif commencent toujours par une minuscule afin de pouvoir être distinguées des identificateurs de constantes, qui commencent par une majuscule. Un nom de variable ne peut apparaître qu'une seule fois dans un même motif. `Sum(x, x)` est donc illégal car la variable `x` apparaît deux fois.

Signification du pattern matching. Une expression de pattern matching

```
e match {case p1 => e1 ... case pn => en }
```

compare les motifs `p1`, …, `pn` à la valeur du sélecteur `e` dans l'ordre de leur apparition.

- Un motif constructeur C(p1, …, pn) capture toutes les valeurs de type C (ou d'un sous-type de C) qui ont été construites avec des paramètres correspondant aux motifs p1, …, pn.
- Une variable de motif x capture n'importe quelle valeur et lie ce nom à la valeur.
- Le motif joker _ capture n'importe quelle valeur mais ne lie aucun nom à cette valeur.
- Un motif constant C capture une valeur égale (au sens de ==) à C.

L'expression pattern matching est réécrite par la partie droite du premier cas dont le motif correspond à la valeur du sélecteur. Les références aux variables de motif sont remplacées par les paramètres correspondants du constructeur. Si aucun motif n'a pu capturer de valeur, l'expression s'interrompt en produisant une exception `MatchError`.

Exemple 7–2–1 : Notre modèle de sustitution de l'évaluation d'un programme s'étend assez naturellement au pattern matching. Voici, par exemple, comment est réécrit eval lorsqu'elle est appliquée à une expression simple :

```
eval(Sum(Number(1), Number(2)))
```

donne, par réécriture de l'application :

```
Sum(Number(1), Number(2)) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

Par réécriture du pattern matching, on obtient :

```
eval(Number(1)) + eval(Number(2))
```

En réécrivant le premier appel, on a :

```
Number(1) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
} + eval(Number(2))
```

Par réécriture du pattern matching, on obtient :

```
1 + eval(Number(2))
```

et, en continuant :

```
1 + 2 -> 3
```

Pattern Matching et méthodes. Dans l'exemple précédent, nous nous sommes servis du pattern matching dans une fonction qui était définie en dehors de la hiérarchie des classes qu'elle compare. Il est évidemment possible de définir une méthode de pattern matching dans la hiérarchie elle-même. Nous pourrions, par exemple, avoir défini evel comme une méthode de lal classe de base Expr et utiliser quand même le pattern matching dans son implémentation :

```
abstract class Expr {
  def eval: Int = this match {
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
  }
}
```

Exercice 7-2-2 : Considérez les définitions suivante d'une arborescence d'entiers. Elle peuvent être vues comme une représentation alternative de `IntSet` :

```
abstract class IntTree
case object EmptyTree extends IntTree
case class Node(elem: Int, left: IntTree, right: IntTree) extends IntTree
```

Complétez les implémentations suivantes des méthodes `contains` et `insert` pour les `IntTree`.

```
def contains(t: IntTree, v: Int): Boolean = t match { ...
  ...
}
def insert(t: IntTree, v: Int): IntTree = t match { ...
  ...
}
```

Fonctions anonymes de pattern matching. Jusqu'à présent, les expressions `case` apparaissaient toujours au sein d'une opération `match`. Mais il est également possible d'utiliser les expressions `case` par elles-mêmes. Un bloc d'expressions case comme :

```
{case P1 => E1 ... case Pn => En }
```

est considéré comme une fonction qui compare ses paramètres aux motifs `P1, ..., Pn` et produit un résultat `E1`, ou … ou `En` (si aucun motif ne correspond, la fonction lèvera une exception `MatchError`). En d'autres termes, l'expression ci-dessus est considérée comme une forme raccourcie de la fonction anonyme :

```
(x => x match { case P1 => E1 ... case Pn => En })
```

où `x` est une nouvelle variable qui n'est utilisée nulle part ailleurs dans l'expression.

Chapitre 8. Types et méthodes génériques

En Scala, les classes peuvent avoir des paramètres de type. Supposons que nous voulions créer un type de données pour représenter les piles d'entiers, avec les méthodes `push`, `top`, `pop` et `isEmpty`. Voici la hiérarchie de classes correspondante :

```
abstract class IntStack {
  def push(x: Int): IntStack = new IntNonEmptyStack(x, this)
  def isEmpty: Boolean
  def top: Int
  def pop: IntStack
}

class IntEmptyStack extends IntStack {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}

class IntNonEmptyStack(elem: Int, rest: IntStack) extends IntStack {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

Il serait bien sûr tout aussi judicieux de définir une pile pour les chaînes. Pour ce faire, nous pourrions reprendre `InStack`, la renommer en `StringStack` et remplacer toutes les occurrences de `Int` par `String`.

Une meilleure approche, qui n'implique pas de dupliquer le code, consiste à paramétrer les définitions des piles avec le type de leurs éléments. Cette paramétrisation nous permet de passer d'une représentation spécifique d'un problème à une représentation plus générale. Jusqu'à maintenant, nous n'avions paramétré que des valeurs, mais nous pouvons faire de même pour les types. Pour disposer d'une version *générique* de `Stack`, il suffit donc de lui ajouter un paramètre de type :

```
abstract class Stack[A] {
  def push(x: A): Stack[A] = new NonEmptyStack[A](x,this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}

class EmptyStack[A] extends Stack[A] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}

class NonEmptyStack[A](elem: A, rest: Stack[A]) extends Stack[A] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

Dans ces définitions, *A* est le *paramètre de type* de la classe `Stack` et de ses sous-classes. Le nom de ce paramètre n'a pas d'importance ; il doit être placé entre crochets afin de le distinguer des paramètres classiques. Voici un exemple d'utilisation de ces classes génériques :

```
val x = new EmptyStack[Int]
val y = x.push(1).push(2)
println(y.pop.top)
```

La première ligne crée une pile vide de `Int`. Notez que le paramètre de type effectif `[Int]` remplace le paramètre de type formel `[A]`.

Les méthodes peuvent également être paramétrées par des types. Voici, par exemple, une méthode générique qui teste si une pile est préfixe d'une autre :

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[A](p.pop, s.pop)
}
```

Les paramètres de la méthode sont *polymorphes* – c'est la raison pour laquelle les méthodes génériques sont également appelées *méthodes polymorphes*. Ce terme est un mot grec qui signifie "avoir plusieurs formes". Pour appliquer une méthode polymorphique comme `isPrefix`, vous devez lui fournir les paramètres de type ainsi que les paramètres valeurs : ,

```
val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix[String](s1, s2))
```

Inférence de type locale. Passer constamment des paramètres de type comme `[Int]` ou `[String]` peut devenir assez lourd lorsque l'on utilise beaucoup les méthodes génériques. Assez souvent, l'indication d'un paramètre de type est redondante car le paramètre correct peut également être déterminé en inspectant les paramètres valeurs de la fonction ou le type de son résultat. Si l'on prend comme exemple l'expression `isPrefix[String](s1, s2)`, on sait que ses paramètres valeurs sont tous les deux de type `Stack[String]` et l'on peut donc déduire que le paramètre de type doit être `String`. Scala dispose d'un inférenceur de type assez puissant permettant d'omettre les paramètres de types des fonctions polymorphes et des constructeurs dans des situations comme celle-ci. Dans l'exemple ci-dessus, nous aurions donc simplement pu écrire `isPrefix(s1, s2)` et le paramètre de type manquant, `[String]`, aurait été inséré par l'inférenceur de type.

Bornes des paramètres de type

Maintenant que nous savons créer des classes génériques, il est naturel de vouloir généraliser certaines de nos classes précédentes. La classe IntSet, par exemple, pourrait être étendue aux ensembles d'éléments de types quelconques :

```
abstract class Set[A] {
  def incl(x: A): Set[A]
  def contains(x: A): Boolean
}
```

Cependant, si nous voulons toujours les implémenter comme des arbres de recherche binaires, nous allons avoir un problème. En effet, les méthodes contains et incl comparent toutes les deux les éléments à l'aide des méthodes < et >. Ceci ne posait pas de problème avec Intset car le type Int dispose de ces deux méthodes mais nous ne pouvons pas le garantir pour un paramètre de type a quelconque. L'implémentation précédente de contains produira donc une erreur de compilation.

```
def contains(x: Int): Boolean =
  if (x < elem) left contains x
        ^ < not a member of type A.
```

Une façon de résoudre ce problème consiste à restreindre les types autorisés à se substituer au type `A` à ceux qui contenant les méthodes < et > avec les signatures adéquates. La bibliothèque standard de Scala contient un trait `Ordered[A]` qui représente les valeurs comparables (avec < et >) à des valeurs de type `A`. Ce trait est défini de la façon suivante :

```
/** Classe représentant les données totalement ordonnées */
trait Ordered[A] {

  /** Résultat de la comparaison de 'this' avec 'that'.
   *  renvoie 'x' où :
   *  x < 0 ssi this < that
   *  x == 0 ssi this == that
   *  x > 0 ssi this > that
   */
  def compare(that: A): Int

  def <  (that: A) : Boolean = (this compare that) <  0
  def >  (that: A) : Boolean = (this compare that) >  0
  def <= (that: A) : Boolean = (this compare that) <= 0
  def >= (that: A) : Boolean = (this compare that) >= 0
  def compareTo(that: A): Int = compare(that)
}
```

Nous pouvons imposer la compatibilité d'un type en demandant qu'il soit un sous-type de `Ordered`. Pour ce faire, nous fixons une borne supérieure au paramètre de type de `Set` :

```
trait Set[A <: Ordered[A]] {
  def incl(x: A): Set[A]
  def contains(x: A): Boolean
}
```

La déclaration `A <: Ordered[A]` précise que `A` est un paramètre de type et qu'il doit être un sous-type de `Ordered[A]`, c'est-à-dire que ses valeurs doivent être comparables entre elles.

Grâce à cette restriction, nous pouvons maintenant implémenter le reste de notre ensemble générique, comme nous l'avions fait pour `IntSet` :

```
class EmptySet[A <: Ordered[A]] extends Set[A] {
  def contains(x: A): Boolean = false
  def incl(x: A): Set[A] = new NonEmptySet(x, new EmptySet[A], new EmptySet[A])
}

class NonEmptySet[A <: Ordered[A]] (elem: A, left: Set[A], right: Set[A])
        extends Set[A] {
  def contains(x: A): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: A): Set[A] =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

Notez que nous n'avons pas indiqué le paramètre de type lors des créations d'objets **new** `NonEmptySet(...)`. Tout comme pour les méthodes polymorphes, les paramètres de types absents lors des appels aux constructeurs peuvent être inférés à partir des paramètres valeurs et/ou du type de résultat attendu.

Voici un exemple utilisant l'abstraction des ensembles génériques. Commençons par créer une sous-classe de `Ordered` :

```
case class Num(value: Double) extends Ordered[Num] {
  def compare(that: Num): Int =
    if (this.value < that.value) -1
    else if (this.value > that.value) 1
    else 0
}
```

Puis :

```
val s = new EmptySet[Num].incl(Num(1.0)).incl(Num(2.0))
s.contains(Num(1.5))
```

Tout va bien puisque le type `Num` implémente le trait `Ordered[Num]`. Par contre, le code suivant est erroné :

```
val s = new EmptySet[java.io.File]
                     ^ java.io.File does not conform to type
                       parameter bound Ordered[java.io.File].
```

Un problème avec les bornes des paramètres de types est qu'il faut être prévoyant : si nous n'avions pas déclaré `Num` comme sous-classe de `Ordered`, nous n'aurions pas pu utiliser les éléments `Num` dans les ensembles. Pour la même raison, les types hérités de Java – comme `Int`, `Double` ou `String` – n'étant pas des sous-classes de `Ordered`, leurs valeurs ne peuvent pas être utilisés comme éléments d'un ensemble.

Une conception plus souple, qui permet d'utiliser des éléments de ces types, consiste à utiliser des *bornes vues* à la place des bornes de vrais types que nous avons vues jusqu'à maintenant. La seule modification de code dans l'exemple précédent consiste à changer les paramètres de généricité :

```
trait Set[A <% Ordered[A]]
  ...

class EmptySet[A <% Ordered[A]]
  ...

class NonEmptySet[A <% Ordered[A]]
  ...
```

Les bornes vues `<%` sont plus faibles que les bornes réélles `<:`. Une clause de borne vue `[A <% T]` précise simplement que le type `A` doit être convertible dans le type de la borne `T` à l'aide d'une conversion implicite.

La bibliothèque standard de Scala prédéfinit des conversions implicites pour un certain nombre de types, notamment les types primitifs et String. Par conséquent, notre nouvelle abstraction des ensembles peut désormais également être instanciées avec ces types. Vous trouverez plus d'informations sur les conversions implicites au chapitre 15.

Annotation de variance

La combinaison des paramètres de type et du sous-typage pose quelques questions intéressantes. Est-ce que `Stack[String]` doit être un sous-type de `Stack[AnyRef]`, par exemple ? Intuitivement, ceci semble être le cas puisqu'une pile de `String` est un cas particulier d'une pile de `AnyRef`. De façon générale, si `T` est un sous-type de `S`, alors `Stack[T]` devrait être un sous-type de `Stack[S]`. Cette propriété est appelé sous-typage *covariant*.

En Scala, les types utilisent, par défaut, un sous-typage covariant. Avec notre classe `Stack` telle qu'elle est définie, les piles de différents types d'éléments ne seront jamais dans une relation de sous-typage, mais nous pouvons imposer un sous-typage covariant des piles en modifiant la première ligne de la définition de la classe :

```
class Stack[+A] {
```

Un paramètre formel de type préfixé par `+` indique que le sous-typage sera covariant pour ce paramètre. Il existe également le préfixe `-` qui indique un sous-type contra-variant : si `Stack` avait été définie par class `Stack[-A] ...`, `Stack[S]` serait un sous-type de `Stack[T]` si `T` est un sous-type de `S` (ce qui, ici, serait assez surprenant !).

Dans un monde purement fonctionnel, tous les types pourraient être covariants. Cependant, la situation change lorsque l'on introduit des données modifiables. Par exemple, les tableaux Java ou .NET sont représentés en Scala par la classe générique `Array` dont voici une définition partielle :

```
class Array[A] {
  def apply(index: Int): A
  def update(index: Int, elem: A)
}
```

Cette classe définit la façon dont sont vus les tableaux Scala par les programmes clients. Dans la mesure du possible, le compilateur fera correspondre cette abstraction aux tableaux sous-jacents du système hôte.

En Java, les tableaux sont évidemment covariants : si T et S sont des types références et que T est sous-type de S, alors Array[T] est également un sous-type de Array[S]. Ceci semble naturel mais peut poser des problèmes nécessitant des vérifications spéciales lors de l'exécution :

```
val x = new Array[String](1)
val y: Array[Any] = x
y(0) = new Rational(1, 2) // sucre syntaxique pour y.update(0, new Rational(1, 2))
```

La première ligne crée un tableau de chaînes. La seconde lie ce tableau à une variable `y` de type `Array[Any]`, ce qui est correct puique les tableaux sont covariants : `Array[String]` est donc un sous-type de `Array[Any]`. Enfin, la dernière ligne stocke un nombre rationnel dans le tableau `y`, ce qui également permis puisque le type `Rational` est un sous-type de `Any`, le type des éléments de `y`. Au final, nous avons donc stocké un nombre rationnel dans un tableau de chaînes, ce qui viole clairement notre notion des types.

Java résoud ce problème en introduisant un test à l'exécution dans la troisième ligne, afin de vérifier que l'élément stocké est compatible avec celui des éléments du tableau tel qu'il a été créé. Nous avons vu dans l'exemple que ce type des éléments n'est pas nécessairement le type statique du tableau mis à jour. Si le test échoue, l'exception `ArrayStoreException` est levée.

Scala, quant à lui, résoud ce problème de façon statique, en interdisant la seconde ligne lors de la compilation puisque les tableaux Scala ont un sous-typage non variant. Ceci pose donc la question de savoir comment un compilateur Scala vérifie que les annotations de variance sont correctes. Si nous avions simplement déclaré les tableaux comme covariants, comment aurait-on pu détecter ce problème ?

Scala utilise une approximation prudente pour vérifier la cohérence des annotations de variance. Un paramètre de type covariant d'une classe ne peut apparaître qu'aux positions covariantes de la classe – les types des valeurs dans la classe, les types des résultats des méthodes de la classe et les paramètres de type des autres types covariants, notamment. Les types des paramètres formels des méthodes ne sont pas des positions covariantes. La définition de classe suivante est donc incorrecte :

```
class Array[+A] {
  def apply(index: Int): A
  def update(index: Int, elem: A)
                              ^ covariant type parameter A appears in
                                contravariant position.
}
```

Pour l'instant, tout va bien. Intuitivement, le compilateur a bien fait de rejeter la procédure `update` dans une classe covariante car cette méthode peut modifier l'état et donc perturber la cohérence du sous-typage covariant.

Cependant, certaines méthodes ne modifient pas l'état et ont un paramètre de type qui apparaîtra donc comme contra-variant – c'est le cas de la méthode `push` de la classe `Stack`, par exemple. Là aussi, le compilateur Scala rejettera la définition de cette méthode pour des piles co-variantes :

```
class Stack[+A] {
  def push(x: A): Stack[A] =
              ^ covariant type parameter A appears in contravariant position.
```

C'est dommage car, à la différence des tableaux, les piles sont des données fonctionnelles pures et elles devraient donc autoriser un sous-typage co-variant. Il existe toutefois un moyen de résoudre ce problème à l'aide d'une méthode polymorphe avec une borne inférieure pour le paramètre de type.

Bornes inférieures

Nous venons de voir les bornes supérieures pour les paramètres de type : dans une clause comme `T <: U`, le paramètre type `T` est limité aux sous-types de `U`. Il existe également des bornes inférieures : dans la clause `T >: S`, le paramètre type `T` est limité aux *supertypes* de `S` (il est également possible de combiner les bornes inférieures et supérieures, avec une clause comme `T >: S <: U`).

Les bornes inférieures vont nous permettre de généraliser la méthode `push` de la classe `Stack` :

```
class Stack[+A] {
def push[B >: A](x: B): Stack[B] = new NonEmptyStack(x, this)
```

Techniquement, ceci résoud notre problème de variance puisque, désormais, le paramètre de type `A` n'est plus un paramètre de type de la méthode `push` mais une borne inférieure d'un autre type paramètre, qui se trouve dans une position covariante. Le compilateur Scala acceptera donc cette nouvelle définition.

En réalité, nous avons non seulement résolu ce problème de variance mais nous avons également généralisé la définition de `push`. Auparavant, cette méthode ne pouvait empiler que des éléments dont les types étaient conformes au type d'élément de la pile. Désormais, nous pouvons également empiler des éléments qui sont d'un supertype de ce type et la pile renvoyée sera d'un type modifié en conséquence. Nous pouvons, par exemple, empiler un `AnyRef` sur une pile de `String` : le résultat sera une pile d'`AnyRef` et non une pile de `String`.

En résumé, n'hésitez pas à ajouter des annotations de variance à vos structures de données car cela produit des relations de sous-typage riches et naturelles. Le compilateur détectera les éventuels problèmes de cohérence. Lorsque l'approximation du compilateur est trop prudente, comme dans le cas de la méthode `push` de la classe `Stack`, elle suggère souvent une généralisation utile de la méthode concernée.

Types minimaux

Scala ne permet pas de paramétrer les objets avec des types. C'est la raison pour laquelle nous avions défini une classe `EmptyStack[A]` alors qu'une unique valeur suffisait pour représenter les piles vides de type quelconque. Pour les piles covariantes, vous pouvez toutefois utiliser l'un des idiomes suivants :

```
object EmptyStack extends Stack[Nothing] { ... }
```

Le type minimal `Nothing` ne contenant aucune valeur, le type `Stack[Nothing]` exprime le fait qu'une `EmptyStack` ne contient aucun élément. En outre, Nothing est un sous-type de tous les autres types ce qui signifie que, pour les piles covariantes, `Stack[Nothing]` est un sous-type de `Stack[T]` quel que soit `T`. Nous pouvons donc désormais utiliser un unique objet pile vide dans le code client :

```
val s = EmptyStack.push("abc").push(new AnyRef())
```

Analysons en détail l'affectation des types dans cette expression. L'objet `EmptyStack` est de type `Stack[Nothing]` et dispose donc d'une méthode

```
push[B >: Nothing](elem: B): Stack[B]
```

L'inférence de type locale déterminera que B doit être instancié par `String` lors de l'appel `EmptyStack.push("abc")`. Le type du résultat de cet appel est donc `Stack[String]` qui, à son tour, dispose d'une méthode

```
push[B >: String](elem: B): Stack[B]
```

La dernière partie de la définition de valeur est l'appel de cette méthode auquel on passe **new** `AnyRef()` en paramètre. L'inférence de type locale déterminera que le paramètre de type devrait cette fois-ci être instancié par `AnyRef` et que le résultat sera donc de type `Stack[AnyRef]`. Le type de la valeur affectée à `s` est donc `Stack[AnyRef]`.

Outre `Nothing`, qui est un sous-type de tous les autres types, il existe également le type `Null`, qui est un sous-type de `scala.AnyRef` et de toutes ses classes filles. En Scala, le littéral `null` est la seule valeur de ce type, ce qui la rend compatible avec tous les types référence, mais pas avec un type valeur comme `Int`.

Nous conclurons cette section par une définition complète de notre classe `Stack`. Les piles ont désormais un sous-typage covariant, la méthode `push` a été généralisée et la pile vide est représentée par un objet unique :

```
abstract class Stack[+A] {
  def push[B >: A](x: B): Stack[B] = new NonEmptyStack(x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}

object EmptyStack extends Stack[Nothing] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}

class NonEmptyStack[+A](elem: A, rest: Stack[A]) extends Stack[A] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

De nombreuses classes de la bibliothèque Scala sont génériques et nous allons maintenant présenter les deux familles les plus courantes : les tuples et les fonctions. Les listes, qui sont également très utilisées, seront présentées dans le prochain chapitre.

Tuples

Une fonction doit parfois pouvoir renvoyer plusieurs résultats. La fonction `divmod`, par exemple renvoie le quotient entier et le reste de la division de ses deux paramètres entiers. Nous pourrions évidemment définir une classe pour encapsuler ces deux résultats de `divmod` :

```
case class TwoInts(first: Int, second: Int)
def divmod(x: Int, y: Int): TwoInts = new TwoInts(x / y, x % y)
```

Mais devoir définir une nouvelle classe pour chaque paire de types de résultats possibles serait très lourd. En Scala, nous pouvons plutôt utiliser la classe générique `Tuple2`, qui est définie ainsi :

```
package scala
case class Tuple2[A, B](_1: A, _2: B)
```

Avec Tuple2, la méthode `divmod` peut être écrite de la façon suivante :

```
def divmod(x: Int, y: Int) = new Tuple2[Int, Int](x / y, x % y)
```

Comme d'habitude, les paramètres de type du constructeur peuvent être omis s'ils peuvent être déduits des paramètres passés à l'appel. Il existe également des classes tuples pour contenir n'importe quel autre nombre d'éléments (l'implémentation actuelle limite ce nombre à une valeur raisonnable).

Les tuples étant des case classes, les éléments des tuples peuvent être accédés de deux façons différentes. La première consiste à utiliser les noms des paramètres `_i` du constructeur, comme dans cet exemple :

```
val xy = divmod(x, y)
println("quotient: " + xy._1 + ", rest: " + xy._2)
```

La seconde consiste à utiliser le pattern matching sur les tuples, comme ici :

```
divmod(x, y) match {
  case Tuple2(n, d) =>
    println("quotient: " + n + ", rest: " + d)
}
```

Notez que les paramètres de types ne sont jamais utilisés dans les motifs – l'écriture `Tuple2[Int, Int](n, d)` est illégale.

Les tuples sont si pratiques que Scala définit une syntaxe spéciale pour faciliter leur utilisation. Pour un tuple de n éléments $x_1, \ldots x_n$, nous pouvons écrire `(`x_1`, … `x_n`)` à la place de `Tuple`n`(`x_1`, …, `x_n`)`. La syntaxe `(…)` fonctionnant à la fois pour les types et les motifs, l'exemple précédent peut donc être réécrit de la manière suivante :

```
def divmod(x: Int, y: Int): (Int, Int) = (x / y, x % y)

divmod(x, y) match {
  case (n, d) => println("quotient: " + n + ", rest: " + d)
}
```

Fonctions

Scala est un langage fonctionnel car les fonctions sont des valeurs de première classe. C'est également un langage orienté objet car chaque valeur est un objet. Il s'ensuit donc qu'en Scala les fonctions sont des objets. Une fonction de String vers Int, par exemple, sera représentée comme une instance du trait `Function1[String, Int]`. Le trait `Function1` est défini de la façon suivante dans la bibliothèque standard :

```
package scala trait Function1[-A, +B] {
   def apply(x: A): B
}
```

Outre `Function1`, il existe également des définitions pour les fonctions d'autres arités (l'implémentation actuelle autorise une limite raisonnable). Il y a donc une seule définition pour chaque nombre de paramètres possible. La syntaxe des types fonctions en Scala, `(`T_1`, …, `T_n`) => S` est simplement un raccourci pour le type paramétré `Functionn[`T_1`, …, `T_n`,S]`.

Scala utilise la même syntaxe `f(x)` pour l'application de fonction, que `f` soit une méthode ou un objet fonction. Ceci est possible en raison de la convention suivante : une application de fonction `f(x)` où `f` est un objet (et non une méthode) est un raccourci de `f.apply(x)`. La méthode `apply` d'un type fonction est donc insérée automatiquement lorsque cela est nécessaire.

C'est également la raison pour laquelle nous avons défini l'indexation des tableaux dans la section 8.2 par une méthode `apply`. Pour tout tableau `a`, l'opération `a(i)` est un raccourci de `a.apply(i)`.

Les fonctions sont un exemple d'utilisation d'un paramètre de type contra-variant. Étudiez, par exemple, le code suivant :

```
val f: (AnyRef => Int) = x => x.hashCode()
val g: (String => Int) = f
g("abc")
```

Il semble logique de lier la valeur `g` de type `String => Int` à `f` qui est de type `AnyRef => Int`. En effet, tout ce que l'on peut faire avec une fonction de type `String => Int` est de lui passer une chaîne afin d'obtenir un entier. Il en va de même pour la fonction `f` : si nous lui passons une chaîne (ou un autre objet), nous obtenons un entier. Ceci démontre que le sous-typage d'une fonction est contra-variant pour le type de son paramètre alors qu'il est covariant pour le type de son résultat. En résumé, `S => T` est un sous-type de `S => T'` si `S'` est un sous-type de `S` et `T` est un sous-type de `T'`.

Exemple 8–6–1 : Ce code :

```
val plus1:(Int => Int) = (x: Int) => x + 1
plus1(2)
```

est traduit dans le code suivant :

```
val plus1: Function1[Int, Int] = new Function1[Int, Int] {
  def apply(x: Int): Int = x + 1
}
plus1.apply(2)
```

Ici, la création d'objet **new** `Function1[Int, Int]{ … }` représente une instance d'une *classe anonyme*. Elle combine la création d'un nouvel objet `Function1` avec une implémentation de la méthode `apply` (qui est abstraite dans `Function1`). Nous aurions également pu utiliser une classe locale, mais cela aurait été plus lourd :

```
val plus1: Function1[Int, Int] = {
  class Local extends Function1[Int, Int] {
    def apply(x: Int): Int = x + 1 new Local: Function1[Int, Int]
  }
}
plus1.apply(2)
```

Chapitre 9. Listes

Pour un grand nombre de programmes Scala, les listes sont des structures de données importantes. Une liste contenant les éléments $x_1, \ldots, x_n$ est notée `List(x1, …, xn)`. Voici quelques exemples de listes :

```
val fruit = List("apples", "oranges", "pears")
val nums  = List(1, 2, 3, 4)
val diag3 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty = List()
```

Les listes ressemblent aux tableaux de C ou Java mais il y a trois différences importantes. Premièrement, les listes sont immutables, ce qui signifie que les éléments d'une liste ne peuvent pas être modifiés par affectation. Deuxièmement, les listes ont une structure récursive alors que les tableaux sont des structures plates. Troisièmement, les listes disposent généralement d'un ensemble d'opérations bien plus riche que les tableaux.

Utilisation des listes

Le type `List` Comme les tableaux, les listes sont *homogènes*, c'est-à-dire que leurs éléments sont tous de même type. Le type d'une liste d'éléments de type `T` est noté `List[T]` :

```
val fruit: List[String]     = List("apples", "oranges", "pears")
val nums : List[Int]        = List(1, 2, 3, 4)
val diag3: List[List[Int]]  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Int]        = List()
```

Constructeurs de listes Les listes sont construites à partir des deux constructeurs fondamentaux, `Nil` et `::` (prononcé "cons"). `Nil` représente une liste vide. L'opérateur `::` infixe exprime l'extension d'une liste : `x :: xs` représente une liste dont le premier élément est `x`, suivi des éléments de la liste `xs`. Les listes précédentes auraient donc pu également être définies de la façon suivante (en réalité, les définitions précédentes sont du sucre syntaxique masquant les définitions ci-dessous) :

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) :: (0 :: (1 :: (0 :: Nil))) ::
                 (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

L'opération `::` est associative à droite : `A :: B :: C` est interprété comme `A :: (B :: C)`. Par conséquent, nous pouvons éliminer les parenthèses des définitions précédentes et écrire, par exemple :

```
val nums  = 1 :: 2 :: 3 :: 4 :: Nil
```

Opérations de base sur les listes Toutes les opérations sur les listes peuvent s'exprimer à l'aide des trois opérations :

`head`	renvoie le premier élément d'une liste.
`tail`	renvoie la liste formée de tous les éléments sauf le premier.
`isEmpty`	renvoie **true** si et seulement si la liste est vide.

Ces opérations sont définies comme des méthodes sur les objets listes. On les invoque donc à l'aide de la notation pointée habituelle :

```
empty.isEmpty         // renvoie true
fruit.isEmpty         // renvoie false
fruit.head            // renvoie "apples"
fruit.tail.head       // renvoie "oranges"
diag3.head            // renvoie List(1, 0, 0)
```

Les méthodes `head` et `tail` ne sont définies que pour des listes non vides. Si elles sont appelées sur une liste vide, elles lancent une exception. Comme exemple de traitement des listes, étudions le tri croissant d'une liste de nombres. Un moyen simple d'y parvenir consiste à utiliser un tri par insertion, dont le principe est le suivant : pour trier une liste non vide dont le premier élément est `x` et le reste `xs`, on trie `xs` et on insère l'élément `x` à la bonne position dans le résultat. Le tri d'une liste vide donne une liste vide. Exprimé en Scala, cela donne :

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))
```

Exercice 9–1–1 : Proposez une implémentation de la fonction `insert`

Motifs de listes En fait, :: est défini comme une case classe dans la bibliothèque standard de Scala. Il est donc possible de décomposer les listes par pattern matching en utilisant des motifs composés à partir des constructeurs `Nil` et ::. La méthode `isort` pourrait donc être écrite de la façon suivante :

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case x :: xs1 => insert(x, isort(xs1))
}
```

où :

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List() => List(x)
  case y :: ys => if (x <= y) x :: xs else y :: insert(x, ys)
}
```

Définition de la classe `List` : méthodes du premier ordre (I)

Les listes ne sont pas prédéfinies en Scala : elles sont définies par une classe abstraite `List` qui est fournie avec deux sous-classes :: et `Nil`. Dans cette section, nous passerons en revue la classe `List`.

```
package scala
abstract class List[+A] {
```

`List` étant une classe abstraite, nous ne pouvons pas définir les éléments en appelant le constructeur `List` (en faisant **new** `List`). La classe a un paramètre de type `A` covariant, ce qui signifie que `List[S] <: List[T]` pour tous les types `S` et `T` tels que `S <: T`. Cette classe se trouve dans le paquetage `scala`, qui contient les principales classes standard de Scala. `List` définit un certains nombres de méthodes que nous décrirons ci-dessous.

Décomposition des listes. Les trois méthodes de base sont `isEmpty`, `head` et `tail`. Leur implémentation avec le pattern matching est triviale :

```
def isEmpty: Boolean = this match {
  case Nil => true case x :: xs => false
}
def head: A = this match {
  case Nil => error("Nil.head")
  case x :: xs => x
}
def tail: List[A] = this match {
  case Nil => error("Nil.tail")
  case x :: xs => xs
}
```

La fonction `length` calcule la longueur d'une liste :

```
def length: Int = this match {
  case Nil => 0
  case x :: xs => 1 + xs.length
}
```

Exercice 9–2–1 : Écrivez une version récursive terminale de `length`.

Les deux fonctions suivantes sont les compléments de `head` et `tail` :

```
def last: A
def init: List[A]
```

`xs.last` renvoie le dernier élément de la liste `xs` tandis que `xs.init` renvoie tous les éléments de `xs` sauf le dernier. Ces deux fonctions doivent parcourir toute le liste et sont donc moins efficaces que leurs homologues `head` et `tail`. Voici l'implémentation de `last` :

```
def last: A = this match {
  case Nil      => error("Nil.last")
  case x :: Nil => x
  case x :: xs => xs.last
}
```

L'implémentation de `init` est analogue.

Les trois fonctions suivantes renvoient un préfixe de la liste, un suffixe ou les deux.

```
def take(n: Int): List[A] =
  if (n == 0 || isEmpty) Nil else head :: tail.take(n-1)

def drop(n: Int): List[A] =
  if (n == 0 || isEmpty) this else tail.drop(n-1)

def split(n: Int): (List[A], List[A]) = (take(n), drop(n))
```

`(xs take n)` renvoie les `n` premiers éléments de la liste `xs` ou la liste entière si elle a moins de `n` éléments. `(xs drop n)` renvoie tous les éléments de `xs` sauf les `n` premiers. Enfin, `(xs split n)` renvoie une paire formée des listes obtenues par `xs take n` et `xs drop n`.

La fonction suivante renvoie l'élément situé à un indice donné, elle est donc analogue à l'indexation des tableaux. Les indices commencent à 0.

```
def apply(n: Int): A = drop(n).head
```

En Scala, la méthode `apply` a une signification spéciale car tout objet disposant de cette méthode peut se voir appliquer des paramètres comme s'il était une fonction. Pour obtenir le 3^{e} élément d'une liste `xs`, par exemple, vous pouvez écrire `xs.apply(2)` ou `xs(2)` (n'oubliez pas que les indices commencent à zéro) – la deuxième expression sera traduite dans la première.

Avec `take` et `drop`, nous pouvons extraire de la liste initiale des sous-listes d'éléments consécutifs. Pour extraire la sous-liste $\texttt{xs}_{\texttt{m}},\ldots\texttt{xs}_{\texttt{n-1}}$ de la liste `xs`, il suffit d'écrire :

```
xs.drop(m).take(n - m)
```

Combinaison de listes. La fonction suivante combine deux listes en une liste de paires. À partir de deux listes `xs = List(x1, …, xn)` et `ys = List(y1, …, yn)`, l'expression `xs zip ys` renvoie la liste `List((x1, y1), ..., (xn, yn))`. Si les deux listes n'ont pas le même nombre d'éléments, la plus longue est tronquée. Voici la définition de `zip` – vous remarquerez que c'est une méthode polymorphe :

```
def zip[B](that: List[B]): List[(a,b)] =
  if (this.isEmpty || that.isEmpty) Nil
  else (this.head, that.head) :: (this.tail zip that.tail)
```

Extension de listes. Comme tout opérateur infixe, :: est également implémenté comme une méthode portant sur un objet qui est, ici, la liste à étendre. Ceci est possible car les opérateurs se terminant par le caractère `:` sont traités de façon particulière par Scala : ils sont considérés comme des méthodes de leur opérande droit :

```
x :: y = y.::(x)
```

alors que :

```
x + y = x.+(y)
```

Notez cependant que les opérandes d'une opération binaire sont, dans tous les cas, évalués de gauche à droite. Par conséquent, si `D` et `E` sont des expressions avec d'éventuels effets de bord, `D :: E` est traduit en `{val x = D; E.::(x)}` afin de maintenir l'ordre d'évaluation des opérandes.

Une autre différence des opérateurs se terminant par `:` est qu'ils sont associatifs à droite alors que les autres sont associatifs à gauche :

```
x :: y :: z = x :: (y :: z)
```

alors que :

```
x + y + z = (x + y) + z
```

La définition de :: comme méthode de la classe `List` est la suivante :

```
def ::[B >: A](x: B): List[B] = new scala.::(x, this)
```

Notez que :: est défini pour tous les éléments `x` de type `B` et les listes de type `List[A]` tels que `B` est un supertype de `A`. Le résultat est une liste d'éléments de type `B`. Ceci est exprimé par le fait que, dans la signature de ::, le paramètre de type `B` a pour borne inférieure `A`.

Concaténation de listes. La concaténation de listes ressemble à :: et est notée :::. Le résultat de `(xs ::: ys)` est une liste formée de tous les éléments de `xs` suivis de tous les éléments de `ys`. ::: se terminant par un caractère `:`, elle est associative à droite et considérée comme une méthode de son opérande droit. Par conséquent :

```
xs ::: ys ::: zs  =     xs ::: (ys ::: zs)
                  =     zs.:::(ys).:::(xs)
```

Voici l'implémentation de la méthode ::: :

```
def :::[B >: A](prefix: List[B]): List[B] = prefix match {
  case Nil => this
  case p :: ps => this.:::(ps).::(p)
}
```

Renversement de listes. Une autre opération utile consiste à renverser les listes. C'est ce que fait la méthode `reverse` de `List`. Considérons cette implémentation possible :

```
def reverse[A](xs: List[A]): List[A] = xs match {
  case Nil => Nil
  case x :: xs => reverse(xs) ::: List(x)
}
```

Cette implémentation a l'avantage d'être simple, mais elle n'est pas très efficace car elle effectue une concaténation pour chaque élément de la liste. La concaténation prenant un temps proportionnel à la longueur de la première opérande, la complexité de cette version de `reverse` est donc :

```
n + (n - 1) + ... + 1 = n(n + 1)/2
```

où `n` est la longueur de `xs`. Peut-on implémenter `reverse` de façon plus efficace ? Nous verrons plus tard qu'il existe une autre implémentation qui a simplement une complexité linéaire.

Exemple : tri par fusion

Le tri par insertion que nous avons présenté plus haut dans ce chapitre est simple à formuler mais n'est pas très performant – sa complexité moyenne est en effet proportionnelle au carré de la longueur de la liste à trier. Nous allons donc concevoir un programme pour trier les éléments d'une liste de façon plus efficace. Le *tri par fusion* est un bon algorithme et fonctionne de la façon suivante :

Si la liste est vide ou n'a qu'un seul élément, elle est déjà triée et l'on renvoie donc cette liste inchangée. Les listes plus longues sont divisées en deux sous-listes contenant chacune la moitié des éléments de la liste initiale. Chaque sous-liste est ensuite triée par un appel récursif à la fonction de tri et les deux sous-listes ainsi triées sont ensuite fusionnées.

Pour une implémentation la plus générale possible du tri par fusion, nous devons préciser le type des éléments à trier, ainsi que la fonction utilisée pour les comparer. Ceci nous amène à l'implémentation suivante :

```
def msort[A](less: (A, A) => Boolean)(xs: List[A]): List[A] = {
  def merge(xs1: List[A], xs2: List[A]): List[A] =
    if (xs1.isEmpty) xs2
    else if (xs2.isEmpty) xs1
    else if (less(xs1.head, xs2.head)) xs1.head :: merge(xs1.tail, xs2)
    else xs2.head :: merge(xs1, xs2.tail)
  val n = xs.length/2
  if (n == 0) xs
  else merge(msort(less)(xs take n), msort(less)(xs drop n))
}
```

La complexité de `msort` est `O(N log(N))`, où `N` est la longueur de la liste d'entrée. Pour comprendre pourquoi, il suffit de remarquer que le découpage d'une liste en deux et la fusion de deux listes triées s'effectuent, chacune, dans un temps proportionnel à la longueur des listes passées en paramètre. Chaque appel récursif réduit de moitié le nombre d'éléments en entrée : il y a donc `O(log(N))` appels récursifs consécutifs avant d'atteindre le cas de base des listes de longueur 1. Cependant, pour les listes plus longues, chaque appel déclenche deux autres appels. Si l'on additionne le tout, on constate donc qu'à chacun des `O(log(N))` appels, chaque élément des listes initiales participe à une opération de

découpage et à une opération de fusion. Chaque appel a donc un coût total proportionnel à `O(N)`. Comme il y a `O(log(N))` appels, nous obtenant un coût total de `O(N log(N))`. Ce coût ne dépend pas de la distribution initiale des éléments dans la liste : le coût du pire des cas est donc le même que celui du cas général. Pour cette raison, le tri par fusion est un algorithme intéressant pour trier les listes.

Voici un exemple d'utilisation de `msort` :

```
msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))
```

La définition de `msort` étant curryfiée, il est facile de la spécialiser par des fonctions de comparaison particulières :

```
val intSort = msort((x: Int, y: Int) => x < y)
val reverseSort = msort((x: Int, y: Int) => x > y)
```

Définition de la classe `List` : méthodes du premier ordre (II)

Les exemples étudiés jusqu'à présent montre que les fonctions sur les listes ont souvent des structures similaires. Nous pouvons en effet identifier plusieurs motifs de traitement des listes :

- Transformation de chaque élément d'une liste.
- Extraction de tous les éléments satisfaisant un certain critère.
- Combinaisons des éléments d'une liste à l'aide d'un opérateur.

Les langages fonctionnels permettent aux programmeurs d'écrire des fonctions générales qui implémentent de tels motifs au moyen de fonctions du premier ordre. Nous allons donc maintenant présenter un ensemble de méthodes du premier ordre fréquemment utilisées et qui sont implémentées comme des méthodes de la classe `List`.

Transformations dans une liste. Une opération courante consiste à transformer chaque élément d'une liste et à renvoyer la liste des résultats. Pour, par exemple, multplier chaque élément d'une liste par un facteur donné :

```
def scaleList(xs: List[Double], factor: Double): List[Double] = xs match {
  case Nil => xs
  case x :: xs1 => x * factor :: scaleList(xs1, factor)
}
```

Ce motif peut être généralisé par la méthode `map` de la classe `List` :

```
abstract class List[A] {
  ...
  def map[B](f: A => B): List[B] = this match {
    case Nil => this
    case x :: xs => f(x) :: xs.map(f)
  }
```

On peut donc réécrire `scaleList` de façon plus concise en utilisant `map` :

```
def scaleList(xs: List[Double], factor: Double) =
  xs map (x => x * factor)
```

Comme autre exemple, étudiez le problème consistant à renvoyer une colonne donnée d'une matrice représentée par une liste de lignes, chaque ligne étant elle-même une liste de colonnes. Ce résultat peut être obtenu grâce à la fonction `column` suivante :

```
def column[A](xs: List[List[A]], index: Int): List[A] =
  xs map (row => row(index))
```

La méthode `foreach` est apparentée à `map`. Elle applique une fonction donnée à tous les éléments d'une liste mais, contrairement à map, ne construit pas la liste des résultats. On l'utilise donc uniquement pour son effet de bord. `foreach` est définie de la façon suivante :

```
def foreach(f: A => Unit) {
  this match {
    case Nil => ()
    case x :: xs => f(x); xs.foreach(f)
}
```

Cette fonction peut servir, par exemple, à afficher tous les éléments d'une liste :

```
xs foreach (x => println(x))
```

Exercice 9–4–1 : La fonction `squareList` renvoie la liste des carrés de tous les éléments d'une liste. Complétez les deux définitions suivantes de `squareList`, qui sont équivalentes :

```
def squareList(xs: List[Int]): List[Int] = xs match {
  case List() => ??
  case y :: ys => ??
}
def squareList(xs: List[Int]): List[Int] =
  xs map ??
}
```

Filtrage des listes. Une autre opération fréquente consiste à sélectionner dans une liste tous les éléments qui correspondent à un critère particulier. La fonction suivante, par exemple, renvoie la liste de tous les éléments positifs d'une liste d'entiers :

```
def posElems(xs: List[Int]): List[Int] = xs match {
  case Nil => xs
  case x :: xs1 => if (x > 0) x :: posElems(xs1) else posElems(xs1)
}
```

Ce motif est généralisé par la méthode `filter` de la classe `List` :

```
def filter(p: A => Boolean): List[A] = this match {
  case Nil => this
  case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
}
```

Grâce à filter, nous pouvons écrire posElems de façon plus concise :

```
def posElems(xs: List[Int]): List[Int] =
  xs filter (x => x > 0)
```

Une opération liée au filtrage consiste à tester si *tous* les éléments d'une liste satisfont une certaines condition. On peut également vouloir savoir si *un* élément de la liste correspond au critère. Ces opérations sont encapsulées par les fonctions du premier ordre `forall` et `exists` de la classe `List`.

```
def forall(p: A => Boolean): Boolean =
  isEmpty || (p(head) && (tail forall p))
def exists(p: A => Boolean): Boolean =
  !isEmpty && (p(head) || (tail exists p))
```

Pour illustrer l'utilisation de `forall`, considérons le problème consistant à déterminer si un nombre est premier. Le nombre *n* est premier s'il n'est divisible que par un et par lui-même. La traduction la plus directe de cette définition consiste donc à tester que toutes les divisions de *n* par les nombres de 2 à *n - 1* produisent un reste non nul. La liste des diviseurs possibles peut être produite par une fonction `List.range` qui est définie de la façon suivante dans la classe `List` :

```
package scala
object List { ...
  def range(from: Int, end: Int): List[Int] =
    if (from >= end) Nil else from :: range(from + 1, end)
```

`List.range(2, n)`, par exemple, produit la liste de tous les entiers compris entre 2 et *n-1*. La fonction `isPrime` peut donc être simplement définie de la façon suivante :

```
def isPrime(n: Int) =
  List.range(2, n) forall (x => n % x != 0)
```

Nous avons ainsi pu traduire directement en Scala la définition mathématique.

Exercice : Définissez les fonctions `forall` et `exists` à l'aide de `filter`.

Accumulation et réduction des listes. Une opération courante consiste à combiner les éléments d'une liste à l'aide d'un opérateur. Par exemple :

```
sum(List(x1, ..., xn))      = 0 + x1 + ... + xn
product(List(x1, ..., xn))  = 1 * x1 * ... * xn
```

Nous pourrions, bien sûr, implémenter ces deux fonctions sous forme récursive :

```
def sum(xs: List[Int]): Int = xs match {
  case Nil => 0
  case y :: ys => y + sum(ys)
}

def product(xs: List[Int]): Int = xs match {
  case Nil => 1
  case y :: ys => y * product(ys)
}
```

Mais nous pouvons également utiliser la généralisation de ce schéma fournie par la méthode `reduceLeft` de la classe `List` :

```
List(x1, ..., xn).reduceLeft(op) = (...(x1 op x2) op ... ) op xn
```

Avec `reduceLeft` nous pouvons donc mettre en évidence le motif de traitement utilisé par `sum` et `product` :

```
def sum(xs: List[Int])      = (0 :: xs) reduceLeft {(x, y) => x + y}
def product(xs: List[Int])  = (1 :: xs) reduceLeft {(x, y) => x * y}
```

Voici l'implémentation de `reduceLeft` :

```
def reduceLeft(op: (A, A) => A): A = this match {
  case Nil      => error("Nil.reduceLeft")
  case x :: xs => (xs foldLeft x)(op)
}
def foldLeft[B](z: B)(op: (B, A) => B): B = this match {
  case Nil => z
  case x :: xs => (xs foldLeft op(z, x))(op)
}
```

Nous pouvons constater que la méthode `reduceLeft` est définie à l'aide d'une autre méthode, `foldLeft`. Cette dernière prend un accumulateur `z` comme paramètre supplémentaire, qui est renvoyé lorsque `foldLeft` est appliquée à la liste vide. Par conséquent :

```
(List(x1, ..., xn) foldLeft z)(op) = (...(z op x1) op ... ) op xn
```

Les méthodes `sum` et `product` peuvent donc également être définies à partir de `foldLeft` :

```
def sum(xs: List[Int])      = (xs foldLeft 0) {(x, y) => x + y}
def product(xs: List[Int])  = (xs foldLeft 1) {(x, y) => x * y}
```

foldRight et reduceRight. Les appels de `foldLeft` et `reduceLeft` produisent des arbres penchés à gauche. Il existe également les méthodes duales `foldRight` et `reduceRight` qui produisent des arbres penchés à droite :

```
List(x1, ..., xn).reduceRight(op)       = x1 op ( ... (xn-1 op xn)...)
(List(x1, ..., xn) foldRight acc)(op)   = x1 op ( ... (xn op acc)...)
```

Ces deux méthodes sont définies de la façon suivante :

```
def reduceRight(op: (A, A) => A): A = this match {
  case Nil => error("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs => op(x, xs.reduceRight(op))
}

def foldRight[B](z: B)(op: (A, B) => B): B = this match {
  case Nil => z
  case x :: xs => op(x, (xs foldRight z)(op))
}
```

La classe `List` définit également deux abréviations symboliques de `foldLeft` et `foldRight` :

```
def /:[B](z: B)(f: (B, A) => B): B  = foldLeft(z)(f)
def :\[B](z: B)(f: (A, B) => B): B  = foldRight(z)(f)
```

Les noms des méthodes symbolisent les arbres penchés à gauche/droite des opérations `fold` par des barres penchées vers l'avant ou vers l'arrière. Le caractère `:` pointe toujours vers la liste alors que la barre pointe toujours vers l'accumulateur `z` :

```
(z /: List(x1, ..., xn))(op)  = (...(z op x1) op ... ) op xn
(List(x1, ..., xn) :\ z)(op)  = x1 op ( ... (xn op z)...)
```

Pour les opérateurs associatifs et commutatifs, `/:` et `:\` sont équivalents (bien que leur efficacité puisse être différente).

Exercice 9–4–2 : Soit la fonction `flatten` qui prend une liste en paramètre et renvoie la concaténation de tous les éléments sous la forme d'une liste simple. Voici une implémentation de cette méthode à l'aide de `:\` :

```
def flatten[A](xs: List[List[A]]): List[A] =
  (xs :\ (Nil: List[A])) {(x, xs) => x ::: xs}
```

Si l'on remplace le corps de `flatten` par :

```
((Nil: List[A]) /: xs) ((xs, x) => xs ::: x)
```

quelle serait la différence de complexité asymptotique entre ces deux versions ?

En fait, comme d'autres fonctions utiles, `flatten` est prédéfinie dans l'*objet* `List` de la bibliothèque standard. Elle est donc accessible via un appel à `List.flatten`. Ce n'est pas une méthode de la classe `List` – cela n'aurait aucun sens ici car elle ne s'applique qu'à une liste de listes, pas à toutes les listes en général.

Renversement de listes revisité. Nous avons présenté dans la section 9.2 une implémentation de la méthode `reverse` dont le temps d'exécution était quadratique par rapport à la longueur de la liste à renverser. Nous allons maintenant présenter une nouvelle version ayant un coût linéaire. Le principe consiste à utiliser `foldLeft` en se servant du schéma suivant :

```
class List[+A] { ...
  def reverse: List[A] = (z? /: this)(op?)
```

Il ne reste plus qu'à remplacer les parties `z?` et `op?`. Essayons de les déduire à partir d'exemples :

```
  Nil
= Nil.reverse           // selon la spécification
= (z /: Nil)(op)        // selon le schéma de reverse
= (Nil foldLeft z)(op)  // selon la définition de /:
= z                     // delon la définition de foldLeft
```

`z?` doit donc être `Nil`. Pour déduire le second opérande, étudions le renversement d'une liste de longueur un.

```
  List(x)
= List(x).reverse             // selon la spécification
= (Nil /: List(x))(op)        // selon le schéma de reverse, avec z = Nil
= (List(x) foldLeft Nil)(op)  // selon la définition de /:
= op(Nil, x)                  // selon la définition de foldLeft
```

Donc `op(Nil, x)` est égal à `List(x)`, c'est-à-dire à `x :: Nil`. Ceci suggère que `op` est l'opérateur :: avec ses opérandes échangées. Nous arrivons donc à l'implémentation suivante de `reverse`, qui a une complexité linéaire :

```
def reverse: List[A] =
  ((Nil: List[A]) /: this) {(xs, x) => x :: xs}
```

(Remarque : l'annotation de type de `Nil` est nécessaire pour que l'inférenceur de type puisse fonctionner).

Exercice 9–4–3 : Remplissez les expressions manquantes pour compléter ces définitions des opérations de base sur les listes sous forme d'opérations `fold` :

```
def mapFun[A, B](xs: List[A], f: A => B): List[B] =
  (xs :\ List[B]()){ ?? }

def lengthFun[A](xs: List[A]): int =
  (0 /: xs){ ?? }
```

Transformations imbriquées. Nous pouvons nous servir des fonctions du premier niveau sur les listes afin d'exprimer des calculs qui sont généralement réalisés par des boucles imbriquées dans les langages impératifs.

À titre d'exemple, considérons le problème consistant à trouver toutes les paires d'entiers `i` et `j` telles que `i + j` est premier, avec `1 j < i <` `n` pour une valeur `n` donnée. Pour `n = 7`, par exemple, ces paires sont :

```
i       | 2 3 4 4 5 6 6
j       | 1 2 1 3 2 1 5
=======================
i + j   | 3 5 5 7 7 7 11
```

Un moyen naturel de résoudre ce problème consiste à le traiter en deux étapes. La première produit toutes les paires d'entiers `(i, j)` telles que `1 j < i < n` ; la seconde filtre cette suite pour ne garder que les paires telles que `ì + j` est premier.

Si l'on étudie la première étape plus en détails, on constate que la production d'une suite de paires s'effectue en trois sous-étapes. On génère d'abord tous les entiers `i` compris entre 1 et `n` puis, pour chaque `i`, on produit la liste de paires `(i, 1)` jusqu'à `(i, i -1)`, ce qui peut s'effectuer par une combinaison de `range` et de `map` :

```
List.range(1, i) map (x => (i, x))
```

Enfin, on combine toutes ces sous-listes avec `foldRight` et :::. L'expression complète est donc la suivante :

```
List.range(1, n)
  .map(i => List.range(1, i).map(x => (i, x)))
  .foldRight(List[(Int, Int)]()) {(xs, ys) => xs ::: ys}
  .filter(pair => isPrime(pair._1 + pair._2))
```

Transformations avec aplatissement des listes. La combinaison d'une transformation et de la concaténation des sous-listes produites par cette transformation est une opération si fréquente qu'il existe une méthode prévue pour dans la classe `List` :

```
abstract class List[+A] { ...
  def flatMap[B](f: A => List[B]): List[B] = this match {
    case Nil => Nil
    case x :: xs => f(x) ::: (xs flatMap f)
  }
}
```

Avec `flatMap`, l'expression trouvant les paires dont la somme des éléments est un nombre premier peut s'écrire de façon plus courte :

```
List.range(1, n)
  .flatMap(i => List.range(1, i).map(x => (i, x)))
  .filter(pair => isPrime(pair._1 + pair._2))
```

Résumé

Ce chapitre a présenté les listes qui sont une structure de données fondamentale en programmation. Les listes ne sont pas modifiables et sont très souvent utilisées en programmation fonctionnelle – leur rôle est comparable à celui des tableaux des langages impératifs. Cependant, l'accès aux listes et aux tableaux sont très différents : alors qu'on accède généralement à un tableau par des indices, cette pratique est relativement rare avec les listes. Nous avons vu que `scala.List` définissait une méthode `apply` pour faire de même, mais elle est bien plus coûteuse que dans le cas des tableaux (linéaire au lieu de constant). Au lieu d'utiliser des indices, les listes sont le plus souvent parcourues récursivement, avec des étapes récursives qui reposent généralement sur un pattern matching appliqué à la liste. On dispose également d'un grand nombre de combinateurs du premier ordre qui permettent de représenter les motifs de traitement des listes les plus courants.

Chapitre 10. For en intension

Le chapitre précédent a montré que les fonctions du premier ordre comme `map`, `flatmap` et `filter` fournissaient des constructions puissantes pour traiter les listes. Parfois, cependant, le niveau d'abstraction requis par ces fonctions complique la compréhension des programmes.

Pour faciliter la relecture du code, Scala dispose d'une notation spéciale, qui simplifie les motifs courants d'application des fonctions du premier ordre. Cette notation établit un pont entre les ensembles en intension des mathématiques et les boucles `for` des langages impératifs comme C ou Java. Elle ressemble également beaucoup à la notation utilisée pour exprimer les requêtes des bases de données relationnelles.

Comme premier exemple, supposons que nous ayons une liste de personnes avec des champs `name` et `age`. Pour afficher les noms de toutes les personnes ayant plus de 20 ans, nous pourrions écrire :

```
for (p <- persons if p.age > 20) yield p.name
```

Cette expression est équivalente à celle-ci, qui utilise les fonctions `filter` et `map` :

```
persons filter (p => p.age > 20) map (p => p.name)
```

Le for en intension ressemble à la boucle `for` des langages impératifs, sauf qu'elle construit une *liste* du résultat de toutes ses itérations.

Généralement, un for en intension est de la forme :

```
for ( s ) yield e
```

où `s` est une suite de *générateurs*, de *définitions* et de *filtres*. Un *générateur* est de la forme **val** `x  e`, où `e` est une expression renvoyant une liste. Le générateur lie `x` aux valeurs des éléments successifs de la liste. Une *définition* est de la forme **val** `x = e` et introduit le nom `x` qui désignera la valeur de `e` dans le reste de l'intension. Un filtre est une expression `f` de type `Boolean` – il supprime toutes les liaisons pour lesquelles `f` vaut **false**. La séquence `s` commence toujours par un générateur – s'il y en a plusieurs, les derniers varient plus rapidement que les premiers.

La séquence `s` peut également être placée entre accolades au lieu d'être entre parenthèses. En ce cas, les points-virgules qui séparent les générateurs, les définitions et les filtres peuvent être omis.

Voici deux exemples montrant l'utilisation des for en intension. Reprenons d'abord un exemple du chapitre précédent : à partir d'un nombre `n` entier positif, nous voulons trouver toutes les paires `(i, j)` telles que `1  j < i < n` et `i + j` est premier. Ce problème se résoud de la façon suivante avec un for en intension :

```
for { i <- List.range(1, n)
      j <- List.range(1, i)
      if isPrime(i+j) } yield {i, j}
```

Ce code est, sans conteste, bien plus clair que la solution précédente qui utilisait `map`, `flatMap` et `filter`. Comme second exemple, calculons le produit scalaire de deux vecteurs `xs` et `ys` :

```
sum(for ((x, y) <- xs zip ys) yield x * y)
```

Le problème des N reines

Les for en intension sont spécialement utiles pour résoudre des problèmes combinatoires. Prenons, par exemple, le problème bien connu des 8 reines : sur un échiquier classique, nous voulons placer 8 reines telles qu'aucune d'elles ne peut capturer les autres (une reine ne peut capturer une pièce que si elle est sur la même ligne, la même colonne ou la même diagonale). Nous allons donc développer une solution à ce problème, en le généralisant aux échiquiers de taille quelconque. Le problème est donc de placer `n` reines sur un échiquier de `n × n` cases.

Pour le résoudre, vous noterez que nous devons placer une reine sur chaque ligne. Nous pourrions donc placer les reines sur les lignes successives et vérifiant à chaque fois que la reine que l'on vient de placer ne peut pas être capturée par l'une des reines déjà placées. Au cours de cette recherche, il peut arriver qu'une reine placée sur la ligne k soit en échec dans toutes les colonnes de cette ligne pour les reines placées aux lignes 1 à `k –1`. En ce cas, il faut mettre fin à cette partie de la recherche afin de continuer avec une autre configuration des reines dans les colonnes `1` à `k – 1`

Ceci suggère donc un algorithme récursif. Supposons que l'on ait déjà produit toutes les solutions pour placer `k – 1` reines sur un échiquier de `n × n` cases. Chacune de ces solutions peut être représentée par une liste de `k – 1` numéros de colonnes (qui peuvent varier de 1 à `n`). Nous traitons ces listes de solution partielle comme des piles où le numéro de colonne de la reine située sur la ligne `k – 1` est le premier de la liste, suivi du numéro de colonne de la reine dans la ligne `k – 2`, etc. L'élément au fond de la pile est le numéro de colonne de la reine placée sur la première ligne. Toutes ces solutions sont regroupées dans une liste de listes, dont chaque élément est une solution.

Pour placer la k^{e} reine, nous devons produire toutes les extensions possibles de chacune des solutions précédentes en leur ajoutant une reine supplémentaire. Ceci produit une autre liste de listes solution, cette fois-ci de longueur `k`. Ce processus se poursuit jusqu'à ce que nous ayons des solutions de la taille `n` de l'échiquier. Ce principe peut être implémenté de la façon suivante par la fonction placeQueens :

```
def queens(n: Int): List[List[Int]] = {
  def placeQueens(k: Int): List[List[Int]] =
    if (k == 0) List(List())
    else for { queens <- placeQueens(k - 1)
               column <- List.range(1, n + 1)
               if isSafe(column, queens, 1) } yield column :: queens placeQueens(n)
}
```

Exercice 10–1–1 : Écrivez la fonction

```
def isSafe(col: Int, queens: List[Int], delta: Int): Boolean
```

qui teste si l'on peut placer sans problème une reine dans la colonne `col` par rapport aux reines déjà placées. Ici, `delta` est la différence entre la ligne de la reine à placer et celle de la première reine de la liste.

Interrogation des for en intension

La notation `for` est équivalente aux opérations classique des langages de requêtes des bases de données. Supposons, par exemple, que la base de données `books` soit représentée par une liste de livres où `Book` est défini de la façon suivante :

```
case class Book(title: String, authors: List[String])
```

Voici une petite base de données d'exemple :

```
val books: List[Book] = List(
  Book("Structure and Interpretation of Computer Programs",
       List("Abelson, Harold", "Sussman, Gerald J.")),
  Book("Principles of Compiler Design",
       List("Aho, Alfred", "Ullman, Jeffrey")),
  Book("Programming in Modula-2",
       List("Wirth, Niklaus")),
  Book("Introduction to Functional Programming"),
       List("Bird, Richard")),
  Book("The Java Language Specification",
       List("Gosling, James", "Joy, Bill", "Steele, Guy", "Bracha, Gilad")))
```

Voici comment trouver les titres de tous les livres écrits par Ullman :

```
for (b <- books; a <- b.authors if a startsWith "Ullman")
  yield b.title
```

(Ici, `startsWith` est une méthode de `java.lang.String`).

Pour trouver tous les titres ayant "Program" dans leur titre :

```
for (b <- books if (b.title indexOf "Program") >= 0)
  yield b.title
```

Pour trouver les noms de tous les auteurs qui ont écrit au moins deux livres :

```
for (b1 <- books; b2 <- books if b1 != b2;
    a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
  yield a1
```

Cette solution n'est pas parfaite car les auteurs apparaîtront plusieurs fois dans la liste du résultat. Pour supprimer les auteurs dupliqués, vous pouvez utiliser la fonction suivante :

```
def removeDuplicates[A](xs: List[A]): List[A] =
  if (xs.isEmpty) xs
  else xs.head :: removeDuplicates(xs.tail filter (x => x != xs.head))
```

Notez que la dernière expression de `removeDuplicates` peut s'exprimer à l'aide d'un for en intension :

```
xs.head :: removeDuplicates(for (x <- xs.tail if x != xs.head) yield x)
```

Traduction des for en intension

Tout for en intension peut s'exprimer en termes des trois fonctions du premier ordre, `map`, `flatMap` et `filter`. Voici le schéma de traduction utilisé par le compilateur Scala :

- un for en intension simple

  ```
  for (x <- e) yield e'
  ```

  est traduit en :

  ```
  e.map(x => e')
  ```

- un for en intension

  ```
  for (x <- e if f; s) yield e'
  ```

où `f` est un filtre et `s` une suite (éventuellement vide) de générateurs ou de filtres est traduit en :

```
for (x <- e.filter(x => f); s) yield e'
```

puis cette dernière expression est ensuite traduite.

- un for en intension

```
for (x <- e; y <- e'; s) yield e''
```

où `s` est une suite (éventuellement vide) de générateurs ou de filtres est traduit en :

```
e.flatMap(x => for (y <- e'; s) yield e'')
```

puis cette dernière expression est ensuite traduite.

Si nous reprenons les "paires d'entiers sont la somme est un nombre premier" :

```
for { i <- range(1, n)
      j <- range(1, i)
      if isPrime(i+j)
  } yield {i, j}
```

Voici que l'on obtient en traduisant cette expression :

```
range(1, n)
  .flatMap(i =>
     range(1, i)
       .filter(j => isPrime(i+j))
       .map(j => (i, j)))
```

Inversement, il serait également possible d'exprimer les fonctions `map`, `flatMap` et `filter` à l'aide de for en intension :

```
object Demo {
  def map[A, B](xs: List[A], f: A => B): List[B] =
    for (x <- xs) yield f(x)

  def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =
    for (x <- xs; y <- f(x)) yield y

  def filter[A](xs: List[A], p: A => Boolean): List[A] =
    for (x <- xs if p(x)) yield x
}
```

Sans surprise, la traduction du for en intension dans le corps de `Demo.map` produira un appel à la méthode `map` de la classe `List`. De même, `Demo.flatMap` et `Demo.filter` seront traduites, respectivement, par les méthodes `flatMap` et `filter` de la classe `List`.

Exercice 10–3–1 : Définissez la fonction suivantes à l'aide de `for` :

```
def flatten[A](xss: List[List[A]]): List[A] =
  (xss :\ (Nil: List[A])) ((xs, ys) => xs ::: ys)
```

Exercice 10–3–2 : Traduisez :

```
for (b <- books; a <- b.authors if a startsWith "Bird") yield b.title
for (b <- books if (b.title indexOf "Program") >= 0) yield b.title
```

en fonctions du premier ordre.

Boucles for

Les for en intension ressemblent aux boucles `for` des langages impératifs, sauf qu'ils produisent une liste de résultats. Parfois, nous n'avons pas besoin de cette liste mais nous voulons quand même bénéficier de la souplesse des générateurs et des filtres dans nos parcours des listes. Ceci est possible grâce à une variante de la syntaxe de for en intension, qui permet d'exprimer des boucles `for` :

```
for ( s ) e
```

Cette construction est identique à la syntaxe standard des for en intension, mais sans le mot-clé `yield`. La boucle s'exécute en appliquant l'expression `e` à chaque élément produit par la suite de générateurs et de filtres `s`.

Le code suivant, par exemple, affiche tous les éléments d'une matrice représentée par une liste de listes :

```
for (xs <- xss) {
  for (x <- xs)
  print(x + "\t") println()
}
```

La traduction des boucles `for` en méthodes du premier niveau de la classe `List` est similaire à la traduction des for en intension, mais elle est plus simple. Là ou les for en intension se traduisent en `map` et `flatMap`, les boucles `for` se traduise dans chaque cas en `foreach`.

Généralisation de `for`

Nous avons vu que la traduction des for en intension n'a besoin que des méthodes `map`, `flatMap` et `filter` : il est donc possible d'appliquer la même notation au générateurs qui produisent des objets autres que des listes. Ces objets ont simplement besoin de disposer de ces trois méthodes essentielles.

La bibliothèque standard de Scala fournit plusieurs autres abstractions supportant ces trois méthodes et, avec elles, les for en intension. Nous en présenterons quelques unes dans les chapitres qui suivent. En tant que programmeur, vous pouvez également utiliser ce principe pour que les for en intension puissent s'appliquer à vos propres types – il suffit qu'ils disposent des méthodes `map`, `flatMap` et `filter`.

Il y a de nombreux cas où ceci peut s'avérer utile : pour les interfaces des bases de données, les arborescences XML ou les valeurs facultatives, notamment.

Attention : rien ne garantit automatiquement que le résultat de la traduction d'un for en intension sera correctement typé. Pour vous en assurer, les types de `map`, `flatMap` et `filter` doivent essentiellement être similaires aux types de ces méthodes dans la classe `List`.

Pour que ceci soit plus clair, supposons que vous ayez une classe paramétrée `C[A]` pour laquelle vous voulez autoriser les for en intension. La classe `C` devrait donc définir `map`, `flatMap` et `filter` avec les types suivants :

```
def map[B](f: A => B): C[B]
def flatMap[B](f: A => C[B]): C[B]
def filter(p: A => Boolean): C[A]
```

Il serait intéressant que le compilateur Scala impose statiquement ces types en exigeant, par exemple, que tout type disposant des for en intension implémente un trait standard avec ces méthodes (en Haskell, un langage disposant de la même fonctionnalité, cette abstraction est appelée "monade avec zéro"). Le problème est que ce trait standard devrait rendre abstrait l'identité de la classe `C` en utilisant, par exemple, `C` comme un paramètre de type. Ce paramètre serait un constructeur de type qui serait appliqué à *plusieurs types différents* dans les signatures des méthodes `map` et `flatMap`. Malheureusement, le système de typage de Scala est trop faible pour exprimer cette possibilité car il ne peut gérer que les paramètres de type qui sont des types totalement appliqués.

Chapitre 11. État modifiable

La plupart des programmes que nous avons étudié précédemment n'avaient pas d'effets de bord (nous ne tenons pas compte ici de l'affichage sur la sortie standard, ce qui, techniquement, est un effet de bord). La notion de *temps* n'avait donc pas d'importance. Pour un programme qui se termine, chaque suite d'actions aurait conduit au même résultat ! Ceci est également reflété par le modèle de substitution des calculs, où une étape de réécriture peut s'appliquer n'importe où dans un terme et où toutes les réécritures qui se terminent mènent à la même solution. En fait, cette propriété de *convergence* est un résultat important du lambda-calcul, la théorie sous-jacente de la programmation fonctionnelle.

Dans ce chapitre, nous présenterons des fonctions ayant des effets de bord et nous étudierons leur comportement. Nous verrons que l'une des conséquences est que nous devons modifier fondamentalement le modèle de substitution que nous avions employé jusqu'alors.

Objets avec état

Nous voyons normalement le monde comme un ensemble d'objets, dont certains ont un état qui *change* au cours du temps. Généralement, l'état est associé à un ensemble de variables qui peuvent être modifiées au cours d'un calcul. Il existe également une notion plus abstraite de l'état, qui ne fair référence à aucune construction particulière d'un langage de programmation : un objet *a un état* si son comportement est influencé par son histoire.

Un compte bancaire, par exemple, a un état car la question "puis-je retirer 100 €" peut avoir des réponses différentes au cours de la durée de vie de ce compte.

En Scala, l'état modifiable est construit à partir des variables. Une définition de variable s'écrit comme une définition de valeur mais est introduite par **var** au lieu de **val**. Les deux définitions suivantes, par exemple, introduisent et initialisent deux variables, `x` et `count` :

```
var x: String = "abc"
var count     = 111
```

Comme une définition de valeur, une définition de variable associe un nom à une valeur mais, dans le cas d'une définition de variable, cette association peut être ensuite modifiée par une affectation, qui s'écrit comme en C ou en Java :

```
x = "hello"
count = count + 1
```

En Scala, toute variable doit être initialisée lors de sa définition. L'instruction **`var x: Int;`**, par exemple, n'est *pas* considérée comme une définition de valeur car il manque l'initialisation (Lorsqu'une instruction comme celle-ci apparaît dans une classe, elle est considérée comme une déclaration de variable qui introduit des méthodes d'accès abstraites pour la variable mais ne les associe pas à un état). Si l'on ne connaît pas, ou que l'on ne veut pas connaître, la valeur d'initialisation adéquate, on peut utiliser un joker :

```
val x: T = _
```

initialisera `x` avec une valeur par défaut (**null** pour les types référence, **false** pour les booléens et la version appropriée de 0 pour les types valeurs numériques). Les objets du monde réel avec un état sont représentés en Scala par des objets ayant des variables dans leurs membres. Voici par exemple une classe représentant un compte bancaire :

```
class BankAccount {

  private var balance = 0

  def deposit(amount: Int) {
    if (amount > 0) balance += amount
  }

  def withdraw(amount: Int): Int =
    if (0 < amount && amount <= balance) {
      balance -= amount
      balance
    } else error("insufficient funds")
}
```

Cette classe définit une variable `balance` contenant la balance courante d'un compte. Les méthodes `deposit` et `withdraw` modifient la valeur de cette variable par des affectations. Vous remarquerez que `balance` est privée – elle ne peut donc pas être accédée directement depuis l'extérieur de la classe `BankAccount`.

Pour créer des comptes, on utilise la notation classique de création d'objets :

```
val myAccount = new BankAccount
```

Example 11–1–1 : Voici une session `scala` qui manipule des comptes bancaires.

```
scala> :l bankaccount.scala
Loading bankaccount.scala...
defined class BankAccount
scala> val account = new BankAccount
account: BankAccount = BankAccount\$class@1797795
scala> account deposit 50
res0: Unit = ()
scala> account withdraw 20
res1: Int = 30
scala> account withdraw 20
res2: Int = 10
scala> account withdraw 15
java.lang.Error: insufficient funds
    at scala.Predef$error(Predef.scala:74)
    at BankAccount$class.withdraw(<console>:14)
    at <init>(<console>:5)
scala>
```

Cet exemple montre qu'appliquer deux fois la même opération (`withdraw 20`) à un compte donne des résultats différents. Les comptes sont donc des objets avec état.

Identité et changement. Les affectations posent de nouveaux problèmes lorsque l'on veut savoir si deux expressions "sont identiques". Lorsque les affectations sont exclues et que l'on écrit :

```
val x = E; val y = E
```

où `E` est une expression quelconque, on peut raisonnablement supposer que `x` et `y` sont identiques. On aurait donc pu également écrire :

```
val x = E; val y = x
```

(Cette propriété est généralement appelée transparence référentielle). En revanche, si l'on autorise l'affectation, ces deux suites de définitions sont différentes. Considérez les définitions suivantes, par exemple :

```
val x = new BankAccount; val y = new BankAccount
```

Pour savoir si `x` et `y` sont identiques, nous devons être plus précis dans ce que nous entendons par "identique". La signification de ce mot est capturée par la notion d'*équivalence opérationnelle* qui, de façon plus informelle, est décrite par :

Supposons que nous ayons deux définitions de `x` et `y`. Pour tester si `x` et `y` définissent la même valeur, nous devons :

- Exécuter les définitions suivies par une séquence quelconque `S` d'opérations impliquant `x` et `y` et observer les éventuels résultats.
- Puis, exécuter les définitions avec une autre séquence `S'` produite à partir de `S` en renommant en `x` toutes les occurrences de `y` dans `S`.
- Si le résultat de `S'` est différent, alors `x` et `y` le sont sûrement aussi.
- Par contre, si toutes les paires de séquences {S, S'} possibles produisent le même résultat, alors `x` et `y` sont identiques.

En d'autres termes, l'équivalence opérationnelle considère que deux définitions `x` et `y` définissent la même valeur si aucun test ne permet de les distinguer. Dans ce contexte, un test est formé de deux versions d'un programme quelconque utilisant soit `x` soit `y`.

Testons donc si :

```
val x = new BankAccount; val y = new BankAccount
```

définissent des valeurs `x` et `y` identiques. Voici à nouveau les définitions, suivies d'une séquence de test :

```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> y withdraw 20
java.lang.RuntimeException: insufficient funds
```

Renommons maintenant en `x` toutes les occurrences de `y` dans cette séquence :

```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> x withdraw 20
10
```

Les deux résultats étant différents, nous avons donc établi que `x` et `y` ne sont pas identiques. En revanche, si nous définissons :

```
val x = new BankAccount; val y = x
```

aucune séquence d'opérations ne permet de distinguer `x` et `y`, qui sont donc identiques ici.

Affectation et modèle de substitution. Ces exemples montre que notre précédent modèle de substitution ne peut plus être utilisé pour les calculs. Avec ce modèle, nous pouvions toujours remplacer un nom de valeur par l'expression qui lui était liée :

```
val x = new BankAccount; val y = x
```

Ici, le `x` dans la définition de `y` pouvait être remplacé par **new** BankAccount. Mais nous venons de voir que cette modification menait à un programme différent. Le modèle de substitution n'est donc plus valide dès que l'on ajoute l'affectation.

Structures de contrôle impératives

Scala dispose des boucles **while** et **do–while** de C et Java. Il possède également une instruction **if** dont la partie **else** est facultative et d'une instruction **return** qui termine immédiatement la fonction qui l'invoque. Tout ceci permet donc d'écrire un programme dans un style impératif classique. La fonction suivante, par exemple, calcule le n^{e} puissance du paramètre `x` en utilisant une boucle **while** et un **if** sans alternative :

```
def power(x: Double, n: Int): Double = {
  var r = 1.0
  var i = n
  var j = 0
  while (j < 32) {
    r = r * r
    if (i < 0)
      r *= x
    i = i << 1
    j += 1
  }
  r
}
```

Ces structures de contrôles impératives sont présentes dans le langage pour des raisons de commodité, mais elles auraient pu être absentes car elles peuvent être implémentées uniquement à l'aide de fonctions. Développons, par exemple, une implémentation fonctionnelle de la boucle **while**. `whileLoop` est une fonction qui doit prendre deux paramètres : une condition de type `Boolean` et une commande de type `Unit`. Ces deux paramètres doivent être passés par nom, afin qu'ils soient évalués à chaque itération. Ceci nous conduit donc à la définition suivante :

```
def whileLoop(condition: => Boolean)(command: => Unit) {
  if (condition) {
    command; whileLoop(condition)(command)
  } else ()
}
```

Notez que `whileLoop` est récursive terminale : elle utilise donc une taille de pile constante.

Exercice 11–2–1 : Écrivez une fonction `repeatLoop` qui devra s'appliquer de la façon suivante :

```
repeatLoop { command } ( condition )
```

Peut-on également obtenir une syntaxe de boucle de la forme `repeatLoop { command } until ( condition )` ?

Certaines structures de contrôle de C et Java n'existent pas en Scala : il n'y a ni **break** ni **continue** pour les boucles. Il n'y a pas non plus de boucles **for** au sens de Java – elles ont été remplacées par la construction `for` que nous avons présentée dans la section 10.4.

Exemple : simulation d'événements discrets

Nous allons maintenant construire un simulateur de circuits logiques pour montrer comment combiner efficacement affectations et fonctions du premier ordre.

Cet exemple est tiré du livre d'Abelson et Sussman, *Structure and Interpretation of Computer Programs* - 2^{e} édition (MIT Press, 1996). Nous étendrons leur code de base (écrit en Scheme) par une structure orientée objet qui permet de réutiliser le code au moyen de l'héritage. Cet exemple montre également comment structurer et construire des programmes de simulation d'événements discrets.

Nous commencerons par un petit langage permettant de décrire les circuits logique. Un circuit est composé de connexions et de portes. Les connexions transportent les signaux qui sont transformés par les *portes*. Les signaux seront représentés par les booléens **true** et **false**.

Les portes de base sont :

- *inverseur*, qui inverse le signal.
- *et*, dont la sortie est la conjonction de ses entrées.
- *ou*, dont la sortie est la disjonction de ses entrées.

Toutes les autres portes peuvent être construites à partir de ces trois portes de base.

Lorsque ses entrées sont modifiées, la sortie d'une porte n'est elle-même modifiée qu'après un certain *délai*.

Langage pour les circuits numériques. Nous décrivons les éléments d'un circuit logique par un ensemble de classes et de fonctions Scala.

Il faut d'abord une classe `Wire` pour pouvoir construire des connexions :

```
val a = new Wire
val b = new Wire
val c = new Wire
```

Puis, nous avons besoin des procédures suivantes :

```
def inverter(input: Wire, output: Wire)
def andGate(a1: Wire, a2: Wire, output: Wire)
def orGate(o1: Wire, o2: Wire, output: Wire)
```

qui "créent" les portes de base dont nous avons besoin (sous forme d'effets de bord). Les portes plus complexes peuvent être construites à partir de ces trois procédures. Pour, par exemple, contruire un demi-additionneur, nous pouvons écrire :

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) {
  val d = new Wire
  val e = new Wire
  orGate(a, b, d)
  andGate(a, b, c)
  inverter(c, e)
  andGate(d, e, s)
}
```

Cette abstraction peut elle-même servir à construire un additionneur, par exemple :

```
def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire) {
  val s = new Wire
  val c1 = new Wire
  val c2 = new Wire
  halfAdder(a, cin, s, c1)
  halfAdder(b, s, sum, c2)
  orGate(c1, c2, cout)
}
```

La classe `Wire` et les fonctions `inverter`, `andGate` et `orGate` représentent donc un petit langage permettant aux utilisateurs de définir des circuits logiques. Nous allons maintenant les implémenter afin de pouvoir simuler le fonctionnement de circuits. Ces implémentations reposent sur une API simple et générale pour la simulation d'événements discrets.

L'API de simulation. La simulation d'événements discrets réalise des *actions* définies par l'utilisateur à des *moments* précis. Une action est représentée par une fonction qui ne prend pas de paramètre et renvoie un résultat Unit :

```
type Action = () => Unit
```

Ici, le temps sera simulé.

Une simulation concrète sera représentée par un objet qui hérite de la classe abstraite `Simulation` dont la signature est la suivante :

```
abstract class Simulation {
    def currentTime: Int
    def afterDelay(delay: Int, action: => Action)
    def run()
}
```

Ici, `currentTime` renvoie le temps courant, simulé sous la forme d'un entier. `afterDelay` planifie une action pour qu'elle s'exécute après un délai par rapport à `currentTime` et `run` exécute la simulation jusqu'à ce qu'il n'y ait plus d'actions à réaliser.

La classe Wire. Une connexion doit fournir trois actions de base :

`getSignal`	Booléen renvoyant le signal courant sur la connexion.
`setSignal(sig: Boolean)`	fixe le signal de la connexion à `sig`.
`addAction(p: Action)`	attache la procédure `p` indiquée aux actions de la connexion. Toutes les actions attachées s'exécuteront à chaque fois que le signal de la connexion change.

Voici une implémentation possible de la classe `Wire` :

```
class Wire {
  private var sigVal = false
  private var actions: List[Action] = List()
  def getSignal = sigVal
  def setSignal(s: Boolean) =
    if (s != sigVal) {
      sigVal = s
      actions.foreach(action => action())
    }
  def addAction(a: Action) {
    actions = a :: actions; a()
}
```

L'état d'une connexion est représenté par deux variables privées. `sigVal` contient le signal courant et `actions` représente les procédures actuellement attachées à la connexion.

La classe Inverseur. Un inverseur est implémenté en installant une action sur sa connexion d'entrée – l'action qui place le signal d'entrée complémenté sur le signal de sortie. Cette action doit prendre effet `InverterDelay` unités de temps après la modification de l'entrée :

```
def inverter(input: Wire, output: Wire) {
  def invertAction() {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) { output setSignal !inputSig }
  }
  input addAction invertAction
}
```

La classe ET. Les portes ET sont implémentées comme les inverseurs. L'action d'une telle porte consiste à mettre sur sa connexion de sortie la conjonction de ses signals d'entrée `AndGateDelay` unités de temps après que l'une de ses entrées ait changé :

```
def andGate(a1: Wire, a2: Wire, output: Wire) {
  def andAction() {
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) { output setSignal (a1Sig & a2Sig) }
  }
  a1 addAction andAction
  a2 addAction andAction
}
```

Exercice 11–3-1 : Proposez une implémentation de `orGate`.

Exercice 11–3-2 : Un autre moyen de définir une porte OU consiste à combiner des inverseurs et des portes ET. Définissez une fonction `orGate` en termes de `andGate` et `inverter`. Quel est le délai de cette fonction ?

La classe Simulation. Il reste simplement à implémenter la classe `Simulation` et nous aurons terminé. Le principe est de maintenir un agenda d'actions à réaliser dans un objet `Simulation`. Cet agenda est représenté par une liste de paires d'actions et de moments où ces actions doivent s'exécuter. Cette liste est triée, de sorte que les actions à exécuter en premier soient les premières de la liste.

```
abstract class Simulation {
  case class WorkItem(time: Int, action: Action)
  private type Agenda = List[WorkItem]
  private var agenda: Agenda = List()
```

Cette classe utilise également une variable privée `curtime` pour mémoriser le moment présent (simulé).

```
  private var curtime = 0
```

L'appel de la méthode `afterDelay(delai, bloc)` insère l'élément `WorkItem(currentTime + delai, () => bloc)` au bon endroit dans la liste d'agenda.

```
private def insert(ag: Agenda, item: WorkItem): Agenda =
  if (ag.isEmpty || item.time < ag.head.time) item :: ag
  else ag.head :: insert(ag.tail, item)

def afterDelay(delay: Int)(block: => Unit) {
  val item = WorkItem(currentTime + delay, () => block)
  agenda = insert(agenda, item)
}
```

L'appel de la méthode `run` supprime les éléments de l'agenda et exécute leurs actions, jusqu'à ce que l'agenda soit vide :

```
private def next() {
  agenda match {
    case WorkItem(time, action) :: rest =>
      agenda = rest; curtime = time; action()
    case List() =>
  }
}

def run() {
  afterDelay(0) { println("*** simulation started ***") }
  while (!agenda.isEmpty) next()
}
```

Exécution du simulateur. Pour exécuter le simulateur, nous avons besoin de pouvoir inspecter les modifications des signaux sur les connexions. C'est ce que fait la fonction `probe` :

```
def probe(name: String, wire: Wire) {
  wire addAction { () =>
     println(name + " " + currentTime + " new_value = " + wire.getSignal)
  }
}
```

Pour voir le simulateur en action, définissons maintenant quatre connexions et testons deux d'entre elles :

```
scala> val input1, input2, sum, carry = new Wire

scala> probe("sum", sum)
sum 0 new_value = false

scala> probe("carry", carry)
carry 0 new_value = false
```

Définissons ensuite un demi-additionneur qui relie ces connexions :

```
scala> halfAdder(input1, input2, sum, carry)
```

Enfin, configurons à **true** les signaux sur les deux entrées et lançons la simulation :

```
scala> input1 setSignal true; run
*** simulation started ***
sum 8 new_value = true

scala> input2 setSignal true; run
carry 11 new_value = true
sum 15 new_value = false
```

Résumé

Ce chapitre à présenté les instructions permettant de modéliser l'état en Scala – variables, affectations et structures de contrôles impératives. L'état et l'affectation compliquent notre représentation mentale des calculs ; on perd notamment la transparence référentielle. En revanche, les affectations permettent de formuler élégamment les programmes. Comme toujours, c'est la situation qui dictera le choix entre une programmation purement fonctionnelle et une programmation avec des affectations.

Chapitre 12. Utilisation des flux

Le chapitre précédent a présenté les variables, l'affectation et les objets avec état. Nous avons vu que nous pouvions modéliser les objets du monde réel qui évoluent avec le temps en modifiant l'état des variables au cours du traitement. Les modifications du temps dans le monde réel sont ainsi modélisées par des changements du temps au cours de l'exécution d'un programme. Ces modifications temporelles sont généralement réduites ou compressées mais leur ordre relatif est le même. Tout ceci semble naturel, mais il y a un prix à payer : le modèle de substitution simple et puissant que nous utilisions en programmation fonctionnelle n'est plus applicable dès que l'on introduit des variables et des affectations.

Existe-t-il un autre moyen ? Peut-on modéliser les changements d'états du monde réel en n'utilisant que des fonctions qui ne modifient pas leurs paramètres ? Si l'on se fie aux mathématiques, la réponse est évidemment oui : une quantité de temps variable est simplement modélisée par une fonction f(t), où t est un temps. Il en va de même pour nos programmes. Au lieu d'écraser une variable avec des valeurs successives, nous pouvons représenter toutes ces valeurs comme les éléments successifs d'une liste. Une variable modifiable **var** `x: T` peut donc être remplacée par une valeur non modifiable **val** `x: List[T]`. Dans un certains sens, nous sacrifions l'espace mémoire pour le temps – les différentes valeurs de la variables existent désormais toutes simultanément en tant qu'éléments de la liste. L'avantage de cette vue est que nous pouvons "voyager dans le temps", c'est-à-dire voir plusieurs valeurs successives de la variable. Un autre intérêt est que nous pouvons utiliser la puissance de la bibliothèque de fonctions sur les listes, qui simplifie souvent les traitements. Considérez, par exemple, l'approche impérative pour calculer la somme de tous les nombres premiers compris entre deux bornes :

```
def sumPrimes(start: Int, end: Int): Int = {
  var i = start
  var acc = 0
  while (i < end) {
    if (isPrime(i)) acc += i i += 1
  }
  acc
}
```

Vous remarquerez que la variable `i` "passe sur" toutes les valeurs de l'intervalle `[start .. end - 1]`.

Une approche plus fonctionnelle consiste à représenter directement la liste des valeurs de `i` avec `range(start, end)`. La fonction peut alors être réécrite de la façon suivante :

```
def sumPrimes(start: Int, end: Int) =
  sum(range(start, end) filter isPrime)
```

On voit tout de suite quel programme est le plus court et le plus clair ! Cependant, le programme fonctionnel est également considérablement moins efficace car il construit une liste de tous les nombres de l'intervalle, puis une autre pour les nombres premiers. C'est encore pire si l'on recherche le deuxième nombre premier entre 1000 et 10000 :

```
range(1000, 10000) filter isPrime at 1
```

Ce code construit la liste de tous les nombres compris entre 1000 et 10000 alors que l'essentiel de cette liste ne sera jamais parcouru !

Pour ce type d'exemple, nous pouvons cependant obtenir une exécution efficace en utilisant l'astuce suivante :

> Évitez de calculer la fin d'une séquence sauf si elle est nécessaire au traitement.

Nous allons donc présenter une nouvelle classe pour gérer ce type de séquences : la classe `Stream`.

Les *streams* (flux) sont créés à partir de la constante `empty` et du constructeur `cons` qui sont tous les deux définis dans le module `scala.Stream`. L'expression suivante, par exemple, construit un flux contenant les éléments 1 et 2 :

```
Stream.cons(1, Stream.cons(2, Stream.empty))
```

Voici un autre exemple, produisant un flux analogue à la liste produite par List.range :

```
def range(start: Int, end: Int): Stream[Int] =
  if (start >= end) Stream.empty
  else Stream.cons(start, range(start + 1, end))
```

(cette fonction existe déjà dans le module `Stream`). Cependant, bien que `Stream.range` et `List.range` se ressemblent, leur comportement est totalement différent :

`Stream.range` se termine immédiatement en renvoyant un objet `Stream` dont le premier élément est `start` ; les autres éléments ne seront calculés que lorsqu'ils seront *demandés* par la méthode `tail` (ce qui peut ne jamais arriver).

On accède aux éléments des flux comme aux éléments des listes. Comme ces dernières, les méthodes de base sont `isEmpty`, `head` et `tail`. Nous pouvons par exemple afficher tous les éléments d'un flux de la façon suivante :

```
def print(xs: Stream[A]) {
  if (!xs.isEmpty) { Console.println(xs.head); print(xs.tail) }
}
```

Les flux disposent également de la plupart des méthodes définies sur les listes (voir plus bas). Nous pouvons donc trouver le second nombre premier compris entre 1000 et 10000 en nous servant des méthodes `filter` et `apply` d'un flux :

```
Stream.range(1000, 10000) filter isPrime at 1
```

La différence avec l'implémentation reposant sur une liste est que, désormais, nous ne construisons pas inutilement les nombres supérieurs à 1013 et nous ne testons pas non plus s'ils sont premiers.

Construction et concaténation des flux. Deux méthodes de la classe `List` ne sont pas reconnues par la classe `Stream` : il s'agit de :: et de ::: car le récepteur de ces méthodes est leur opérande droit, ce qui signifie qu'il doit être évalué avant que la méthode ne soit appelée. Dans le cas d'une opération `x` :: `xs` sur une liste, par exemple, il faut évaluer `xs` avant d'appeler :: pour construire la nouvelle liste. Ceci ne fonctionne pas pour les flux car nous voulons justement que la fin d'un flux ne soit évaluée que si cela est demandé par l'opération `tail`. C'est pour la même raison que la concaténation des listes ::: n'est pas non plus adaptée aux flux.

Au lieu de `x` :: `xs`, utilisez `Stream.cons(x, xs)` pour construire un flux dont le premier élément sera `x` et le reste (non évalué) sera `xs`. À la place de `xs` ::: `ys`, utilisez `xs append ys`.

Chapitre 13. Itérateurs

Les itérateurs sont la version impérative des flux. Comme eux, les itérateurs décrivent des listes potentiellement infinies, mais il n'existe pas de structure de données contenant leurs éléments. Les itérateurs permettent plutôt de progresser dans une séquence à l'aide de deux méthodes abstraites, `next` et `hasNext` :

```
trait Iterator[+A] {
  def hasNext: Boolean
  def next: A
}
```

La méthode `next` renvoie les éléments successifs de l'itération tandis que la méthode `hasNext` indique s'il reste des éléments pouvant être renvoyés par `next`. Les itérateurs disposent également d'autres méthodes que nous présenterons plus loin.

Voici, par exemple, une application qui affiche les carrés de tous les nombres de 1 à 100 :

```
val it: Iterator[Int] = Iterator.range(1, 100)
while (it.hasNext) {
  val x = it.next
  println(x * x)
}
```

Méthodes des itérateurs

Outre `next` et `hasNext`, les itérateurs ont un grand nombre de méthodes dont la plupart imitent la fonctionnalité correspondante des listes.

append. La méthode `append` construit un itérateur qui continuera avec l'itérateur indiqué lorsque l'itérateur courant sera terminé.

```
def append[B >: A](that: Iterator[B]): Iterator[B] = new Iterator[B] {
  def hasNext = Iterator.this.hasNext || that.hasNext
  def next = if (Iterator.this.hasNext) Iterator.this.next else that.next
}
```

Les termes `Iterator.this.next` et `Iterator.this.hasNext` dans la définition de `append` appellent les méthodes respectives définies dans la classe `Iterator` du récepteur. Sans le préfixe `Iterator`, les appels `hasNext` et `next` invoqueraient les méthodes définies dans le résultat de `append`, ce qui n'est pas ce que l'on recherche.

map, flatMap, foreach. La méthode `map` construit un itérateur qui renvoie tous les éléments de l'itérateur initial transformés par la fonction `f` indiquée.

```
def map[B](f: A => B): Iterator[B] = new Iterator[B] {
  def hasNext = Iterator.this.hasNext
  def next = f(Iterator.this.next)
}
```

La méthode `flatMap` est comme la méthode `map`, sauf que la fonction de transformation `f` renvoie un itérateur. is like method map, except that the transformation function f now returns an iterator. Son résultat est l'itérateur produit par la concaténation de tous les itérateurs renvoyés par les appels successifs à `f`.

```
def flatMap[B](f: A => Iterator[B]): Iterator[B] = new Iterator[B] {
  private var cur: Iterator[B] = Iterator.empty
  def hasNext: Boolean =
    if (cur.hasNext) true
    else if (Iterator.this.hasNext) { cur = f(Iterator.this.next); hasNext }
    else false
  def next: B =
    if (cur.hasNext) cur.next
    else if (Iterator.this.hasNext) { cur = f(Iterator.this.next); next }
    else error("next on empty iterator")
}
```

La méthode `foreach` est étroitement liée à `map` car elle applique une fonction donnée à tous les éléments d'un itérateur. Cependant, elle ne construit pas la liste des résultats.

```
def foreach(f: A => Unit): Unit =
  while (hasNext) { f(next) }
```

filter. La méthode `filter` construit un itérateur renvoyant tous les éléments de l'itérateur initial qui satisfont le critère donné.

```
def filter(p: A => Boolean) = new BufferedIterator[A] {
  private val source = Iterator.this.buffered
  private def skip =
    { while (source.hasNext && !p(source.head)) { source.next } }
  def hasNext: Boolean =
    { skip; source.hasNext }
  def next: A =
    { skip; source.next }
  def head: A =
    { skip; source.head }
}
```

En fait, `filter` renvoie des instances d'une sous-classe d'itérateurs "tamponnés". Un objet `BufferedIterator` est un itérateur qui a en plus une méthode `head` qui renvoie l'élément qui aurait été renvoyé par `next`, mais sans avancer après cet élément. La valeur renvoyée par `head` sera donc à nouveau renvoyé par le prochain appel à `head` ou à `next`. Voici la définition du trait `BufferedIterator` :

```
trait BufferedIterator[+A] extends Iterator[A] {
  def head: A
}
```

Les méthodes `map`, `flatMap`, `filter` et `foreach` existant pour les itérateurs, il s'ensuit que les for en intension ou les boucles `for` leur sont également applicables. Le code qui affiche les carrés des nombres compris entre 1 et 100 pourrait donc s'exprimer de la façon suivante :

```
for (i <- Iterator.range(1, 100)) println(i * i)
```

zip. La méthode `zip` prend un autre itérateur en paramètre et renvoie un itérateur formé des paires d'éléments correspondants renvoyés par les deux itérateurs.

```
def zip[B](that: Iterator[B]) = new Iterator[(A, B)] {
  def hasNext = Iterator.this.hasNext && that.hasNext
  def next = (Iterator.this.next, that.next)
}
```

Construction d'itérateurs

Les itérateurs concrets doivent fournir les implémentations des deux méthodes `next` et `hasNext` de la classe `Iterator`. L'itérateur le plus simple est `Iterator.empty` qui renvoie toujours une séquence vide :

```
object Iterator {
  object empty extends Iterator[Nothing] {
    def hasNext = false
    def next = error("next on empty iterator")
  }
```

Un itérateur plus intéressant est celui qui énumère tous les éléments d'un tableau. Il est construit par la méthode `fromArray`, qui est également définie dans l'objet `Iterator` :

```
  def fromArray[A](xs: Array[A]) = new Iterator[A] {
    private var i = 0
    def hasNext: Boolean =
      i < xs.length
    def next: A =
      if (i < xs.length) { val x = xs(i); i += 1; x }
      else error("next on empty iterator")
  }
```

Un autre itérateur énumère un intervalle d'entiers. La fonction `Iterator.range` renvoie un itérateur qui parcourt un intervalle de valeurs entières :

```
object Iterator {
  def range(start: Int, end: Int) = new Iterator[Int] {
    private var current = start
    def hasNext = current < end
    def next = {
      val r = current
      if (current < end) current += 1
      else error("end of iterator") r
    }
  }
}
```

Tous les itérateurs que nous venons de présenter finissent par se terminer. Vous pouvez également définir des itérateurs infinis. L'itérateur suivant, par exemple, renvoie tous les entiers à partir de la valeur `value` (à cause de la représentation finie du type `int`, les nombres reboucleront à 2^{31}) :

```
def from(start: Int) = new Iterator[Int] {
  private var last = start - 1
  def hasNext = true
  def next = { last += 1; last }
}
```

Utilisation des itérateurs

Voici deux exemples supplémentaires d'utilisation des itérateurs. Le premier affiche tous les éléments d'un tableau `xs` :

```
Iterator.fromArray(xs) foreach (x => println(x))
```

Ou, avec un for en intension :

```
for (x <- Iterator.fromArray(xs)) println(x)
```

Le second recherche les indices des éléments d'un tableau de `double` supérieurs à `limit` :

```
import Iterator._
fromArray(xs)
 .zip(from(0))
 .filter(case (x, i) => x > limit)
 .map(case (x, i) => i)
```

Ou, avec un for en intension :

```
import Iterator._
for ((x, i) <- fromArray(xs) zip from(0); x > limit)
  yield i
```

Chapitre 14. Valeurs à la demande

Les valeurs à la demande (ou valeurs "paresseuses") permettent de différer l'évaluation d'une valeur jusqu'à ce qu'elle soit utilisée pour la première fois. Ceci peut être notamment utile lorsque l'on manipule des valeurs qui pourraient ne pas être nécessaires au calcul et dont le coût du traitement est significatif. Prenons l'exemple d'une base de données d'employés où chaque employé contient son supérieur hiérarchique et les membres de son équipe :

```
case class Employee(id: Int, name: String, managerId: Int) {
  val manager: Employee = Db.get(managerId)
  val team: List[Employee] = Db.team(id)
}
```

Cette classe Employee initialisera immédiatement tous ses champs en chargeant toute la table des employés en mémoire. Ceci n'est sûrement pas optimal et peut être aisément amélioré en utilisant des champs `lazy` : l'accès à la base de données sera ainsi différé jusqu'au moment où elle est réellement nécessaire.

```
case class Employee(id: Int, name: String, managerId: Int) {
  lazy val manager: Employee = Db.get(managerId)
  lazy val team: List[Employee] = Db.team(id)
}
```

Pour voir ce qui se passe réellement, nous pouvons nous servir de cette base d'exemple qui affiche quand ses enregistrements sont lus :

```
object Db {
  val table = Map(1 -> (1, "Haruki Murakami", -1),
                  2 -> (2, "Milan Kundera", 1),
                  3 -> (3, "Jeffrey Eugenides", 1),
                  4 -> (4, "Mario Vargas Llosa", 1),
                  5 -> (5, "Julian Barnes", 2))

  def team(id: Int) = {
    for (rec <- table.values.toList; if rec._3 == id)
      yield recToEmployee(rec)
  }

  def get(id: Int) = recToEmployee(table(id))

  private def recToEmployee(rec: (Int, String, Int)) = {
    println("[db] fetching " + rec._1) Employee(rec._1, rec._2, rec._3)
  }
}
```

Lorsque l'on exécute un programme qui récupère un seul employé, l'affichage confirme que la base de données n'est accédée que lorsque l'on accède aux variables paresseuses.

Un autre cas d'utilisation de ces variables est la résolution de l'ordre d'initialisation des applications composées de plusieurs modules : avant que nous ne connaissions l'existence des variables paresseuses, nous utilisions pour cela des définitions d'objets. Considérons, par exemple, un compilateur composé de plusieurs modules. Examinons d'abord une table des symboles qui définit une classe pour les symboles et deux fonctions prédéfinies :

```
class Symbols(val compiler: Compiler) {
  import compiler.types._

  val Add = new Symbol("+", FunType(List(IntType, IntType), IntType))
  val Sub = new Symbol("-", FunType(List(IntType, IntType), IntType))
```

```
  class Symbol(name: String, tpe: Type) {
    override def toString = name + ": " + tpe
  }
}
```

Le module `symbols` est paramétré par une instance de `Compiler` qui fournit l'accès à d'autres services, comme le module `types`. Dans notre exemple, il n'existe que deux fonctions prédéfinies, l'addition et la soustraction, dont les définitions dépendent du module `types`.

```
class Types(val compiler: Compiler) {
  import compiler.symtab._

  abstract class Type
  case class FunType(args: List[Type], res: Type) extends Type
  case class NamedType(sym: Symbol) extends Type
  case object IntType extends Type
}
```

Pour lier ces deux composants, on crée un objet compilateur et on leur passe en paramètre :

```
class Compiler {
  val symtab = new Symbols(this)
  val types = new Types(this)
}
```

Malheureusement, cette approche échoue à l'exécution car le module `symtab` a besoin du module `types`. En règle générale, la dépendance entre les modules peut être compliquée et il est difficile d'obtenir le bon ordre d'initialisation – voire impossible lorsqu'il y a des cycles. Une solution simple à ce problème consiste à rendre ces champs paresseux et à laisser le compilateur s'occuper de l'ordre :

```
class Compiler {
  lazy val symtab = new Symbols(this)
  lazy val types = new Types(this)
}
```

Les deux modules seront désormais initialisés lors de leur premier accès et le compilateur peut s'exécuter comme on l'avait prévu.

Syntaxe

Le modificateur `lazy` n'est autorisé qu'avec les définitions de valeurs concrètes. Toutes les règles de typage s'appliquent également aux valeurs paresseuses et les valeurs locales récursives sont également autorisées.

Chapitre 15. Paramètres et conversions implicites

Les paramètres et les conversions implicites sont des outils puissants qui permettent d'adapter les bibliothèques existantes et de créer des abstractions de haut niveau. À titre d'exemple, commençons par la classe abstraite des demi-groupes qui disposent d'un opération `add` :

```
abstract class SemiGroup[A] {
  def add(x: A, y: A): A
}
```

La sous-classe `Monoid` ajoute un élément unité :

```
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
}
```

Voici deux implémentations des monoïdes :

```
object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x.concat(y)
  def unit: String = ""
}

object intMonoid extends Monoid[Int] {
  def add(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

Une méthode `sum` qui fonctionnerait avec des monoïdes quelconques pourrait s'écrire de la façon suivante en Scala :

```
def sum[A](xs: List[A])(m: Monoid[A]): A =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(m)(xs.tail)
```

Cette méthode `sum` pourra alors être appelée ainsi :

```
sum(List("a", "bc", "def"))(stringMonoid)
sum(List(1, 2, 3))(intMonoid)
```

Tout ceci fonctionne mais n'est pas très agréable à lire. Le problème est qu'il faut passer les implémentations des monoïdes à tous les codes qui les utilisent. Nous aimerions parfois que le système puisse trouver automatiquement les bons paramètres, un peu comme ce qui se passe lorsque les paramètres de types sont inférés. C'est exactement ce que permettent de faire les paramètres implicites.

Paramètres implicites : les bases

Le mot-clé `implicit` a été introduit en Scala 2 et peut apparaître au début d'une liste de paramètres. Sa syntaxe est la suivante :

```
ClausesParam ::= {'(' [Param {',' Param}] ')'}
                 ['(' implicit Param {',' Param} ')']
```

Lorsque ce mot-clé est présent, tous les paramètres de la liste deviennent implicites. La version suivante de `sum`, par exemple, a un paramètre `m` implicite :

```
def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail))
```

Comme on le peut le constater avec cet exemple, il est possible de combiner des paramètres normaux et implicites. Cependant, il ne peut y avoir qu'une seule liste de paramètres implicites pour une méthode ou un constructeur donnés et elle doit être placée à la fin.

`implicit` peut également servir de modificateur pour les définitions et les déclarations :

```
implicit object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x.concat(y)
  def unit: String = ""
}
implicit object intMonoid extends Monoid[Int] {
  def add(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

L'idée principale derrière les paramètres implicites est que ces paramètres peuvent être absents lors de l'appel : ils seront alors inférés par le compilateur Scala.

Les paramètres effectifs qui peuvent être passés comme paramètres implicites sont tous les identificateurs X accessibles à l'endroit de l'appel de de méthode sans préfixe et qui dénotent une définition ou un paramètre implicite.

Si plusieurs paramètres peuvent correspondre au type du paramètre implicite, le compilateur choisira le plus spécifique en se servant des règles classiques de résolution de la surcharge statique. Supposons par exemple, que l'appel

```
sum(List(1, 2, 3))
```

s'effectue dans un contexte où `stringMonoid` et `intMonoid` sont tous les deux visibles. Nous savons que le paramètre de type `A` de `sum` doit être instancié en `Int` et la seule valeur possible correspondant au paramètre formel implicite `Monoid[Int]` est alors `intMonoid` : c'est donc cet objet qui sera passé comme paramètre implicite.

Cette explication montre également que les types implicites sont inférés après tous les paramètres de types.

Conversions implicites

Supposons que vous ayez une expression `E` de type `T` et que vous attendez un type `S`. `T` n'est pas conforme à `S` et n'est pas convertible en `S` par une conversion prédéfinie. Le compilateur Scala tentera d'appliquer en dernier ressort une conversion implicite `I(E)` où `I` est un identificateur représentant une définition ou un paramètre implicite accessible sans préfixe à l'endroit de la conversion et pouvant s'appliquer à des paramètres de type `T` pour produire un résultat conforme au type `S`.

Les conversions implicites peuvent également s'appliquer lors des sélections de membres. Si, dans une sélection `E.x`, `x` n'est pas un membre du type `E`, le compilateur essaira d'insérer une conversion implicite `I(E).x` pour que `x` soit un membre de `I(E)`.

Voici un exemple de fonction de conversion implicite, qui convertit des entiers en instances de la classe `scala.Ordered` :

```
implicit def int2ordered(x: Int): Ordered[Int] = new Ordered[Int] {
  def compare(y: Int): Int =
    if (x < y) -1 else if (x > y) 1 else 0
}
```

Bornes vues

Les bornes vues sont du sucre syntaxique pour représenter les paramètres implicites. Soit la méthode de tri générique suivante :

```
def sort[A <% Ordered[A]](xs: List[A]): List[A] =
  if (xs.isEmpty || xs.tail.isEmpty) xs
  else {
    val {ys, zs} = xs.splitAt(xs.length / 2)
    merge(ys, zs)
}
```

Le paramètre de type borné par une vue `[A <% Ordered[A]]` indique que `sort` est applicable aux listes d'éléments d'un type pour lequel il existe une conversion implicite de `A` vers `Ordered[A]`. Cette définition est donc un raccourci pour la signature suivante, qui a un paramètre implicite :

```
def sort[A](xs: List[A])(implicit c: A => Ordered[A]): List[A] = ...
```

(Ici, le nom du paramètre `c` a été choisi pour ne pas entrer en conflit avec les autres noms du programme).

Pour un exemple plus détaillé, étudiez la méthode `merge` utilisée par la méthode `sort` :

```
def merge[A <% Ordered[A]](xs: List[A], ys: List[A]): List[A] =
   if (xs.isEmpty) ys
   else if (ys.isEmpty) xs
   else if (xs.head < ys.head) xs.head :: merge(xs.tail, ys)
   else if ys.head :: merge(xs, ys.tail)
```

Après la traduction de la borne vue et l'insertion de la conversion implicite, cette implémentation devient :

```
def merge[A](xs: List[A], ys: List[A]) (implicit c: A => Ordered[A]): List[A] =
  if (xs.isEmpty) ys
  else if (ys.isEmpty) xs
  else if (c(xs.head) < ys.head) xs.head :: merge(xs.tail, ys)
  else if ys.head :: merge(xs, ys.tail)(c)
```

Les deux dernières lignes de cette définition illustrent deux utilisations différentes du paramètre implicite `c`. Dans l'avant-dernière ligne, il est appliqué dans une conversion alors que, dans la dernière ligne, il est passé comme paramètre implicite à l'appel récursif.

Chapitre 16. Inférence de type Hindley/Milner

Ce chapitre présente les types de données de Scala et le pattern matching en développant un système d'inférence de type à la Hindler/Milner (Robin Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences, 17:348–375, Dec 1978.). Le langage source pour l'inférenceur de type est Mini-ML, du lambda-calcul avec une instruction `let`. Les arbres syntaxiques abstraits pour Mini-ML sont représentés par les types de données suivants :

```
abstract class Term {}
case class Var(x: String) extends Term {
  override def toString = x
}
case class Lam(x: String, e: Term) extends Term {
  override def toString = "(\\" + x + "." + e + ")"
}
case class App(f: Term, e: Term) extends Term {
  override def toString = "(" + f + " " + e + ")"
}
case class Let(x: String, e: Term, f: Term) extends Term {
  override def toString = "let " + x + " = " + e + " in " + f
}
```

Il y a donc quatre constructeurs d'abres : `Var` pour les variables, `Lam` pour les abstractions de fonctions, `App` pour l'application des fonctions et `Let` pour les expressions `let`. Chaque case classe redéfinit la méthode `toString` de la classe `Any` de sorte que les termes s'affichent de façon lisible.

Nous définissons ensuite les types qui sont calculés par le système d'inférence :

```
sealed abstract class Type {}
case class Tyvar(a: String) extends Type {
  override def toString = a
}
case class Arrow(t1: Type, t2: Type) extends Type {
  override def toString = "(" + t1 + "->" + t2 + ")"
}
case class Tycon(k: String, ts: List[Type]) extends Type {
  override def toString =
    k + (if (ts.isEmpty) "" else ts.mkString("[", ",", "]"))
}
```

Il y a trois constructeurs de type : `Tyvar` pour les variables de type, `Arrow` pour les types des fonctions et `Tycon` pour les constructeurs de types comme `Boolean` ou `List`. Ce dernier prend en paramètre la liste de leurs paramètres de type – elle est vide pour les constantes de type comme `Boolean`. Les constructeurs de type implémentent également la méthode `toString` pour que les types s'affichent correctement.

Notez que `Type` est une classe `sealed`, ce qui signifie qu'on ne peut pas l'étendre avec des sous-classes ou des constructeurs de type en dehors de la suite de définitions dans lequel il est défini. `Type` est donc un type algébrique *fermé* qui n'a que trois alternatives. `Term`, au contraire, est un type algébrique *ouvert* pour lequel on pourra définir d'autres alternatives.

Les parties principales de l'inférenceur de type sont contenues dans l'objet typeInfer. Nous définissons d'abord une fonction utilitaire qui crée de nouvelles variables de type :

```
object typeInfer {
  private var n: Int = 0
  def newTyvar(): Type = { n += 1; Tyvar("a" + n)
}
```

Nous définissons ensuite une classe pour les substitutions. Une substitution est une fonction idempotente des variables de types vers les types. Elle fait correspondre un nombre fini de variables de types à des types et ne modifie pas les autres variables de types. La signification d'une substitution est étendue à une correspondance point à point des types vers les types.

```
abstract class Subst extends Function1[Type,Type] {

  def lookup(x: Tyvar): Type

  def apply(t: Type): Type = t match {
    case tv @ Tyvar(a) => val u = lookup(tv); if (t == u) t else apply(u)
    case Arrow(t1, t2) => Arrow(apply(t1), apply(t2))
    case Tycon(k, ts) => Tycon(k, ts map apply)
  }

  def extend(x: Tyvar, t: Type) = new Subst {
    def lookup(y: Tyvar): Type = if (x == y) t else Subst.this.lookup(y)
  }
  val emptySubst = new Subst { def lookup(t: Tyvar): Type = t }
}
```

Les substitutions sont représentées par des fonctions de type `Type => Type`. Pour ce faire, la classe `Subst` hérite du type de fonction unaire `Function1[Type, Type]` (la classe hérite du type fonction sous la forme d'un mixin plutôt que comme une superclasse directe car, dans l'implémentation actuelle de Scala, le type `Function1` est une interface Java – il ne peut donc pas être utilisé comme superclasse d'une autre classe). Pour être une instance de ce type, une substitution `s` doit implémenter une méthode `apply` prenant un `Type` comme paramètre et renvoyant un autre `Type` comme résultat. Une application de fonction `s(t)` est alors interprétée comme `s.apply(t)`

Dans la classe `Subst`, la méthode `lookup` est abstraite. Il existe deux formes concrètes de substitutions qui diffèrent par leur implémentation de cette méthode. La première est définie par la valeur `emptySubst`, la seconde par la méthode `extend` de la classe `Subst`.

Le type suivant définit les schémas de types, qui sont formés d'un type et d'une liste de noms de variables de types qui apparaissent avec des quantificateurs universels dans le schéma. Le schéma $\forall a \forall b.a \rightarrow b$, par exemple, serait représenté sous la forme suivante dans le vérificateur de type :

```
TypeScheme(List(Tyvar("a"), Tyvar("b")), Arrow(Tyvar("a"), Tyvar("b")))
```

La définition de classe des schémas de types n'a pas de clause `extends`, ce qui signifie que ces schémas sont des classes qui héritent directement de `AnyRef`. Bien qu'il n'y ait qu'un moyen possible de contruire un schéma de type, nous avons choisi d'utiliser une case classe car ceci permet de décomposer simplement une instance en ses différentes parties :

```
case class TypeScheme(tyvars: List[Tyvar], tpe: Type) {
  def newInstance: Type = {
    (emptySubst /: tyvars) ((s, tv) => s.extend(tv, newTyvar())) (tpe)
  }
}
```

Les objets schémas de types disposent d'une méthode `newInstance` qui renvoie le type contenu dans le schéma après que toutes les variables de types quantifiée universellement ait été renommées en nouvelles variables. L'implémentation de cette méthode accumule (avec `/:`) les variables de types du schéma avec une opération qui étend une substitution `s` donnée en renommant une variable de type `tv` donnée en nouvelle variable de type. La substitution ainsi obtenue renomme tous les variables de types du schéma. Elle est ensuite appliquée à la partie type du schéma.

Le dernier type dont nous avons besoin dans l'inférenceur de type est `Env`, un type pour les environnements qui associe des noms de variables à des schémas de type. Il est représenté par l'alias de type `Env` dans le module typeInfer :

```
type Env = List[(String, TypeScheme)]
```

Il y a deux opérations possibles sur les environnements. La fonction `lookup` renvoie le schéma de type associé à un nom ou **null** si le nom n'est pas enregistré dans l'environnement :

```
def lookup(env: Env, x: String): TypeScheme = env match {
  case List() => null
  case (y, t) :: env1 => if (x == y) t else lookup(env1, x)
}
```

La fonction `gen` transforme un type donné en un schéma de type en quantifiant toutes les variables de types qui sont libres dans le type mais pas dans l'environnement :

```
def gen(env: Env, t: Type): TypeScheme =
  TypeScheme(tyvars(t) diff tyvars(env), t)
```

L'ensemble des variables de types libres d'un type est simplement l'ensemble de toutes les variables de types qui apparaissent dans ce type. On le représente ici comme une liste de variables de types :

```
def tyvars(t: Type): List[Tyvar] = t match {
  case tv @ Tyvar(a) => List(tv)
  case Arrow(t1, t2) => tyvars(t1) union tyvars(t2)
  case Tycon(k, ts) => (List[Tyvar]() /: ts) ((tvs, t) => tvs union tyvars(t))
}
```

La syntaxe `tv @ ...` du premier motif introduit une variable qui est liée au motif qui suit l'arobase. Notez également que le paramètre de type explicite `[Tyvar]` dans l'expression de la troisième clause est nécessaire pour que l'inférence de type locale puisse fonctionner.

L'ensemble des variables de types libres d'un schéma de type est l'ensemble des variables de types libres de son composant type, exception faite des variables de types quantifiées :

```
def tyvars(ts: TypeScheme): List[Tyvar] =
  tyvars(ts.tpe) diff ts.tyvars
```

Enfin, l'ensemble des variables de types libres d'un environnement est l'union des variables de types libres de tous les schémas de types qui y sont enregistrés :

```
def tyvars(env: Env): List[Tyvar] =
   (List[Tyvar]() /: env) ((tvs, nt) => tvs union tyvars(nt._2))
```

Avec Hindley/Milner, l'unification est une opération essentielle de la vérification de type. L'unification calcule une substitution pour que deux types donnés soient égaux (cette substitution est appelée *unificateur*). La fonction `mgu` calcule l'unificateur le plus général de deux types `t` et `u` donnés sous une substitution `s` existante. Elle renvoie donc la substitution `s'` la plus générale qui étend `s` et qui est telle que les types `s(t)` et `s(u)` sont égaux :

```
def mgu(t: Type, u: Type, s: Subst): Subst =
  (s(t), s(u)) match {
    case (Tyvar(a), Tyvar(b)) if (a == b) =>
      s
    case (Tyvar(a), _) if !(tyvars(u) contains a) =>
      s.extend(Tyvar(a), u)
    case (_, Tyvar(a)) =>
      mgu(u, t, s)
    case (Arrow(t1, t2), Arrow(u1, u2)) =>
      mgu(t1, u1, mgu(t2, u2, s))
    case (Tycon(k1, ts), Tycon(k2, us)) if (k1 == k2) =>
```

```
      (s /: (ts zip us)) ((s, tu) => mgu(tu._1, tu._2, s))
    case _ =>
      throw new TypeError("cannot unify " + s(t) + " with " + s(u))
}
```

La fonction `mgu` lance une exception `TypeError` si aucune substitution unificatrice n'existe. Ceci peut arriver lorsque les deux types ont des constructeurs différents ou parce qu'une variable de type est unifiée avec un type qui contient la variable de type elle-même. Ces exceptions sont modélisées ici comme des instances des case classes qui héritent de la classe `Exception` prédéfinie.

```
case class TypeError(s: String) extends Exception(s) {}
```

La tâche principale du vérificateur de type est implémentée par la fonction `tp` qui prend en paramètres un environnement `env`, un terme `e`, un proto-type `t` et une substitution existante `s`, et qui renvoie une substitution `s'` qui étend `s` et qui transforme `s (env) e : s (t)` en jugement de type dérivable selon les règles de dérivation du système de typage Hindley/Milner. Si cette substitution n'existe pas, la fonction lève l'exception `TypeError` :

```
def tp(env: Env, e: Term, t: Type, s: Subst): Subst = {
  current = e
  e match {
    case Var(x) =>
      val u = lookup(env, x)
      if (u == null) throw new TypeError("undefined: " + x)
      else mgu(u.newInstance, t, s)

    case Lam(x, e1) =>
      val a, b = newTyvar()
      val s1 = mgu(t, Arrow(a, b), s)
      val env1 = {x, TypeScheme(List(), a)} :: env
      tp(env1, e1, b, s1)

    case App(e1, e2) =>
      val a = newTyvar()
      val s1 = tp(env, e1, Arrow(a, t), s)
      tp(env, e2, a, s1)

    case Let(x, e1, e2) =>
      val a = newTyvar()
      val s1 = tp(env, e1, a, s)
      tp({x, gen(env, s1(a))} :: env, e2, t, s1)

    }
}
var current: Term = null
```

Pour faciliter la détection des erreurs, la fonction `tp` stocke le sous-terme analysé dans la variable `current` : si la vérification de type échoue avec une exception `TypeError`, cette variable contiendra le sous-terme qui a posé problème.

La dernière fonction du module d'inférence de type, `typeOf`, est une façade simplifiée de `tp`. Elle calcule le type d'un terme `e` dans un environnement `env`. Pour ce faire, elle crée une nouvelle variable `a`, calcule une substitution de type qui transforme `env e : a` en jugement de type dérivable et renvoie le résultat de l'application de la substitution à `a` :

```
  def typeOf(env: Env, e: Term): Type = {
    val a = newTyvar()
    tp(env, e, a, emptySubst)(a)
  }
} // fin de typeInfer
```

Pour appliquer l'inférenceur de type, il est pratique de disposer d'un environnement prédéfini contenant des liaisons pour les constantes souvent utilisées. Le module `predefined` définit par conséquent un environnement `env` qui contient des liaisons pours les types booléens, numériques et listes, ainsi que certaines opérations primitives définies sur ces types. Il définit également un opérateur de point fixe `fix` permettant de représenter la récursion :

```
object predefined {
  val booleanType = Tycon("Boolean", List())
  val intType = Tycon("Int", List())
  def listType(t: Type) = Tycon("List", List(t))

  private def gen(t: Type): typeInfer.TypeScheme = typeInfer.gen(List(), t)
  private val a = typeInfer.newTyvar()
  val env = List(
    {"true", gen(booleanType)},
    {"false", gen(booleanType)},
    {"if", gen(Arrow(booleanType, Arrow(a, Arrow(a, a))))},
    {"zero", gen(intType)},
    {"succ", gen(Arrow(intType, intType))},
    {"nil", gen(listType(a))},
    {"cons", gen(Arrow(a, Arrow(listType(a), listType(a))))},
    {"isEmpty", gen(Arrow(listType(a), booleanType))},
    {"head", gen(Arrow(listType(a), a))},
    {"tail", gen(Arrow(listType(a), listType(a)))},
    {"fix", gen(Arrow(Arrow(a, a), a))}
  )
}
```

Voici un exemple d'utilisation de l'inférenceur de type. Définissons une fonction `showType` qui renvoie le type d'un terme donné, calculé dans l'environnement `predefined.env` :

```
object testInfer {
  def showType(e: Term): String =
    try {
      typeInfer.typeOf(predefined.env, e).toString
    } catch {
      case typeInfer.TypeError(msg) =>
        "\n cannot type: " + typeInfer.current +
        "\n reason: " + msg
    }
```

L'application

```
> testInfer.showType(Lam("x", App(App(Var("cons"), Var("x")), Var("nil"))))
```

donnerait donc la réponse suivante :

```
> (a6->List[a6])
```

Exercise 16–0–1 : Étendez l'inférenceur de type Mini-ML avec une construction `letrec` permettant de définir des fonctions récursives et dont la syntaxe est :

```
letrec ident "=" term in term
```

Le typage de `letrec` est identique à `let` sauf que l'identificateur ainsi défini est visible dans l'expression. Avec `letrec`, la fonction `length` des listes peut être définie de la façon suivante :

```
letrec length = \xs.
  if (isEmpty xs)
    zero
    (succ (length (tail xs)))
in ...
```

Chapitre 17. Abstractions pour les programmes concurrents

Ce chapitre passe en revue les patrons de conceptions les plus courants pour la programmation concurrente et montre leur implémentation en Scala.

Signaux et moniteurs

Exemple 17–1–1 Un *moniteur* est l'outil de base pour garantir l'exclusion mutuelle des processus en Scala. Chaque instance de la classe `AnyRef` peut servir de moniteur grâce aux méthodes suivantes :

```
def synchronized[A] (e: => A): A
def wait()
def wait(msec: Long)
def notify()
def notifyAll()
```

La méthode `synchronized` exécute le traitement qui lui est passé en paramètre en exclusion mutuelle : à un instant précis, il ne peut y avoir qu'un seul thread qui exécute le paramètre `synchronized` d'un moniteur donné.

Les threads peuvent être suspendus dans un moniteur dans l'attente d'un signal. Les threads qui appellent la méthode `wait` attendent jusqu'à ce qu'un autre thread appelle ensuite la méthode `notify` du même objet. Les appels à `notify` alors qu'aucun thread n'attend le signal sont ignorés.

Il existe également une forme de `wait` avec délai, qui suspend le thread en attente d'un signal ou pendant le délai indiqué (en millisecondes). En outre, la méthode `notifyAll` débloque tous les threads qui attendent le signal. En Scala, ces méthodes, ainsi que la classe `Monitor`, sont des primitives – elles sont implémentées à partir du système d'exécution sous-jacent.

Le plus souvent, un thread attend qu'une certaine condition soit vérifiée. Si celle-ci le l'est pas lors de l'appel à `wait`, le thread se bloque jusqu'à ce que autre thread établisse cette condition. C'est cet autre thread qui est responsable du réveil des threads bloqués via un appel à `notify` ou `notifyAll`. Notez cependant qu'il n'est pas garanti qu'un thread bloqué redémarre immédiatement après l'appel à `notify` : cet appel pourrait réveiller d'abord un autre thread qui pourrait invalider à nouveau la condition. La forme correcte de l'attente d'une condition `C` consiste donc à utiliser une boucle `while` :

```
while (!C) wait()
```

Pour illustrer l'utilisation des moniteurs, prenons l'exemple d'une classe tampon de taille bornée :

```
class BoundedBuffer[A](N: Int) {
  var in = 0, out = 0, n = 0
  val elems = new Array[A](N)

  def put(x: A) = synchronized {
    while (n >= N) wait()
    elems(in) = x ; in = (in + 1) % N ; n = n + 1
    if (n == 1) notifyAll()
  }
```

```
  def get: A = synchronized {
    while (n == 0) wait()
    val x = elems(out) ; out = (out + 1) % N ; n = n - 1
    if (n == N - 1) notifyAll() x
  }
}
```

Voici un programme utisant un tampon borné pour communiquer entre un processus producteur et un processus consommateur :

```
import scala.concurrent.ops._
...
  val buf = new BoundedBuffer[String](10)
  spawn { while (true) { val s = produceString ; buf.put(s) } }
  spawn { while (true) { val s = buf.get ; consumeString(s) } }
}
```

La méthode `spawn` crée un thread qui exécute l'expression contenue dans son paramètre. Cette méthode est définie de la façon suivante dans l'objet `concurrent.ops` :

```
def spawn(p: => Unit) {
  val t = new Thread() { override def run() = p }
  t.start()
```

Variables de synchronisation

Une variable synchronisée (ou "syncvar") fournit les opérations `get` et `put` pour lire et modifier le contenu de la variable. Les opération `get` sont bloquantes tant que la variable n'a pas été définie. L'opération `unset` réinitialise la variable et la place dans l'état indéfini.

Voici l'implémentation standard des variables synchronisées :

```
package scala.concurrent class SyncVar[A] {
  private var isDefined: Boolean = false
  private var value: A = _
  def get = synchronized {
    while (!isDefined) wait()
    value
  }
  def set(x: A) = synchronized {
    value = x; isDefined = true; notifyAll()
  }
  def isSet: Boolean = synchronized {
    isDefined
  }
  def unset = synchronized {
    isDefined = false
  }
}
```

Futures

Une *future* est une valeur qui est traitée en parallèle d'un autre thread client et qui pourra être utilisé par ce dernier dans un temps futur. Les futures permettent de mieux utiliser les ressources de traitement parallèle. Leur utilisation classique est de la forme :

```
import scala.concurrent.ops._
...
  val x = future(calculTrèsLong)
  unAutreCalculTrèsLong
  val y = f(x()) + g(x())
}
```

La méthode `future` est définie de la façon suivante dans l'objet `scala.concurrent.ops` :

```
def future[A](p: => A): Unit => A = {
  val result = new SyncVar[A]
  fork { result.set(p) }
  (() => result.get)
}
```

Cette méthode prend en paramètre un traitement `p` à réaliser. Le type de ce traitement est quelconque – il est représenté par le paramètre de type `A` de `future`. La méthode utilise variable synchronisée `result` pour stocker le résultat du traitement. Puis, elle crée un thread pour effectuer le traitement et affecter son résultat à `result`. En parallèle de ce thread, la fonction renvoie une fonction anonyme de type `A` qui, lorsqu'elle est appelée, attend le résultat du traitement et le renvoie. Lorsque cette fonction sera appelée de nouveau, elle renverra immédiatement le résultat déjà calculé.

Traitements parallèles

L'exemple suivant présente la fonction `par` qui prend en paramètre une paire de traitements et qui renvoie une paire contenant leurs résultats. Ces deux traitements s'effectuent en parallèle.

Cette fonction est définie de la façon suivante dans l'objet `scala.concurrent.ops` :

```
def par[A, B](xp: => A, yp: => B): (A, B) = {
  val y = new SyncVar[B]
  spawn { y set yp }
  (xp, y.get)
}
```

Au même endroit est définie la fonction `replicate` qui exécute plusieurs fois le même traitement en parallèle. Chaque réplique reçoit en paramètre un nombre entier qui l'identifie :

```
def replicate(start: Int, end: Int)(p: Int => Unit) {
  if (start == end)
    ()
  else if (start + 1 == end)
    p(start)
  else {
    val mid = (start + end) / 2
    spawn { replicate(start, mid)(p) }
    replicate(mid, end)(p)
  }
}
```

La fonction suivante utilise `replicate` pour traiter en parallèle tous les éléments d'un tableau :

```
def parMap[A,B](f: A => B, xs: Array[A]): Array[B] = {
  val results = new Array[B](xs.length)
  replicate(0, xs.length) { i => results(i) = f(xs(i)) }
  results
}
```

Sémaphores

Les *verrous* (ou sémaphores) sont un mécanisme classique de synchronisation des processus. Un verrou a deux actions atomiques : `acquire` et `release`. Voici l'implémentation d'un verrou en Scala :

```
package scala.concurrent

class Lock {
  var available = true
  def acquire = synchronized {
    while (!available) wait()
    available = false
  }
  def release = synchronized {
    available = true
    notify()
  }
}
```

Lecteurs/rédacteurs

Une forme de synchronisation plus complexe fait la distinction entre les *lecteurs* qui accèdent à une ressource commune sans la modifier et les *rédacteurs* qui peuvent à la fois y accéder et la modifier. Pour synchroniser les lecteurs et les rédacteurs, il faut implémenter les opérations `startRead`, `startWrite`, `endRead` et `endWrite` telles que :

- il peut y avoir plusieurs lecteurs concurrents
- il ne peut y avoir qu'un seul rédacteur à la fois
- les demandes d'écriture en attente ont priorité sur les demandes de lecture, mais ne préemptent pas les opérations de lecture en cours.

L'implémentation suivante d'un verrou lecteurs/rédacteurs repose sur le concept de boîte aux lettres (voir la section 17–10) :

```
import scala.concurrent._

class ReadersWriters {
  val m = new MailBox
  private case class Writers(n: Int), Readers(n: Int) { m send this }
  Writers(0); Readers(0)
  def startRead = m receive {
    case Writers(n) if n == 0 => m receive {
       case Readers(n) => Writers(0); Readers(n+1)
    }
  }
  def startWrite = m receive {
    case Writers(n) =>
      Writers(n+1)
      m receive { case Readers(n) if n == 0 => }
  }
  def endRead = m receive {
    case Readers(n) => Readers(n-1)
  }
  def endWrite = m receive {
    case Writers(n) => Writers(n-1); if (n == 0) Readers(0)
  }
}
```

Canaux asynchrones

Les canaux asynchrones sont un mécanisme fondamental de communication inter-processus. Leur implémentation utise une classe simple pour les listes chaînées :

```
class LinkedList[A] {
  var elem: A = _
  var next: LinkedList[A] = null
}
```

Pour faciliter l'insertion et la suppression des éléments dans les listes chaînées, chaque référence de la liste pointe vers le noeud qui précède le noeud qui, conceptuellement, forme le debut de la liste. Les listes chaînées vides commencent par un noeud factice dont le successeur est **null**.

La classe `Channel` utilise une liste chaînée pour stocker les données qui ont été envoyées mais non encore lue. À l'autre extrémité, les threads qui veulent lire dans un canal vide enregistrent leur présence en incrémentant le champ `nreaders` et attendent d'être prévenus.

```
package scala.concurrent

class Channel[A] {
  class LinkedList[A] {
    var elem: A = _
    var next: LinkedList[A] = null
  }
  private var written = new LinkedList[A]
  private var lastWritten = written
  private var nreaders = 0

  def write(x: A) = synchronized {
    lastWritten.elem = x
    lastWritten.next = new LinkedList[A]
    lastWritten = lastWritten.next
    if (nreaders > 0) notify()
  }

  def read: A = synchronized {
    if (written.next == null) {
      nreaders = nreaders + 1; wait(); nreaders = nreaders - 1
    }
    val x = written.elem
    written = written.next
    x
  }
}
```

Canaux synchrones

Voici une implémentation de canaux synchrones, où l'expéditeur d'un message est bloqué tant que son message n'a pas été reçu. Les canaux synchrones n'ont besoin que d'une simple variable pour stocker les messages en transit, mais de trois signaux pour coordonner les processus lecteur et rédacteur.

```
package scala.concurrent

class SyncChannel[A] {
  private var data: A = _
  private var reading = false
  private var writing = false

  def write(x: A) = synchronized {
    while (writing) wait()
```

```
    data = x
    writing = true
    if (reading) notifyAll()
    else while (!reading) wait()
  }

  def read: A = synchronized {
    while (reading) wait()
    reading = true
    while (!writing) wait()
    val x = data
    writing = false
    reading = false
    notifyAll()
    x
   }
}
```

Ouvriers

Voici une implémentation d'un serveur de calculs en Scala. Ce serveur implémente une méthode `future` qui évalue une expression en parallèle de son appelant. À la différence de l'implémentation de la section 17–3, le serveur ne traite les futures qu'avec un nombre prédéfini de threads. Un implémentation possible de ce serveur pourrait, par exemple, exécuter chaque thread sur un processeur différente et ainsi éviter le surcoût inhérent au changement de contexte qui intervient lorsque plusieurs threads doivent se partager le même processeur.

```
import scala.concurrent._, scala.concurrent.ops._

class ComputeServer(n: Int) {

  private abstract class Job {
    type T
    def task: T
    def ret(x: T)
  }

  private val openJobs = new Channel[Job]()

  private def processor(i: Int) {
    while (true) {
      val job = openJobs.read
      job.ret(job.task)
    }
  }

  def future[A](p: => A): () => A = {
    val reply = new SyncVar[A]()
    openJobs.write {
      new Job {
        type T = A
        def task = p
        def ret(x: A) = reply.set(x)
      }
    }
    () => reply.get
  }

 spawn(replicate(0, n) { processor })
}
```

Les expressions à calculer (c'est-à-dire les paramètres passés à `future`) sont écrites dans le canal `openJobs`. Un *job* est un objet avec :

- un type abstrait `T` qui décrit le résultat de l'expression à calculer.

- une méthode `task` sans paramètre, renvoyant un résultat de type `T` qui représente l'expression à calculer.
- une méthode `ret` qui consomme le résultat lorsqu'il a été calculé.

Le serveur de calcul crée `n` processus `processor` au cours de son initialisation. Chacun de ces processus consomme sans fin un job ouvert, évalue la méthode `task` de ce job et passe le résultat à la méthode `ret` du job. La méthode `future` polymorphe crée un nouveau job dont la méthode `ret` est implémentée par une variable de synchronisation `reply` ; elle insère le job dans l'ensemble des jobs ouverts puis attend que la variable de synchronisation correspondante soit appelée.

Cet exemple montre comment utiliser les types abstraits. Le type `T` mémorise le type du résultat d'un job, qui peut varier d'un job à l'autre. Sans les types abstraits, il serait impossible d'implémenter la même classe avec un typage statique et l'utilisateur devrait tester dynamiquement les types et utiliser le transtypage.

Voici un exemple d'utilisation du serveur de calcul pour évaluer l'expression `41 + 1` :

```
object Test with Executable {
  val server = new ComputeServer(1)
  val f = server.future(41 + 1)
  println(f())
}
```

Boîtes aux lettres

Les boîtes aux lettres sont des constructions souples et de haut niveau permettant de synchroniser des processus et de les faire communiquer. Elles permettent d'envoyer et de recevoir des *messages* qui, dans ce contexte, sont des objets quelconques. Le message spécial `TIMEOUT` sert à signaler l'expiration d'un délai :

```
case object TIMEOUT
```

Les boîtes aux lettres ont la signature suivante :

```
class MailBox {
  def send(msg: Any)
  def receive[A](f: PartialFunction[Any, A]): A
  def receiveWithin[A](msec: Long)(f: PartialFunction[Any, A]): A
}
```

L'état d'une boîte aux lettres est consitué d'un multi-set de messages. Les messages sont ajoutés à la boîte par sa méthode `send` et supprimés par la méthode `receive` qui reçoit en paramètre un traitement de message `f` – une fonction partielle des messages vers un type de résultat quelconque. Généralement, cette méthode est implémentée par un pattern matching : elle se bloque jusqu'à ce qu'il y ait un message dans la boîte qui corresponde à l'un des motifs. Le message en question est alors supprimé de la boîte et le thread bloqué est relancé en appliquant le traitement au message. Les envois et les réceptions de messages sont ordonnés dans le temps ; un récepteur `r` n'est appliqué à un message correspondant `m` que s'il n'y a pas d'autre paire {*message*, *récepteur*} antérieure à `m`,`r`.

À titre d'exemple, considérons un tampon à une case :

```
class OnePlaceBuffer {
  private val m = new MailBox              // Boîte aux lettres interne
  private case class Empty, Full(x: Int) // Types des messages gérés
  m send Empty                           // Initialisation
  def write(x: Int) {
    m receive { case Empty => m send Full(x) }
```

```
  }
  def read: Int =
    m receive { case Full(x) => m send Empty; x }
}
```

Voici comment peut être implémentée la classe `MailBox` :

```
class MailBox {
  private abstract class Receiver extends Signal {
    def isDefined(msg: Any): Boolean
    var msg = null
  }
```

Nous définissons une classe interne pour les récepteurs, dotée d'une méthode de test `isDefined`, qui indique si le récepteur est défini pour un message donné. Cette classe hérite de la classe `Signal` la méthode `notify` qui sert à réveiller un thread récepteur. Lorsque ce dernier est réveillé, le message auquel il doit s'appliquer est stocké dans la variable `msg` de `Receiver`.

```
  private val sent = new LinkedList[Any]
  private var lastSent = sent
  private val receivers = new LinkedList[Receiver]
  private var lastReceiver = receivers
```

La classe `MailBox` gère deux listes chaînées, une pour les messages envoyés mais non encore consommés, l'autre pour les récepteurs en attente.

```
  def send(msg: Any) = synchronized {
    var r = receivers, r1 = r.next
    while (r1 != null && !r1.elem.isDefined(msg)) {
      r = r1; r1 = r1.next
    }
    if (r1 != null) {
      r.next = r1.next; r1.elem.msg = msg; r1.elem.notify
    } else {
      lastSent = insert(lastSent, msg)
    }
  }
```

La méthode `send` commence par tester si un récepteur en attente peut s'appliquer au message envoyé. En ce cas, ce récepteur est prévenu par `notify`. Sinon, le message est ajouté à la fin de la liste des messages envoyés.

```
  def receive[A](f: PartialFunction[Any, A]): A = {
    val msg: Any = synchronized {
      var s = sent, s1 = s.next
      while (s1 != null && !f.isDefinedAt(s1.elem)) {
        s = s1; s1 = s1.next
       }
       if (s1 != null) {
         s.next = s1.next; s1.elem
       } else {
         val r = insert(lastReceiver, new Receiver {
           def isDefined(msg: Any) = f.isDefinedAt(msg)
         })
         lastReceiver = r
         r.elem.wait()
         r.elem.msg
       }
    }
   f(msg)
  }
```

La méthode `receive` teste d'abord si la fonction `f` de traitement du message peut s'appliquer à un message qui a déjà été envoyé mais qui n'a pas encore été consommé. En ce cas, le thread se poursuit en appliquant `f` au message. Sinon, la méthode crée un nouveau récepteur et le place dans la liste des récepteurs, puis le thread se bloque et attend d'être réveillé par ce récepteur. Lorsqu'il se réveille, le thread applique `f` au message qui était stocké dans le récepteur. La méthode `insert` des listes chaînées est définie de la façon suivante :

```
def insert(l: LinkedList[A], x: A): LinkedList[A] = {
  l.next = new LinkedList[A]
  l.next.elem = x
  l.next.next = l.next
  l
}
```

La classe `MailBox` définit également une méthode `receiveWithin` qui ne bloque le thread que pendant le temps maximal indiqué. Si aucun message n'est reçu dans cet intervalle de temps (exprimé en millisecondes), le traitement du message est débloqué par le message spécial `TIMEOUT`. L'implémentation de `receiveWithin` ressemble beaucoup à celle de `receive` :

```
  def receiveWithin[A](msec: Long)(f: PartialFunction[Any, A]): A = {
    val msg: Any = synchronized {
      var s = sent, s1 = s.next
      while (s1 != null && !f.isDefinedAt(s1.elem)) {
        s = s1; s1 = s1.next
      }
      if (s1 != null) {
      s.next = s1.next; s1.elem
      } else {
        val r = insert(lastReceiver, new Receiver {
          def isDefined(msg: Any) = f.isDefinedAt(msg)
        })
        lastReceiver = r
        r.elem.wait(msec)
        if (r.elem.msg == null) r.elem.msg = TIMEOUT
        r.elem.msg
      }
    }
    f(msg)
  }
} // Fin de MailBox
```

Les seules différences entre les deux méthodes sont l'appel temporisé à `wait` et l'instruction qui le suit.

Acteurs

Le chapitre 3 a présenté un exemple de programme implémentant un service d'enchères. Ce service reposait sur des processus acteurs de haut niveau qui fonctionnaient en inspectant les messages de leurs boîtes aux lettres à l'aide du pattern matching. Le paquetage `scala.actors` contient une implémentation améliorée et optimisée des acteurs, mais nous n'en présenterons ici qu'une version simplifiée.

Un acteur simplifié est simplement un thread qui utilise une boîte aux lettres pour communiquer. Il peut donc être vu comme un mélange de la classe `Thread` standard de Java et de la classe `MailBox`. Nous redéfinissons également la méthode `run` de la classe `Thread` pour qu'elle ait le comportement de l'acteur défini par sa méthode `act`. La méthode `!` appelle simplement la méthode `send` de la classe `MailBox` :

```
abstract class Actor extends Thread with MailBox {
  def act(): Unit
  override def run(): Unit = act()
  def !(msg: Any) = send(msg)
}
```