SCIENCES SUP

Cours et exercices

Licence • IUT • Écoles d'ingénieurs

PROGRAMMATION

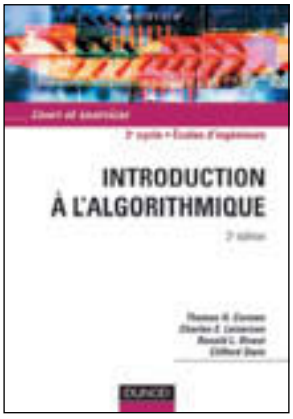
Concepts, techniques et modèles

Peter Van Roy
Seif Haridi

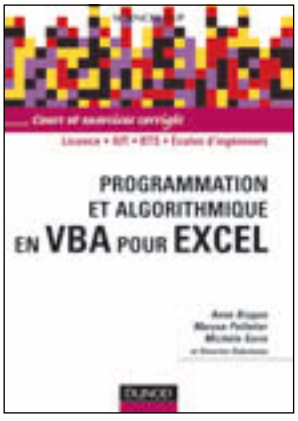
DUNOD

PROGRAMMATION

Concepts, techniques et modèles



Introduction à l'algorithmique
2e édition
Thomas Cormen, Charles Leiserson,
Ronald Rivest, Clifford Stein
1176 pages
Dunod, 2003



Programmation et algorithmique en VBA pour Excel
Anne Brygoo, Maryse Pelletier,
Michèle Soria, Séverine Dubuisson
240 pages
Dunod, 2007

PROGRAMMATION

Concepts, techniques et modèles

Peter Van Roy
Professeur à l'Université catholique
de Louvain à Louvain-la-Neuve

Seif Haridi
Professeur au Royal Institute of Technologie (Suède)
et « chief scientist » au Swedish Institute of Computer Science

DUNOD

Pour l'édition originale anglaise :
Concepts, techniques, and Models of Computer Programming

Illustration de couverture : *digitalvision*®

ISBN 978-2-10-051196-9

Table des matières

Avant-propos

D'une certaine manière, un langage de programmation est semblable au langage humain : il favorise certaines métaphores, images et façons de penser.
— Mindstorms : Children, Computers, and Powerful Ideas, Seymour Papert (1980)

Ce livre est un cours de programmation complet. Il couvre les trois principaux paradigmes de programmation : la programmation déclarative, la programmation concurrente dataflow et la programmation orientée objet, dans un cadre uniforme avec une panoplie de techniques pratiques de développement de programmes. Il propose une sémantique simple et complète permettant de comprendre tous les concepts sans le flou que l'on trouve presque toujours dans les premiers cours sur la programmation. Spécialement conçue pour les programmeurs, cette sémantique contient les concepts utiles sans sacrifier la rigueur. Elle permet une compréhension profonde de tous les concepts et de toutes les techniques. Pour beaucoup d'étudiants, en particulier pour ceux qui ne se spécialiseront pas en informatique, ce cours est l'unique occasion de voir une sémantique rigoureuse pour un langage de programmation.

Vous trouverez ici toute la matière de mon cours de deuxième année à l'Université catholique de Louvain (UCL), à Louvain-la-Neuve, pour les étudiants-ingénieurs dans tous les domaines (environ 250 étudiants par an). C'est un cours de deuxième niveau de programmation qui présuppose que les étudiants ont déjà suivi un premier cours d'introduction à la programmation, par exemple avec Java. Le langage du premier cours importe peu ; Java est un bon choix parce qu'il est populaire (les étudiants doivent le voir de toute façon dans leur cursus) et parce que le premier cours ne va pas assez loin pour révéler ses faiblesses.

Mon cours à l'UCL se donne sur une période de 15 semaines avec une séance magistrale et une séance de travaux pratiques par semaine, de deux heures chacune. Pour évaluer les connaissances des étudiants, il y a une interrogation facultative vers le milieu du cours, un projet de programmation obligatoire vers la fin et un examen final. Je donne ce cours en écrivant, en exécutant et parfois en déboguant des petits programmes devant les étudiants.

Le livre est accompagné d'un ensemble d'outils :

– Le **Labo interactif** coédité avec ScienceActive (voir page xiii). Il contient de nombreux exemples de programmes et intègre le système Mozart complet. Vous pouvez modifier et exécuter tous les exemples sans le quitter. Il constitue une bon complément pour le livre.
– Un **site Web** qui contient les éléments du cours : de nombreux transparents, des travaux pratiques, des examens, un lexique bilingue français/anglais des termes techniques et beaucoup d'autres suppléments. L'adresse du site Web est :

  `www.ctm.info.ucl.ac.be/fr`

  Nous vous conseillons particulièrement de consulter le lexique, parce que les dictionnaires bilingues de l'informatique sont très pauvres en vocabulaire de la science de l'informatique.
– Le **Mozart Programming System** [69], une plate-forme de développement qui peut exécuter tous les programmes du livre et qui est disponible en logiciel libre sur le site Web suivant :

  `www.mozart-oz.org`

  Mozart implémente le langage Oz. Nous avons choisi Oz car il contient tous les concepts des paradigmes dans un cadre uniforme, il a une sémantique simple et une implémentation de haute qualité, à savoir Mozart.

Selon notre expérience, les trois principaux paradigmes de programmation sont la programmation déclarative (fonctionnelle), la programmation concurrente dataflow (avec des flots) et la programmation avec abstractions de données (l'état et les objets). La figure 0.1 présente ces trois paradigmes et les chapitres leur correspondant. Le chapitre 1 est une introduction à tous les concepts de ces paradigmes.

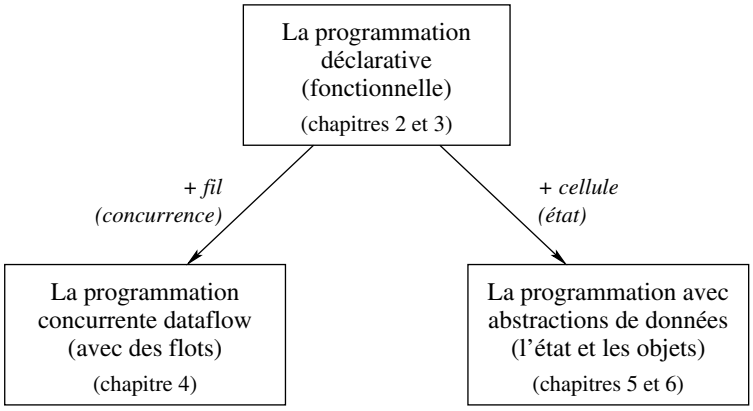


Figure 0.1 Les trois principaux paradigmes de programmation.

Vous trouverez ici une sémantique opérationnelle de tous ces paradigmes dans un cadre uniforme permettant de bien visualiser leurs relations les uns avec les autres. La sémantique est une machine abstraite qui représente l'exécution comme un mécanisme d'horlogerie. Les relations entre les paradigmes sont simples. Le paradigme de base est la programmation déclarative. Chacun des deux autres contient un concept de plus que la programmation déclarative : le fil (« *thread* ») pour la programmation concurrente dataflow et la cellule (variable affectable, « *cell* ») pour la programmation avec abstractions de données. Tout le reste, ce sont des techniques de programmation !

Ce livre est le résultat de nombreuses années de recherche et d'enseignement. Du côté recherche, il intègre les travaux d'une communauté active sur les langages de programmation depuis le début des années 1990. Je tiens à mentionner Gert Smolka [38, 40, 41, 81, 82, 84, 85], Hassan Aït-Kaci [3, 6, 5, 4], Seif Haridi [45, 36, 46] et Sverker Janson [45, 36, 46, 44], les grands pionniers, mais il y a beaucoup d'autres contributeurs ; je n'oublie pas les étudiants et les assistants qui ont subi nos cours pendant de longues années. Ce cours est la traduction d'environ un tiers du livre « *Concepts, Techniques, and Models of Computer Programming* » (CTM), que j'ai écrit avec Seif Haridi [97, 95, 86, 33, 32, 37] et dont j'ai tenu à réaliser moi-même la traduction et l'adaptation en français. Je tiens à remercier Raphaël Collet, Yves Jaradin, Stéphanie Landrain et Chantal Poncin pour leur aide dans la traduction. Si vous voulez aller plus loin dans la découverte de la programmation et des paradigmes, je vous conseille de lire CTM.

Si vous avez des remarques ou si vous trouvez des erreurs, n'hésitez pas à me contacter par courriel ou à contacter Seif Haridi. Nous tenons à remercier toutes les personnes qui nous ont aidés dans la réalisation de ce livre. Seif Haridi voudrait spécialement remercier ses parents Ali et Amina et sa famille Eeva, Rebecca et Alexander. Je voudrais également remercier particulièrement mes parents Frans et Hendrika et ma famille Marie-Thérèse, Johan et Lucile, pour m'avoir aidé et soutenu.

Peter Van Roy
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve, Belgique

Avril 2007

Labo interactif

La programmation s'apprend par la pratique, la simple lecture de livres ne suffit pas. Pour soutenir cet apprentissage, ce livre contient un grand nombre de programmes et de fragments de programmes qui peuvent tous être exécutés. Pour faciliter l'exécution de ces programmes, **Yves Jaradin et Peter Van Roy** ont conçu le *Labo interactif*, publié par ScienceActive. Au lieu d'utiliser directement le système Mozart, ce qui demande de connaître son interface et d'entrer tous les exemples au clavier, il est beaucoup plus simple d'utiliser le *Labo interactif*. Il contient tous les exemples dans un environnement interactif qui permet non seulement de les exécuter immédiatement mais également de les modifier et de les enregistrer pour une exécution ultérieure. L'apprentissage de la programmation à l'aide de ces exemples se trouve ainsi facilité. Cet outil pédagogique innovant simplifie, accélère et approfondit, de façon intuitive, la compréhension et la résolution de problèmes complexes. Il est conçu pour soutenir l'auto-apprentissage de la programmation en accompagnement du livre. Il contient beaucoup d'exemples de codes supplémentaires qui ne se trouvent pas dans le livre, ainsi qu'un glossaire complet de tous les termes techniques auxquels on peut accéder en un simple clic.

LE PUBLIC VISÉ

Pour l'étudiant, c'est l'outil idéal pour apprendre, réviser et explorer la programmation, en interaction avec la lecture du livre.

Pour l'enseignant, le *Labo interactif* est un outil proactif essentiel pour les cours et les séances de travaux pratiques, permettant de modifier un exemple tiré directement du livre et de l'exécuter sur le champ ou de comparer et de modifier immédiatement différents exemples.

Pour le développeur professionnel, le *Labo interactif* permet de rafraîchir et d'approfondir ses connaissances de la manière la plus facile qui soit.

LA NAVIGATION ET L'EXPLORATION

Vous trouverez dans le *Labo interactif*, les mêmes titres de chapitres et de sections et les mêmes exemples de codes que dans le livre, ainsi que beaucoup d'autres exemples

Figure 0.2 Copie d'écran du *Labo interactif*.

apparentés mais pas les explications détaillées du livre. Chaque exemple est accompagné d'un commentaire succinct pour expliquer ce qu'il fait. Vous pouvez naviguer dans cette structure et ouvrir plusieurs fenêtres sur les exemples et leur exécution.

LA PUISSANCE ET LA FACILITÉ

Conçu sous forme électronique pour une utilisation interactive, cet outil s'affranchit des contraintes d'apprentissage des techniques de programmation par la lecture de livres. Avec le *Labo interactif* vous avez à votre disposition un environnement de travail, d'exercice et d'expérimentation équipé et enrichi de toutes les fonctionnalités du système Mozart, renforçant l'apprentissage effectif et immédiat des savoirs dispensés par le présent livre. Vous pouvez exécuter et modifier en temps réel tous les exemples du livre. Vous pouvez aller au-delà du livre pour découvrir tous les paradigmes de programmation. À l'avant-garde de l'innovation dans le domaine de l'édition électronique, ce labo interactif sera mis à jour continuellement.

OÙ ACHETER LE *LABO INTERACTIF* ?

Le *Labo interactif* peut être acheté en ligne et téléchargé à partir du site web de ScienceActive (www.scienceactive.com). Pour toute information sur les licences multipostes ou de site, contactez ScienceActive (info@scienceactive.com).

Chapitre 1

Introduction aux concepts de programmation

Il n'y a pas de voie royale vers la géométrie.
– La réponse d'Euclide à Ptolémée, Euclide (approx. 300 avant J.-C.)

Il faut suivre la route de briques jaunes.
– *Le Magicien d'Oz*, L. Frank Baum (1856-1919)

La programmation consiste à dire à un ordinateur ce qu'il doit faire. Ce chapitre est une introduction simple à beaucoup des concepts importants de la programmation. Nous supposons que vous avez déjà eu une première expérience de programmation. Nous utiliserons le langage Oz pour introduire les concepts de programmation de manière progressive. Nous vous encourageons à exécuter vous-même les exemples de ce chapitre avec le *Labo interactif* (voir page xiii) ou sur un système Mozart qui tourne chez vous (voir annexe A) [53].

Cette introduction n'aborde qu'une petite partie des concepts que nous verrons. Les chapitres ultérieurs permettent une compréhension profonde des concepts de ce chapitre ainsi que d'autres et des techniques de programmation avec ceux-ci.

1.1 UNE CALCULATRICE

Commençons par utiliser le système pour faire des calculs. Démarrez le *Labo interactif* et regardez le premier exemple. Sélectionnez l'exemple suivant :

```
{Browse 9999*9999}
```

Exécutez l'exemple en cliquant sur le bouton. Le système fait alors le calcul `9999*9999` et affiche le résultat, `99980001`. Les accolades `{ ... }` entourent toujours un appel de fonction ou de procédure. `Browse` est une procédure avec un argument, qui est appelée comme `{Browse X}`.

Vous pouvez aussi utiliser le système Mozart directement en tapant l'instruction suivante :

```
oz
```

ou en double-cliquant sur une icône Mozart. L'interface interactive de Mozart s'ouvre. Ensuite, tapez l'exemple dans la fenêtre d'édition et sélectionnez le texte avec la souris. Pour donner le texte sélectionné au système, activez le menu `Oz` et sélectionnez `Feed Region`.

1.2 LES VARIABLES

En travaillant avec la calculatrice, nous voudrions garder un ancien résultat de calcul afin de pouvoir l'utiliser plus tard sans avoir à le retaper. Nous pouvons le faire avec une variable :

```
declare
V=9999*9999
```

La variable `V` est déclarée et liée à `99980001`. Nous pouvons utiliser cette variable plus tard :

```
{Browse V*V}
```

La réponse `9996000599960001` est affichée. Une variable est simplement un raccourci pour une valeur. Les variables ne peuvent pas être affectées plus d'une fois. Mais vous pouvez déclarer une autre variable avec le même nom qu'une variable précédente. La variable précédente devient alors inaccessible. Les calculs déjà faits avec cette variable ne sont pas changés pour autant. C'est parce qu'il y a deux concepts qui se cachent derrière le mot « variable » :

- L'**identificateur**. C'est la séquence de caractères que vous tapez. Un identificateur commence avec une majuscule et peut être suivi par un nombre quelconque de lettres ou de chiffres. Par exemple, la séquence de caractères `Var1` est un identificateur.
- La **variable en mémoire**. C'est ce que le système utilise pour faire des calculs. Elle fait partie de la mémoire du système.

L'instruction **`declare`** crée une nouvelle variable en mémoire désignée par l'identificateur. Les anciens calculs avec le même identificateur ne sont pas changés parce que l'identificateur fait référence à une autre variable en mémoire.

1.3 LES FONCTIONS

Faisons maintenant un calcul plus compliqué. Supposons que nous voulions calculer la fonction factorielle $n!$, que l'on peut définir comme $1 \times 2 \times \cdots \times (n-1) \times n$. C'est le nombre de permutations de n objets, c'est-à-dire le nombre de manières différentes de les placer dans une séquence. La factorielle de 10 est :

```
{Browse 1*2*3*4*5*6*7*8*9*10}
```

Il s'affiche alors `3628800`. Que faire pour calculer la factorielle de 100 ? Nous voudrions que le système fasse le travail encombrant de taper tous les entiers de 1 à 100. Nous faisons plus : nous expliquons au système comment calculer la factorielle pour tout n. Nous le faisons en définissant une fonction :

```
declare
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

L'instruction **`declare`** crée la nouvelle variable `Fact`. L'instruction **`fun`** définit une fonction. La variable `Fact` est liée à la fonction. La fonction a un argument `N`, qui est une variable locale, c'est-à-dire qu'elle n'est connue que dans le corps de la fonction. Chaque fois que nous appelons la fonction, une nouvelle variable est créée.

La récursion

Le corps de la fonction est une instruction appelée **expression `if`**. Quand la fonction est appelée, l'expression **`if`** fait les choses suivantes :

- Elle teste d'abord si `N` est égal à 0 en faisant le test `N==0`.
- Si le test réussit, l'expression juste après le **`then`** sera calculée. Cela renvoie le nombre 1, parce que la factorielle de 0 est 1.
- S'il échoue, l'expression juste après le **`else`** sera calculée. Si `N` n'est pas 0, l'expression `N*{Fact N-1}` sera calculée. Cette expression utilise `Fact`, la même fonction que nous sommes en train de définir ! Cette technique s'appelle la **récursion**. C'est parfaitement normal et ne devrait pas nous inquiéter.

`Fact` se base sur la définition mathématique suivante de la factorielle :

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \text{ if } n > 0 \end{aligned}$$

Cette définition est récursive parce que la factorielle de `N` est `N` fois la factorielle de `N-1`. Essayons la fonction `Fact` :

```
{Browse {Fact 10}}
```

Ce code affiche `3628800` comme avant, ce qui nous confirme que `Fact` fait le bon calcul. Essayons un argument plus grand :

```
{Browse {Fact 100}}
```

Un nombre énorme est affiché (que nous écrivons ici en groupes de cinq chiffres pour améliorer la lisibilité) :

```
                                                        933 26215
 44394 41526 81699 23885 62667 00490 71596 82643 81621 46859
 29638 95217 59999 32299 15608 94146 39761 56518 28625 36979
 20827 22375 82511 85210 91686 40000 00000 00000 00000 00000
```

C'est un exemple d'un calcul arithmétique avec une précision arbitraire, parfois appelée **précision infinie**, même si la précision n'est pas vraiment infinie. Elle est limitée par la quantité de mémoire dans votre système. Un ordinateur personnel avec 1 Go de mémoire peut traiter des entiers avec des centaines de milliers de chiffres.

Le lecteur sceptique se posera la question : ce nombre énorme est-il vraiment la factorielle de 100 ? Comment le saurait-on ? Faire le calcul à la main sera long et donnera probablement un résultat erroné. Nous verrons plus tard comment nous pouvons avoir confiance dans le système.

Les combinaisons

Écrivons une fonction qui calcule le nombre de combinaisons de k objets pris dans un ensemble de n objets. C'est aussi le nombre de sous-ensembles de taille k pris dans un ensemble de taille n. On l'écrit $\binom{n}{k}$ ou C_n^k en notation mathématique. Nous pouvons définir cette opération avec la factorielle :

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

ce qui nous amène de façon naturelle à la fonction suivante :

```
declare
fun {Comb N K}
   {Fact N} div ({Fact K}*{Fact N-K})
end
```

Par exemple, `{Comb 10 3}` est 120, qui est le nombre de manières de prendre trois objets d'un ensemble de dix. Ce n'est pas la définition la plus efficace de `Comb`, mais c'est probablement la plus simple.

L'abstraction fonctionnelle

La définition de `Comb` utilise la fonction existante `Fact`. Il est toujours possible d'utiliser les fonctions existantes pour définir de nouvelles fonctions. L'utilisation de fonctions pour construire de nouvelles fonctions s'appelle l'**abstraction fonctionnelle**. De cette façon, un programme a une structure d'oignon, avec des couches successives de fonctions qui appellent des fonctions. Ce style de programmation est étudié dans le chapitre 3.

1.4 LES LISTES

Nous pouvons maintenant calculer des fonctions des entiers. Mais un entier n'est pas vraiment grand chose. Supposons que nous voulions effectuer un calcul avec beaucoup d'entiers. Par exemple, si nous voulons calculer le triangle de Pascal[1] :

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
. . . . . . . . . . .
```

La première rangée contient l'entier 1. Chaque élément est la somme des deux éléments directement au-dessus à gauche et à droite. (S'il n'y a pas d'élément, comme sur les bords, alors on prend zéro.) Nous voulons définir une fonction qui calcule la nième rangée en une fois. La nième rangée contient n entiers. Nous pouvons résoudre le problème en utilisant des listes d'entiers.

Une **liste** est une séquence d'éléments, avec une syntaxe délimitée à gauche et à droite par des crochets, comme `[5 6 7 8]`. Pour des raisons historiques, la liste vide est écrite `nil` (et non `[]`). Une liste peut être affichée comme un nombre :

```
{Browse [5 6 7 8]}
```

La notation `[5 6 7 8]` est un raccourci. Une liste est en réalité une chaîne de liens, où chaque lien contient deux choses : un élément de liste et une référence vers le reste de la chaîne. Les listes sont toujours créées élément par élément, en commençant avec `nil` et en ajoutant des liens un à un. Un nouveau lien est écrit `H|T`, où `H` est le nouvel élément et `T` est l'ancienne chaîne. Construisons une liste. Nous commençons avec

1. Le triangle de Pascal est un concept important de l'analyse combinatoire. Les éléments de la nième rangée de ce triangle sont les combinaisons $\binom{n}{k}$, où k va de 0 à n. Il y a un rapport fort avec le binôme de Newton, qui dit $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{(n-k)}$ pour tout entier $n \geqslant 0$.

`Z=nil`. Nous ajoutons un premier lien `Y=7|Z` et ensuite un deuxième lien `X=6|Y`. Maintenant `X` référence une liste avec deux liens, une liste qui peut aussi être écrite comme `[6 7]`.

Le lien `H|T` est souvent appelé un « *cons* » dans la littérature (prononcer à l'anglaise), un terme qui vient du langage Lisp[2]. Si `T` est une liste, alors on pourra utiliser `H` et `T` pour faire la nouvelle liste `H|T` :

```
declare
H=5
T=[6 7 8]
{Browse H|T}
```

La liste `H|T` peut être écrite `[5 6 7 8]`. On dit que sa tête est `5` et sa queue est `[6 7 8]`. La paire `H|T` peut être décomposée pour retrouver la tête et la queue :

```
declare
L=[5 6 7 8]
{Browse L.1}
{Browse L.2}
```

Cela utilise l'opérateur de sélection « `.` » (point), qui permet de sélectionner le premier ou le deuxième élément d'une paire de liste. La sélection `L.1` renvoie la tête de `L`, l'entier `5`. La sélection `L.2` renvoie la queue de `L`, la liste `[6 7 8]`. La figure 1.1 montre ce qui se passe : `L` est une sorte de chaîne dans laquelle chaque lien a un élément et `nil` marque la fin. `L.1` renvoie le premier élément et `L.2` renvoie le reste de la chaîne.

La correspondance de formes

Une manière plus compacte de regarder les composants d'une liste est d'utiliser l'instruction **`case`**, qui permet d'en extraire la tête et la queue en même temps :

```
declare
L=[5 6 7 8]
case L of H|T then {Browse H} {Browse T} end
```

L'affichage montre `5` et `[6 7 8]`, comme avant. L'instruction **`case`** déclare deux variables locales, `H` et `T`, et les lie ensuite avec la tête et la queue de la liste `L`. On dit que l'instruction **`case`** fait de la correspondance de formes (« *pattern matching* » en

2. Une grande partie de la terminologie actuelle des listes a été introduite par le langage Lisp vers la fin des années 1950 [63]. Notre utilisation de la barre verticale vient de Prolog, un langage de programmation logique qui a été inventé au début des années 1970 [15, 88]. En Lisp, le « cons » s'écrit `(H . T)` et on l'appelle aussi une **paire de liste**.

anglais), parce qu'elle décompose `L` selon la « forme » `H|T`. Les variables locales déclarées avec **`case`** se comportent exactement comme les variables déclarées avec **`declare`**, sauf qu'une variable déclarée par **`case`** n'existe que dans le corps de l'instruction **`case`**, entre le **`then`** et le **`end`**.

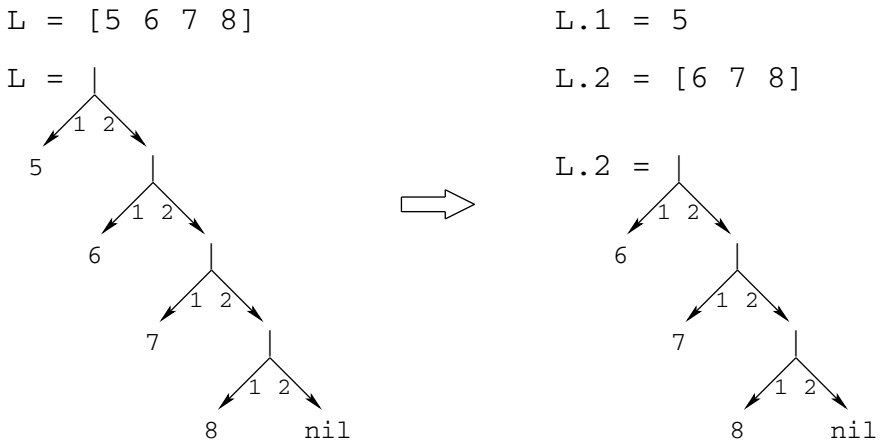


Figure 1.1 Décomposer la liste `[5 6 7 8]`.

1.5 LES FONCTIONS AVEC LES LISTES

Maintenant que nous pouvons calculer avec des listes, définissons une fonction `{Pascal N}` pour calculer la nième rangée du triangle de Pascal. Il faut d'abord comprendre comment faire le calcul à la main. La figure 1.2 montre comment calculer la cinquième rangée à partir de la quatrième. Regardons comment cela fonctionne quand chaque rangée est une liste d'entiers. Pour calculer une rangée, il faut utiliser la rangée précédente. On la déplace d'une position à gauche et d'une position à droite. Ensuite on additionne les rangées déplacées. Par exemple, prenons la quatrième rangée :

```
  [1   3   3   1]
```

Nous déplaçons cette rangée à gauche et à droite pour ensuite additionner élément par élément :

```
  [1   3   3   1   0]
+ [0   1   3   3   1]
```

On remarque qu'un déplacement à gauche ajoute un zéro à l'extrémité droite et un déplacement à droite ajoute un zéro à l'extrémité gauche. Faire l'addition donne :

```
[1   4   6   4   1]
```

ce qui est exactement la cinquième rangée.

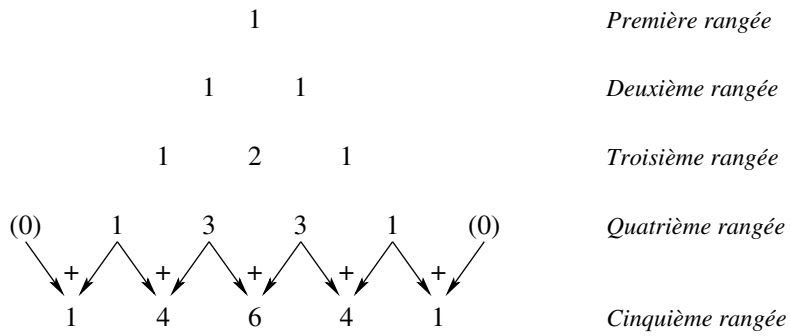


Figure 1.2 Le calcul de la cinquième rangée du triangle de Pascal.

La fonction principale

Maintenant que nous comprenons comment résoudre le problème à la main, nous pouvons écrire une fonction qui fait les mêmes opérations. La voici :

```
declare Pascal AddList ShiftLeft ShiftRight
fun {Pascal N}
   if N==1 then [1] else
     {AddList {ShiftLeft {Pascal N-1}}
              {ShiftRight {Pascal N-1}}} end
end
```

Il y a trois fonctions auxiliaires, `AddList`, `ShiftLeft` et `ShiftRight`, qui restent à définir.

Les fonctions auxiliaires

Pour compléter la solution, il faut définir les trois fonctions auxiliaires : `ShiftLeft`, qui déplace une liste une position à gauche, `ShiftRight`, qui déplace une liste une position à droite, et `AddList`, qui additionne deux listes de même longueur. Voici `ShiftLeft` et `ShiftRight` :

```
fun {ShiftLeft L}
   case L of H|T then H|{ShiftLeft T} else [0] end
end
fun {ShiftRight L} 0|L end
```

`ShiftRight` met simplement un zéro à gauche. `ShiftLeft` est plus compliquée : elle traverse `L` élément par élément et construit le résultat un élément à la fois. Nous avons ajouté un **`else`** à l'instruction **`case`**. Son comportement est analogue à un **`else`** dans un **`if`** : la branche **`else`** sera exécutée si la forme du **`case`** ne correspond pas. Quand `L` est vide (`nil`), le résultat sera `[0]`, une liste avec un élément zéro. Voici `AddList` :

```
fun {AddList L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then H1+H2|{AddList T1 T2} end
   else nil end
end
```

Cette fonction utilise deux **`case`**, l'une imbriquée dans l'autre, parce qu'il faut décomposer deux listes, `L1` et `L2`. Nous avons maintenant la définition complète de `Pascal`. Nous pouvons calculer toutes les rangées du triangle de Pascal. Par exemple, l'appel `{Pascal 20}` renvoie la vingtième rangée :

```
[1 19 171 969 3876 11628 27132 50388 75582 92378
 92378 75582 50388 27132 11628 3876 969 171 19 1]
```

Cette réponse est-elle correcte ? Comment le savoir ? Elle semble correcte : elle est symétrique (inverser la liste donne la même liste) et les premier et deuxième éléments sont respectivement 1 et 19, ce qui est correct. En regardant la figure 1.2, il est facile de comprendre que le deuxième élément de la nième rangée est toujours $n - 1$ (il est toujours un de plus que la rangée précédente et il a la valeur zéro pour la première rangée). Dans la section 1.6, nous verrons comment raisonner sur l'exactitude d'un programme.

Le développement descendant du logiciel

Voici un résumé de la méthodologie que nous avons utilisée pour écrire `Pascal` :

- La première étape est de comprendre comment faire le calcul à la main.
- La deuxième étape est d'écrire une fonction principale pour résoudre le problème, en supposant que les fonctions auxiliaires existent (ici, `ShiftLeft`, `ShiftRight` et `AddList`).
- La troisième étape est de compléter la solution en définissant les fonctions auxiliaires.

La méthodologie qui consiste d'abord à écrire la fonction principale et ensuite à la compléter s'appelle le **développement descendant**. Cette approche est l'une des mieux connues pour le développement du logiciel mais elle n'est pas complète, comme nous le verrons plus tard.

1.6 L'EXACTITUDE

Un programme est correct (« exact ») s'il fait ce que nous voulons qu'il fasse. Comment peut-on s'assurer qu'un programme est correct ? Dans la plupart des cas il est impossible de refaire manuellement les calculs du programme. Il nous faut d'autres techniques. Une technique simple, que nous avons déjà utilisée, est de vérifier que le programme est correct pour les résultats que nous connaissons. Cela augmente notre confiance dans le programme. Mais cela ne va pas très loin. Pour prouver l'exactitude de façon sûre, il faut raisonner sur le programme. Cela signifie trois choses :

- Nous avons besoin d'un modèle mathématique des opérations du langage de programmation qui définit ce qu'elles font. Ce modèle s'appelle la **sémantique du langage**.
- Nous devons définir ce que nous voulons que le programme fasse. C'est une définition mathématique des entrées dont le programme a besoin et des résultats qu'il calcule. Cela s'appelle la **spécification du programme**.
- Nous utilisons des techniques mathématiques pour raisonner sur le programme en utilisant la sémantique du langage. Nous voudrions démontrer que le programme satisfait la spécification.

Un programme que l'on a prouvé correct pourra toujours donner de faux résultats si le système sur lequel il s'exécute est mal implémenté. Comment pouvons-nous être sûr que le système satisfait la sémantique ? La vérification d'un système est une tâche lourde : il faut vérifier le compilateur, le système d'exécution, le système d'exploitation, le matériel et la physique sur laquelle est basée le matériel ! Toutes ces tâches sont importantes, mais elles sont hors de notre portée.[3]

L'induction mathématique

Une technique utile pour raisonner sur un programme est l'induction mathématique. Cette technique se compose de deux étapes. D'abord, nous démontrons que le programme est correct pour le cas le plus simple de l'entrée. Ensuite, nous démontrons que, si le programme est correct pour un cas donné, il sera correct pour le cas suivant. Si nous sommes sûrs que tous les cas seront traités tôt ou tard, alors l'induction mathématique nous permettra de conclure que le programme est toujours correct. On peut appliquer cette technique aux entiers et aux listes :

- Pour les entiers, le cas le plus simple est 0 et pour un entier donné n le cas suivant est $n + 1$.

3. Certains, en s'inspirant des paroles de Thomas Jefferson, diront que le prix de l'exactitude est la vigilance éternelle.

– Pour les listes, le cas le plus simple est `nil` (la liste vide) et pour une liste donnée `T` le cas suivant est `H|T` (sans aucune condition sur `H`).

Voyons comment l'induction fonctionne avec la fonction factorielle :

– `{Fact 0}` renvoie la bonne réponse, à savoir 1.
– Supposons que `{Fact N-1}` est correct. Alors considérons l'appel `{Fact N}`. Nous observons que l'instruction **`if`** choisit la branche **`else`** (parce que `N` n'est pas égale à zéro), et calcule `N*{Fact N-1}`. Selon notre hypothèse, `{Fact N-1}` renvoie la bonne réponse. Donc, en supposant que la multiplication est correcte, `{Fact N}` renvoie aussi la bonne réponse.

Ce raisonnement utilise la définition mathématique de la factorielle, $n! = n \times (n-1)!$ si $n > 0$, et $0! = 1$. Plus loin dans le livre nous verrons d'autres techniques de raisonnement. Mais l'approche de base reste inchangée : commencer avec la sémantique du langage et la spécification du problème, et utiliser le raisonnement mathématique pour démontrer que le programme implémente correctement la spécification.

1.7 LA COMPLEXITÉ CALCULATOIRE

La fonction `Pascal` que nous avons définie ci-dessus devient très lente quand on essaie de calculer des rangées de plus en plus grandes. La rangée 20 prend une fraction de seconde. La rangée 30 prend plusieurs dizaines de secondes.[4] Si vous essayez des rangées encore plus grandes, il faudra attendre patiemment le résultat. Pourquoi la fonction prend-elle autant de temps ? Regardons encore une fois la définition de la fonction `Pascal` :

```
fun {Pascal N}
   if N==1 then [1] else
     {AddList {ShiftLeft {Pascal N-1}}
              {ShiftRight {Pascal N-1}}} end
end
```

Chaque appel de `{Pascal N}` appellera `{Pascal N-1}` deux fois. Donc, l'appel `{Pascal 30}` appellera `{Pascal 29}` deux fois, ce qui fait quatre appels de `{Pascal 28}`, huit de `{Pascal 27}`, et ainsi de suite, doublant avec chaque rangée suivante. Cela donne 2^{29} appels de `{Pascal 1}`, qui est environ un demi-milliard. Il n'est pas étonnant que `{Pascal 30}` soit lent. Pouvons-nous l'accélérer ? Oui, il y a une manière simple : il suffit d'appeler `{Pascal N-1}` une fois au lieu de deux. Comme le deuxième appel donne le même résultat que le premier, si on pouvait

4. Ces chiffres dépendent de la vitesse de votre ordinateur, mais si votre ordinateur peut calculer la rangée 50 dans un temps raisonnable avec cette définition contactez-nous !

le garder, alors un appel suffirait. Nous pouvons le garder en utilisant une variable locale. Voici une nouvelle fonction, `FastPascal`, qui utilise une variable locale :

```
declare
fun {FastPascal N}
   if N==1 then [1] else L in
      L={FastPascal N-1}
      {AddList {ShiftLeft L} {ShiftRight L}} end
end
```

Nous déclarons la variable locale `L` en ajoutant « `L` **`in`** » à la branche **`else`**. C'est comme l'instruction **`declare`**, sauf que l'identificateur peut seulement être utilisé entre le **`else`** et le **`end`**. Nous lions `L` au résultat de `{FastPascal N-1}`. Nous pouvons ensuite utiliser `L` partout où nous en avons besoin. Quelle est la vitesse de `FastPascal` ? Essayez de calculer la rangée 30. Cela prend des dizaines de secondes avec `Pascal`, mais c'est pratiquement instantané avec `FastPascal`. Une leçon à tirer de cet exemple est qu'il est plus important d'avoir un bon algorithme que d'avoir l'ordinateur le plus rapide.

Des garanties sur le temps d'exécution

Comme le montre cet exemple, il est important d'avoir des informations sur le temps d'exécution d'un programme. Connaître le temps exact est moins important que de savoir que le temps n'explosera pas avec la taille de l'entrée. Le temps d'exécution d'un programme en fonction de la taille de l'entrée, à un facteur constant près, s'appelle la **complexité temporelle** du programme. La forme de cette fonction dépend de la façon dont on mesure la taille de l'entrée. Il faut prendre une taille qui a un sens pour l'utilisation du programme. Par exemple, nous prenons l'entier `N` comme la taille de l'entrée de `{Pascal N}` (et pas, comme on pourrait imaginer, la quantité de mémoire nécessaire pour stocker l'entier `N`).

La complexité temporelle de `{Pascal N}` est proportionnelle à 2^n. C'est une fonction exponentielle en n, qui augmente rapidement quand n augmente. Quelle est la complexité temporelle de `{FastPascal N}` ? Il y a n appels récursifs et chaque appel prend un temps proportionnel à n. La complexité temporelle est donc proportionnelle à n^2. C'est une fonction polynomiale en n, qui augmente beaucoup plus lentement qu'une fonction exponentielle. Les programmes dont la complexité temporelle est exponentielle ne sont pas pratiques sauf pour des entrées très petites. En revanche, les programmes sont pratiques si la complexité temporelle est un polynôme de petit ordre.

1.8 LA PROGRAMMATION D'ORDRE SUPÉRIEUR

Nous avons défini une fonction efficace, `FastPascal`, qui calcule les rangées du triangle de Pascal. Maintenant nous voudrions faire des expériences avec des variations du triangle. Par exemple, au lieu d'additionner des nombres pour calculer une rangée, nous voudrions les soustraire, faire un « `Ou` exclusif » (pour voir s'ils sont pairs ou impairs) et ainsi de suite. Une approche est de définir une nouvelle version de `FastPascal` pour chaque variation. Mais cela devient vite exaspérant. Est-il possible d'avoir une version qui permette de réaliser toutes les variations ? C'est possible en effet. On l'appellera `GenericPascal`. Elle a un argument de plus par rapport à `FastPascal` : une fonction qui définit comment calculer les rangées (addition, `Ou` exclusif, etc.). La capacité de passer une fonction dans un argument s'appelle la **programmation d'ordre supérieur**.

Voici la définition de `GenericPascal` avec son argument supplémentaire `Op` :

```
declare GenericPascal OpList
fun {GenericPascal Op N}
   if N==1 then [1] else L in
      L={GenericPascal Op N-1}
      {OpList Op {ShiftLeft L} {ShiftRight L}}
   end
end
```

C'est comme `FastPascal` sauf que nous avons remplacé `AddList` par `OpList`. L'argument `Op` est passé à `OpList`. Comme `ShiftLeft` et `ShiftRight` ne doivent pas connaître `Op`, on peut garder les anciennes versions. Voici la définition de `OpList` :

```
fun {OpList Op L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         {Op H1 H2}|{OpList Op T1 T2} end
   else nil end
end
```

Au lieu de faire l'addition `H1+H2`, cette version fait `{Op H1 H2}`.

Des variations sur le triangle de Pascal

Définissons quelques fonctions pour essayer `GenericPascal`. Pour obtenir le triangle original, il suffit de définir une addition :

```
declare
fun {Add X Y} X+Y end
```

Pour avoir la cinquième rangée on appelle `{GenericPascal Add 5}`.[5] Voici la définition de `FastPascal` avec `GenericPascal` :

```
declare
fun {FastPascal N} {GenericPascal Add N} end
```

Définissons une autre fonction :

```
declare
fun {Xor X Y} if X==Y then 0 else 1 end end
```

Cela fait un `Ou` exclusif, une opération booléenne qui est définie avec le tableau suivant :

X	Y	{Xor X Y}
0	0	0
0	1	1
1	0	1
1	1	0

Un `Ou` exclusif nous permet de calculer la parité de chaque nombre dans le triangle : s'il est pair ou impair. Les nombres eux-mêmes ne sont pas calculés. L'appel `{GenericPascal Xor N}` donne :

```
                1
              1   1
            1   0   1
          1   1   1   1
        1   0   0   0   1
      1   1   0   0   1   1
. . . . . . . . . . . . .
```

Vous trouverez d'autres variations dans les exercices.

1.9 LA CONCURRENCE

Nous voulons que notre programme ait plusieurs activités indépendantes, chacune s'exécutant à sa propre vitesse. Ce concept s'appelle la **concurrence**. Il ne devrait pas y avoir d'interférence entre les activités, sauf si le programmeur décide qu'elles doivent communiquer. Dans le monde réel, cela marche ainsi. Nous voudrions que le système puisse faire la même chose.

5. Nous pouvons aussi appeler `{GenericPascal Number.´+´ 5}`, parce que l'opération d'addition `´+´` fait partie du module `Number`. Mais nous ne parlerons pas des modules dans ce chapitre.

Nous introduisons la concurrence en créant des fils. Un **fil** (« *thread* » en anglais) est simplement un programme en exécution, comme les fonctions que l'on a vues auparavant. La différence est que maintenant un programme peut avoir plusieurs fils. Les fils sont créés avec l'instruction **`thread`**. Vous vous souvenez de la lenteur de la première fonction `Pascal` ? Nous pouvons appeler `Pascal` dans son propre fil. Ainsi, elle n'empêchera pas les autres calculs d'avancer. Ils pourront ralentir si `Pascal` fait vraiment beaucoup de calculs. Cela arrivera si les fils partagent le même processeur. Mais aucun fil ne s'arrêtera. Voici un exemple :

```
thread {Browse {Pascal 30}} end
{Browse 99*99}
```

qui crée un nouveau fil. Dans ce fil, nous appelons `{Pascal 30}` et puis `Browse` pour afficher le résultat. Le nouveau fil fait beaucoup de calculs. Mais cela n'empêche pas le système d'afficher `99*99` sans délai.

1.10 LE DATAFLOW

Que se passe-t-il si une opération essaie d'utiliser une variable qui n'est pas encore liée ? D'un point de vue esthétique, il serait agréable que l'opération attende tout simplement. Si un autre fil lie la variable, l'opération pourra continuer. Ce comportement civilisé s'appelle le **dataflow** (« flux de données »). La figure 1.3 montre un exemple simple : les deux multiplications attendent jusqu'à ce que leurs arguments soient liés et l'addition attend jusqu'à ce que les multiplications soient complétées. Comme nous le verrons plus loin, il y a de nombreuses raisons d'utiliser le comportement dataflow. Pour l'instant, voyons comment le dataflow et la concurrence fonctionnent ensemble. Prenez par exemple :

```
declare X in
thread {Delay 10000} X=99 end
{Browse start} {Browse X*X}
```

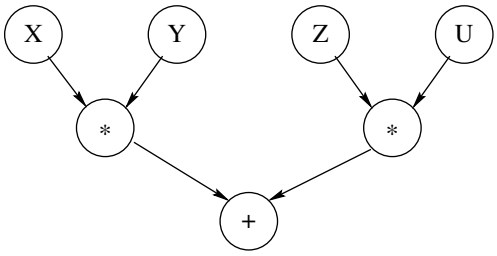


Figure 1.3 Un exemple simple de l'exécution dataflow.

Le premier `Browse` affiche tout de suite `start`. La multiplication `X*X` attend jusqu'à ce que `X` soit liée. Le deuxième `Browse` attend la multiplication, alors il n'affichera encore rien. L'appel `{Delay 10000}` fait une pause d'au moins 10000 ms (10 secondes). `X` ne sera liée qu'après ce délai. Quand `X` est liée, alors la multiplication continue et le deuxième `Browse` affiche 9801. Les deux opérations `X=99` et `X*X` peuvent être faites dans n'importe quel ordre avec n'importe quel délai ; l'exécution dataflow donnera toujours le même résultat. Le seul effet possible d'un délai est un ralentissement, pas un changement de la valeur du résultat. Par exemple, faisons un changement dans l'ordre des opérations :

```
declare X in
thread {Browse start} {Browse X*X} end
{Delay 10000} X=99
```

Le comportement est exactement comme avant : le Browse affiche 9801 après 10 secondes. Cet exemple montre deux propriétés agréables de l'exécution dataflow. Premièrement, les calculs sont corrects indépendamment de leur partitionnement entre fils. Deuxièmement, les calculs sont patients : ils ne signalent pas d'erreurs, mais attendent.

L'ajout des fils et des délais à un programme peut radicalement changer son apparence. Mais aussi longtemps que les mêmes opérations sont appelées avec les mêmes arguments, les résultats du programme ne changeront pas du tout. C'est la propriété majeure de la concurrence dataflow. C'est pourquoi elle possède la plupart des avantages de la programmation concurrente sans les difficultés qui y sont typiquement associées. La concurrence dataflow est traitée dans le chapitre 4.

1.11 L'ÉTAT EXPLICITE

Comment pouvons-nous définir une fonction qui peut apprendre de son passé ? C'est-à-dire une fonction ayant une sorte de mémoire interne qui l'aide à faire son travail. Ce genre de mémoire est appelée l'état explicite. Tout comme la concurrence, l'état explicite modélise un aspect essentiel du fonctionnement du monde réel. Nous voudrions pouvoir faire cela à l'intérieur du système. Nous verrons ultérieurement des raisons plus profondes d'utiliser l'état explicite (voir chapitre 5). Pour le moment, nous nous contenterons de voir comment il fonctionne, par exemple si nous voulons savoir combien de fois la fonction `FastPascal` est appelée. Pouvons-nous modifier `FastPascal` pour qu'elle se souvienne combien de fois elle a été appelée ? On peut le faire avec l'état explicite.

Une cellule de mémoire

Il y a beaucoup de manières de définir un état explicite. La manière la plus simple est de définir une cellule de mémoire. C'est une sorte de boîte dans laquelle vous pouvez

insérer un contenu arbitraire. Beaucoup de langages de programmation l'appellent une « variable ». Nous l'appelons une « cellule » pour éviter la confusion avec les variables que nous avons utilisées auparavant, qui sont des vraies variables mathématiques, c'est-à-dire des raccourcis pour des valeurs. Il y a trois opérations sur les cellules : `NewCell` crée une nouvelle cellule, `:=` (affectation) met une nouvelle valeur dans la cellule et `@` (accès) donne accès à la valeur actuellement dans la cellule. L'accès et l'affectation sont aussi appelés lecture et écriture. Par exemple :

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

Ce code crée d'abord une cellule `C` avec un contenu initial 0, ajoute ensuite 1 à ce contenu, et enfin l'affiche.

Ajouter une mémoire interne à `FastPascal`

Avec une cellule mémoire, `FastPascal` peut compter le nombre de fois qu'elle est appelée. D'abord nous créons une cellule à l'extérieur de `FastPascal`. Ensuite, à l'intérieur de `FastPascal`, nous ajoutons 1 au contenu de la cellule lors d'un appel. Voici le code :

```
declare
C={NewCell 0}
fun {FastPascal2 N}
   C:=@C+1
   {FastPascal Add N}
end
```

Cette définition fait une nouvelle fonction `FastPascal2` qui appelle `FastPascal` et qui gère la cellule.

1.12 LES OBJETS

Une fonction avec une mémoire interne est généralement appelée un **objet**. La nouvelle fonction `FastPascal2` définie ci-dessus est un objet. Il semble qu'il s'agisse de petites bêtes très utiles. Un autre exemple, à savoir un objet compteur, contient une cellule qui garde le compte actuel. Il a deux opérations, `Bump` et `Read`, que nous appelons son interface. `Bump` incrémente le compteur par 1 et renvoie le résultat. `Read` renvoie simplement le compte actuel. Voici la définition :

```
declare
local C in
   C={NewCell 0}
   fun {Bump} C:=@C+1 @C end
   fun {Read} @C end
end
```

L'instruction **local** déclare une nouvelle variable C qui est visible seulement jusqu'au **end** correspondant. Il y a quelque chose d'extraordinaire dans cet exemple : comme la cellule est référencée par une variable locale, elle est complètement invisible de l'extérieur. Cette technique s'appelle l'**encapsulation**. Si on utilise l'encapsulation, les utilisateurs du compteur ne pourront pas modifier l'intérieur du compteur. Nous pouvons donc garantir qu'il fonctionnera toujours correctement. Ce n'était pas vrai pour FastPascal2 parce que sa cellule est globale : n'importe qui peut la modifier.

Il s'ensuit qu'aussi longtemps que l'interface de l'objet compteur ne change pas, son utilisateur ne peut pas savoir si l'implémentation a changé. La séparation de l'interface et de l'implémentation est l'essence même de l'abstraction de données. Elle peut largement simplifier le programme de l'utilisateur. Un programme qui utilise un compteur fonctionnera pour toute implémentation aussi longtemps que l'interface restera la même. Cette propriété s'appelle le **polymorphisme**. L'abstraction de données avec l'encapsulation et le polymorphisme est traitée dans le chapitre 5.

Nous pouvons incrémenter le compteur :

```
{Browse {Bump}}
{Browse {Bump}}
```

Qu'est-ce qui est affiché ? Bump peut être appelé partout dans un programme pour compter le nombre de fois que quelque chose se produit. Par exemple, FastPascal2 peut utiliser Bump :

```
declare
fun {FastPascal2 N}
   {Browse {Bump}}
   {FastPascal Add N}
end
```

1.13 LES CLASSES

Dans la dernière section nous avons défini un objet compteur. Que faire si on a besoin de plusieurs compteurs ? Il serait bien d'avoir une « usine » pour fabriquer autant de compteurs que nécessaire. Une telle usine s'appelle une **classe**. Voici une manière de la définir :

```
declare
fun {NewCounter}
C Bump Read in
   C={NewCell 0}
   fun {Bump} C:=@C+1 @C end
   fun {Read} @C end
   counter(bump:Bump read:Read)
end
```

`NewCounter` est une fonction qui crée une nouvelle cellule et renvoie de nouvelles fonctions `Bump` et `Read` qui utilisent cette cellule. Nous remarquons qu'une fonction peut renvoyer une fonction comme résultat. La possibilité de créer des fonctions à tout moment et de les traiter comme n'importe quelle autre valeur s'appelle la programmation d'ordre supérieur. La section 1.8 en donne un autre exemple. C'est une technique de programmation qui est souvent utilisée.

Nous groupons les fonctions `Bump` et `Read` ensemble dans un enregistrement, qui est une structure de données composée permettant un accès facile à ses parties. L'enregistrement `counter(bump:Bump read:Read)` est caractérisé par son étiquette `counter` et par ses deux champs, appelés `bump` et `read`. Nous pouvons créer deux compteurs :

```
declare
Ctr1={NewCounter}
Ctr2={NewCounter}
```

Chaque compteur a sa propre mémoire interne et ses propres fonctions `Bump` et `Read`. Nous pouvons accéder à ses fonctions avec l'opérateur de sélection « `.` » (point). `Ctr1.bump` donne la fonction `Bump` du premier compteur. Nous pouvons incrémenter le premier compteur et afficher sa valeur :

```
{Browse {Ctr1.bump}}
```

Vers la programmation orientée objet

Nous avons donné un exemple d'une classe `NewCounter` qui définit les deux opérations `Bump` et `Read`. Les opérations définies dans une classe sont appelées ses méthodes. La classe peut être utilisée pour faire autant d'objets compteurs qu'il nous faut. Tous ces objets partagent les mêmes méthodes, mais chacun a sa propre mémoire interne. La programmation avec les classes et les objets s'appelle la **programmation basée objet**.

Si nous ajoutons une dernière idée, l'héritage, à la programmation basée objet, nous obtiendrons la **programmation orientée objet**. L'héritage nous permet de définir une nouvelle classe par rapport à des classes existantes, simplement en spécifiant en

quoi la nouvelle classe est différente. Par exemple, supposons que nous ayons une classe compteur avec une seule méthode, `Bump`. Nous pouvons définir une nouvelle classe qui prend la première classe et ajoute une deuxième méthode `Read`. On dit que la nouvelle classe hérite de l'ancienne. L'héritage est un concept puissant pour la structuration des programmes. Le chapitre 6 étudie la programmation orientée objet et montre comment programmer avec l'héritage.

1.14 LE NON-DÉTERMINISME ET LE TEMPS

Nous avons vu comment ajouter la concurrence et l'état explicite à un programme, mais de façon séparée. Que se passe-t-il quand un programme utilise les deux ensemble ? Nous constatons que les utiliser ensemble est difficile, car le même programme peut donner des résultats différents d'une exécution à une autre puisque l'ordre dans lequel les fils utilisent l'état peut changer d'une exécution à une autre. Cette variabilité s'appelle le **non-déterminisme**. Le non-déterminisme existe car nous ne connaissons pas le temps exact de l'exécution de chaque opération. Si nous pouvions connaître ce temps, il n'y aurait pas de non-déterminisme. Mais nous ne pouvons le connaître, simplement parce que les fils sont indépendants. Chaque fil ne peut donc pas connaître quelles sont les instructions exécutées par les autres fils à un moment donné.

Le non-déterminisme en soi n'est pas un problème ; nous l'avons déjà avec la concurrence. Les difficultés apparaîtront seulement si le non-déterminisme devient visible dans le programme, c'est-à-dire s'il devient observable. Un non-déterminisme observable est parfois appelé une course (« *race condition* »). En voici un exemple simple :

```
declare
C={NewCell 0}
thread C:=1 end
thread C:=2 end
```

Quel est le contenu de `C` après l'exécution de ce programme ? La figure 1.4 montre deux exécutions possibles de ce programme. Selon l'exécution qui est faite, le contenu final de la cellule peut être 1 ou 2. Le problème est que nous ne pouvons pas prévoir ce qui arrivera. C'est un exemple de non-déterminisme observable. Le comportement peut devenir encore plus compliqué. Par exemple, utilisons une cellule pour stocker un compteur qui peut être incrémenté par plusieurs fils :

```
declare
C={NewCell 0}
thread I in I=@C C:=I+1 end
thread J in J=@C C:=J+1 end
```

Quel est le contenu de `C` après l'exécution de ce programme ? Il semble que chaque fil ajoute 1 au contenu, ce qui fait 2 à la fin. Mais il y a une surprise : le contenu final

peut être 1 ! Comment est-ce possible ? Essayez de comprendre pourquoi avant de continuer.[6]

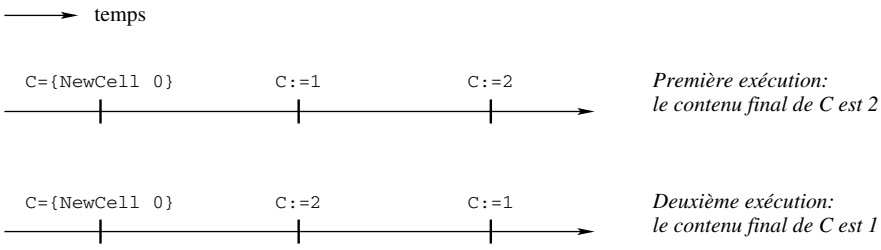


Figure 1.4 Toutes les exécutions possibles du premier exemple non-déterministe.

L'entrelacement

Le contenu peut être 1 parce que l'exécution des fils est entrelacée. Chaque fil attend son tour pour s'exécuter un peu. Il faut supposer que tous les entrelacements possibles peuvent arriver. Par exemple, prenez l'exécution montrée dans la figure 1.5. Les variables `I` et `J` sont toutes les deux liées à 0. Ensuite, parce que `I+1` et `J+1` font tous les deux 1, la cellule sera affectée à 1 deux fois. Le résultat final est que le contenu de la cellule est 1.

Cet exemple est simple. Dans les programmes plus compliqués, le nombre d'entrelacements est beaucoup plus grand. Il faut écrire le programme pour qu'il soit correct pour chaque entrelacement. Ce n'est pas une tâche facile. Dans l'histoire des ordinateurs, un grand nombre de « bugs » célèbres et dangereux viennent du fait que les concepteurs n'avaient pas réalisé la vraie difficulté. L'appareil de radiothérapie Therac-25 en est un exemple connu. À cause des erreurs de programmation concurrente, l'appareil émettait parfois des doses de radiation des milliers de fois plus grandes que la normale, ce qui provoquait chez le patient une blessure grave ou même mortelle [60].

Il y a donc une première leçon à retenir pour les programmes qui utilisent l'état et la concurrence : si possible, ne les utilisez pas ensemble ! Il apparaît que souvent il n'est pas nécessaire d'utiliser les deux ensemble. Quand un programme a besoin des deux, il peut presque toujours être conçu pour limiter leur interaction à une toute petite partie du programme.

6. Pour faciliter votre raisonnement, l'exemple est codé pour rendre les variables intermédiaires visibles. Nous aurions pu écrire **`thread`** `C:=@C+1` **`end`**, qui a le même comportement mais sans montrer de variable intermédiaire !

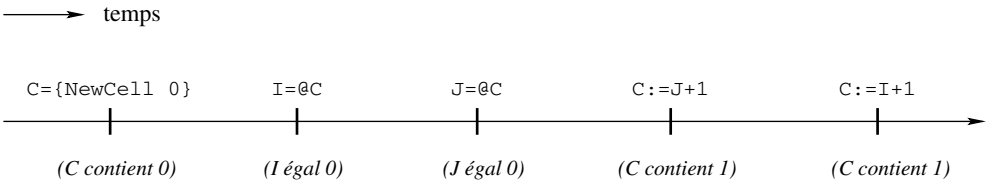


Figure 1.5 Une exécution possible du deuxième exemple non-déterministe.

1.15 EXERCICES

➤ **Exercice 1** — *Une calculatrice*

La section 1.1 montre comment utiliser le *Labo interactif* et le système Mozart comme une calculatrice. Explorons les possibilités :

a) Calculez la valeur exacte de 2^{100} sans utiliser de fonctions. Essayez d'utiliser des raccourcis pour éviter de taper cent fois le chiffre 2 dans la formule `2*2*2*...*2`.

Indice : Utilisez des variables pour garder les résultats intermédiaires.

b) Calculez la valeur exacte de 100! sans utiliser de fonctions. Des raccourcis sont-ils possibles ?

➤ **Exercice 2** — *Le calcul des combinaisons*

La section 1.3 définit la fonction `Comb` qui calcule les combinaisons. Cette fonction n'est pas très efficace parce qu'elle demande parfois le calcul de grandes factorielles.[7] Le but de cet exercice est d'écrire une version plus efficace de `Comb`.

a) Dans un premier temps, utilisez la définition suivante pour faire une fonction plus efficace :

$$\binom{n}{k} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 1}$$

Calculez le numérateur et le dénominateur séparément et faites ensuite la division. Vérifiez que le résultat est 1 quand $k = 0$.

b) Dans un deuxième temps, utilisez l'identité suivante :

$$\binom{n}{k} = \binom{n}{n-k}$$

7. Par contre, elle a le grand avantage de toujours donner la bonne réponse ! C'est parce que dans notre modèle les entiers sont de vrais entiers à cause de l'arithmétique qui se fait avec une précision arbitraire. On ne peut pas en dire autant pour la même fonction en Java, parce que le type `int` en Java n'est pas un vrai entier (c'est un entier modulo 2^{32}, ce qui est bien plus compliqué). Il y a une leçon à retenir de cet exemple : les abstractions (comme les entiers) doivent être simples.

pour encore augmenter l'efficacité. Si $k > n/2$ on pourra faire le calcul avec $n - k$ au lieu de k.

➤ **Exercice 3** — *L'exactitude d'un programme*

La section 1.6 explique les idées de base pour vérifier l'exactitude des programmes et les utilise pour démontrer que la fonction factorielle définie dans la section 1.3 est correcte. Dans cet exercice, appliquez les mêmes idées pour démontrer l'exactitude de la fonction `Pascal` de la section 1.5.

➤ **Exercice 4** — *La complexité calculatoire d'un programme*

Que dit la section 1.7 par rapport aux programmes dont la complexité temporelle est un polynôme de grand ordre ? Sont-ils pratiques ou pas ? Qu'en pensez-vous ?

➤ **Exercice 5** — *La programmation d'ordre supérieur*

La section 1.8 explique comment utiliser la programmation d'ordre supérieur pour calculer des variations du triangle de Pascal. Le but de cet exercice est d'explorer ces variations.

a) Calculez quelques rangées avec la soustraction, la multiplication et d'autres opérations. Pourquoi la multiplication donne-t-elle un triangle rempli de zéros ? Essayez la multiplication suivante :

```
fun {Mul1 X Y} (X+1)*(Y+1) end
```

À quoi ressemble la dixième rangée avec la fonction `Mul1` ?

b) L'instruction de boucle **`for`** peut calculer et afficher dix rangées à la fois :

```
for I in 1..10 do
   {Browse {GenericPascal Op I}}
end
```

Utilisez cette instruction pour faciliter l'exploration des variations.

➤ **Exercice 6** — *L'état explicite*

Cet exercice compare les variables et les cellules. Nous montrons deux fragments de programme. Le premier utilise des variables :

```
local X in
   X=23
   local X in X=44 end
   {Browse X}
end
```

Le deuxième utilise une cellule :

```
local X in
   X={NewCell 23}
   X:=44
   {Browse @X}
end
```

Dans le premier fragment, l'identificateur `X` fait référence à deux variables différentes. Dans le deuxième, `X` fait référence à une cellule. Qu'est-ce qui est affiché par le `Browse` dans chaque fragment ? Expliquez votre réponse.

➤ **Exercice 7** — *L'état explicite et les fonctions*
Cet exercice explore l'utilisation des cellules dans les fonctions. Nous définissons une fonction `{Accumulate N}` qui accumule toutes ses entrées, c'est-à-dire qu'elle additionne les arguments de tous ses appels. Voici un exemple :

```
{Browse {Accumulate 5}}
{Browse {Accumulate 100}}
{Browse {Accumulate 45}}
```

Ces appels afficheront 5, 105 et 150, en supposant que l'accumulateur contient zéro au début. Voici une manière erronée d'écrire `Accumulate` :

```
declare
fun {Accumulate N}
Acc in
   Acc={NewCell 0}
   Acc:=@Acc+N
   @Acc
end
```

Quelle est l'erreur dans cette définition ? Comment la corrigeriez-vous ?

➤ **Exercice 8** — *L'état explicite et la concurrence*
La section 1.14 donne un exemple d'une cellule qui est incrémentée par deux fils.

a) Exécutez cet exemple plusieurs fois. Quels sont vos résultats ? Obtenez-vous parfois le résultat 1 ? Pourquoi oui ou pourquoi non ?

b) Modifiez l'exemple en ajoutant des appels à `Delay` dans chaque fil. Cela change l'entrelacement des fils sans changer les calculs faits par les fils. Pouvez-vous trouver un jeu de délais qui donne toujours le résultat 1 ?

Chapitre 2

La programmation déclarative

Non sunt multiplicanda entia praeter necessitatem.
Ne pas multiplier les entités au-delà de la nécessité.
– Le rasoir d'Ockham, d'après Guillaume d'Ockham (1285 ?-1347/49)

La programmation se compose de trois éléments :

- Premièrement, d'un modèle de calcul : un système formel qui définit un langage et comment les phrases de ce langage (expressions et instructions) sont exécutées par une machine abstraite. Il y a beaucoup de modèles différents de calcul. Ici nous nous attachons aux modèles qui sont particulièrement intuitifs et utiles pour les programmeurs.
- Deuxièmement, d'un ensemble de techniques de programmation et de principes de conception pour écrire des programmes dans le langage du modèle de calcul. Nous appelons cela un modèle de programmation.
- Troisièmement, d'un ensemble de techniques pour raisonner sur les programmes, pour augmenter leur fiabilité et pour calculer leur efficacité.

Cette définition du modèle de calcul est très générale. Tous les modèles ainsi définis ne sont pas raisonnables pour les programmeurs. Quand un modèle est-il raisonnable ? Intuitivement, nous disons qu'un modèle est raisonnable si on peut l'utiliser pour résoudre un grand nombre de problèmes pratiques, s'il a des techniques pratiques et faciles de raisonnement et s'il peut être implémenté de façon efficace. Le premier modèle que nous étudierons est aussi le plus simple : c'est la **programmation déclarative**. Pour le moment, nous la définissons comme l'évaluation des fonctions sur des

structures de données partielles. On l'appelle aussi la **programmation sans état**, par opposition à la programmation avec état qui est traitée dans le chapitre 5.

Le modèle déclaratif de ce chapitre est un modèle clé. Il contient le noyau de deux paradigmes de programmation importants, la programmation fonctionnelle et la programmation logique. Il permet la programmation des fonctions sur les valeurs complètes, comme dans les langages Scheme et Standard ML. Il permet aussi la programmation logique déterministe, comme dans Prolog quand la recherche n'est pas utilisée. Et finalement, on peut l'étendre avec la concurrence tout en gardant ses bonnes propriétés déclaratives (voir le chapitre 4).

La programmation déclarative est un domaine riche : elle contient la plupart des idées des modèles plus expressifs, si ce n'est qu'en forme embryonnaire. Nous la présentons en deux chapitres. Le chapitre 2 définit le modèle de calcul et un langage déclaratif basé sur celui-ci. Il ajoute aussi le concept d'exception (voir section 2.7), qui nous permet de gérer les erreurs dans un programme. Le chapitre 3 présente des techniques de programmation pour le langage déclaratif. Il ajoute aussi le concept de nom, une constante infalsifiable (voir section 3.5), ce qui permet de construire des types de données abstraits. Les chapitres suivants enrichissent le modèle de base avec encore d'autres concepts. Le chapitre 4 ajoute celui de fil (« *thread* »), qui permet de faire de la programmation concurrente dataflow (avec des flots). Le chapitre 5 ajoute le concept de cellule (« *cell* », ou variable affectable), qui permet de faire de la programmation avec état et abstractions de données. Le chapitre 6 explique la programmation orientée objet, un ensemble de techniques pour bien utiliser la programmation avec état.

D'autres concepts ne peuvent pas être décrits ici, faute de place. Par exemple, les capacités (pour l'encapsulation et la sécurité), l'exécution paresseuse, les canaux de communication et la programmation par envoi de messages, les objets actifs et la programmation par état partagé avec les verrous, les moniteurs et les transactions. Tous ces concepts se trouvent dans la version originale du livre [97], qui contient aussi des chapitres sur les interfaces homme-machine, la programmation répartie, la programmation relationnelle (comme le langage Prolog) et la programmation par contraintes.

Parmi ces concepts, les capacités sont particulièrement importantes. Une **capacité objet** est une référence infalsifiable à un objet [67]. Un système basé sur les capacités objet donne une sécurité en profondeur qui permet d'éviter la plupart des problèmes des virus et autres programmes malicieux, sans sacrifier la facilité de son utilisation. Pour des raisons historiques, la plupart des systèmes d'exploitation actuels ne soutiennent pas assez les capacités objet pour jouir de ces propriétés.

La structure du chapitre

Le chapitre contient sept parties :

- La section 2.1 explique comment définir la syntaxe et la sémantique des langages de programmation pratiques. La syntaxe est définie par une grammaire hors-contexte qui est étendue avec des contraintes propres au langage. La sémantique est définie en deux parties : d'abord en traduisant les programmes du langage pratique en un langage noyau simple et ensuite en donnant la sémantique du langage noyau. Cette approche sera utilisée partout. Ce chapitre l'utilise pour définir le modèle de calcul déclaratif.
- Les trois sections suivantes définissent la syntaxe et la sémantique du modèle déclaratif :
  - La section 2.2 définit les structures de données : la mémoire à affectation unique et son contenu, les valeurs partielles et les variables dataflow.
  - La section 2.3 définit la syntaxe du langage noyau.
  - La section 2.4 définit la sémantique du langage noyau avec une machine abstraite simple. Elle est conçue pour être intuitive et permettre un raisonnement simple sur l'exactitude et la complexité calculatoire.
- La section 2.5 utilise la machine abstraite pour comprendre comment les calculs se comportent dans la mémoire. Nous étudierons l'optimisation terminale et le cycle de vie mémoire.
- La section 2.6 définit un langage pratique au-dessus du langage noyau.
- La section 2.7 étend le modèle déclaratif avec le traitement d'exceptions, ce qui permet aux programmes de traiter des situations imprévues et des erreurs exceptionnelles.

2.1 DÉFINIR UN LANGAGE DE PROGRAMMATION PRATIQUE

Les langages de programmation sont bien plus simples que les langages naturels, mais ils peuvent tout de même avoir une syntaxe, des abstractions et des librairies étonnamment riches. C'est particulièrement vrai pour les langages de programmation conçus pour solutionner des problèmes dans le monde réel. Nous les appelons langages pratiques. Un langage pratique est comme la boîte à outils d'un mécanicien expérimenté : il y a beaucoup d'outils différents pour des utilisations différentes et tous les outils ont une raison d'être.

Cette section traite les éléments de base. Elle explique comment nous présentons la syntaxe (la « grammaire ») et la sémantique (le « sens ») des langages de programmation pratiques. Avec ces bases nous serons prêts pour définir le premier modèle de

calcul, le modèle déclaratif. Nous continuerons à utiliser ces bases pour définir les autres modèles.

2.1.1 La syntaxe d'un langage

La syntaxe d'un langage définit quels sont les programmes légaux, c'est-à-dire les programmes qui peuvent être exécutés. Pour l'instant nous ne nous intéressons pas à ce que font ces programmes. Le comportement des programmes lors de leur exécution est donné par la sémantique. Celle-ci sera traitée dans la section 2.1.2.

Les grammaires

Une grammaire est un ensemble de règles qui définit comment construire des « phrases » à partir des « mots ». Les grammaires peuvent être utilisées pour les langages naturels, comme le français ou le suédois, tout comme les langages artificiels, comme les langages de programmation. Pour les langages de programmation, les « phrases » sont appelées des « instructions » et les « mots » sont appelés des « jetons ». Tout comme les mots sont construits de lettres, les jetons sont faits de caractères. Il y a deux niveaux de structure :

instruction (« phrase »)	=	séquence de jetons (« mots »)
jeton (« mot »)	=	séquence de caractères (« lettres »)

Les grammaires sont utiles pour définir les instructions et les jetons. La figure 2.1 contient un exemple qui montre comment une séquence de caractères est transformée en instruction. L'exemple dans la figure est la définition de `Fact` :

```
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

L'entrée est une séquence de caractères où ´ ´ représente un espace et ´\n´ représente un saut de ligne. Cette séquence est d'abord transformée en une séquence de jetons et ensuite en un arbre syntaxique. (La syntaxe des deux séquences dans la figure est compatible avec la syntaxe des listes du langage Oz.)

Alors que les séquences sont « plates », l'arbre syntaxique montre la structure de l'instruction. Un programme qui prend une séquence de caractères et renvoie une séquence de jetons s'appelle un **analyseur de jetons** ou un **analyseur lexical**. Un programme qui prend une séquence de jetons et renvoie un arbre syntaxique s'appelle un **parseur**.

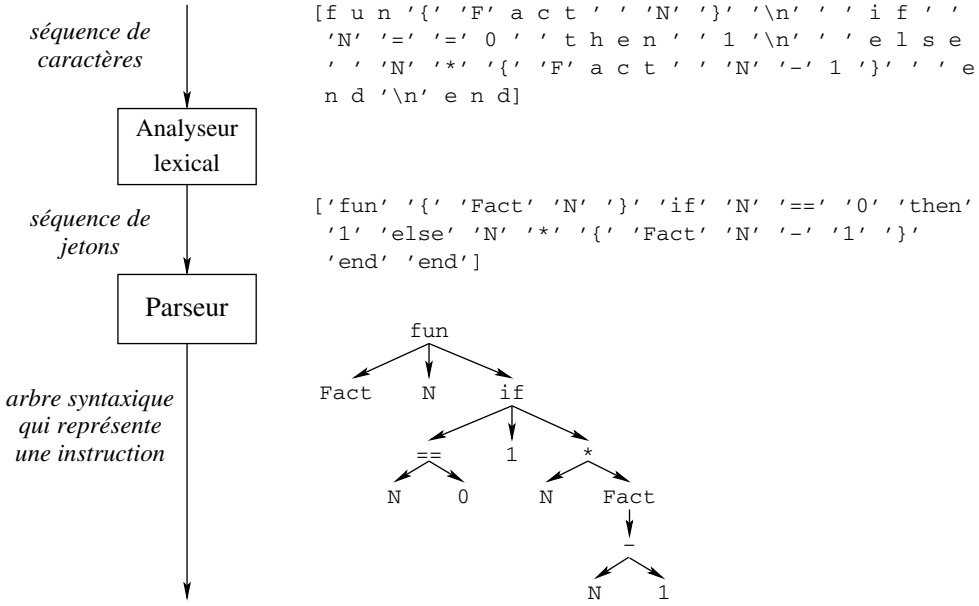


Figure 2.1 Des caractères aux instructions.

La Forme Étendue de Backus-Naur (EBNF)

Un des formalismes les plus populaires pour définir les grammaires s'appelle la Forme Étendue de Backus-Naur (« *Extended Backus-Naur Formalism* », EBNF), d'après ses inventeurs John Backus et Peter Naur. Le formalisme EBNF distingue des symboles terminaux et des symboles non terminaux. Un symbole terminal est simplement un jeton. Un symbole non terminal représente une séquence de jetons. Le non terminal est défini avec une règle de grammaire, qui montre comment le convertir en une séquence de jetons. Par exemple, la règle suivante définit le non terminal ⟨digit⟩ :

⟨digit⟩ : := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Elle dit que ⟨digit⟩ représente un des dix jetons 0, 1, . . . , 9. Le symbole « | » est lu « ou » ; il désigne un choix entre des alternatives. Les règles de grammaire peuvent référer à d'autres non terminaux. Par exemple, nous pouvons définir le non terminal ⟨int⟩ qui définit comment écrire les entiers positifs :

⟨int⟩ : := ⟨digit⟩ { ⟨digit⟩ }

Cette règle dit qu'un entier est écrit comme un chiffre suivi d'un nombre quelconque de chiffres, y compris zéro. Les accolades « { ⟨digit⟩ } » autour de ⟨digit⟩ définissent la répétition de ⟨digit⟩ zéro ou plusieurs fois.

Comment lire les grammaires

Il faut commencer par un symbole non terminal comme ⟨int⟩. Lire la règle correspondante de gauche à droite donne une séquence de jetons selon le schéma suivant :

- Chaque symbole terminal rencontré est ajouté à la séquence.
- Pour chaque symbole non terminal rencontré, lisez sa règle de grammaire et remplacez le non terminal par la séquence de jetons qu'il définit.
- Chaque fois qu'il y a un choix (le symbole « | »), choisissez une des alternatives.

La grammaire peut être utilisée pour vérifier qu'une instruction est légale et pour générer des instructions.

Les grammaires hors-contexte et sensibles au contexte

Tout ensemble de mots s'appelle un **langage formel**. Par exemple, l'ensemble de toutes les instructions générées par une grammaire et un symbole non terminal font un langage formel. Les techniques pour définir des grammaires peuvent être classifiées selon leur expressivité, c'est-à-dire selon les caractéristiques des langages qu'elles peuvent générer. Par exemple, le formalisme EBNF défini ci-dessus définit une classe de grammaires que l'on appelle **grammaires hors-contexte**. Elles s'appellent ainsi parce que l'expansion d'un non terminal, comme ⟨digit⟩, donne toujours la même séquence indépendamment de la position du non terminal dans une règle.

Pour la plupart des langages de programmation pratiques, il n'y a pas de grammaire hors-contexte qui génère exactement l'ensemble des programmes légaux. Par exemple, dans beaucoup de langages une variable doit être déclarée avant son utilisation. Cette condition ne peut pas être exprimée dans une grammaire hors-contexte parce que l'expansion du non terminal qui génère la variable dépend des variables déjà déclarées. C'est une dépendance du contexte. Une grammaire qui contient un non terminal dont l'utilisation dépend du contexte dans lequel il est utilisé s'appelle une **grammaire sensible au contexte**.

La syntaxe de la plupart des langages de programmation pratiques est donc définie en deux parties (voir figure 2.2) : une grammaire hors-contexte et un ensemble de conditions supplémentaires imposées par le langage. Nous gardons la grammaire hors-contexte au lieu d'un formalisme plus expressif parce qu'elle est simple à lire et à comprendre. Elle a une importante propriété de localité : un symbole non terminal peut être compris en examinant seulement les règles qui le définissent ; les règles (parfois beaucoup plus nombreuses) qui l'utilisent peuvent être ignorées. La grammaire hors-contexte est corrigée en imposant des conditions supplémentaires, comme la restriction « déclaration avant utilisation » des variables. Avec ces conditions, la grammaire devient sensible au contexte.

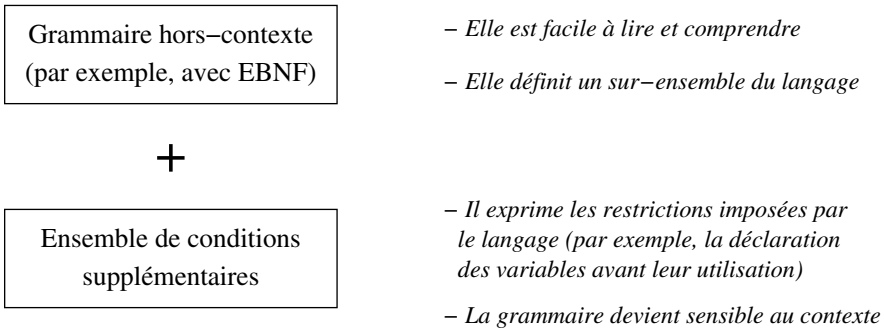


Figure 2.2 L'approche hors-contexte pour définir la syntaxe d'un langage.

L'ambiguïté

Les grammaires hors-contexte peuvent être **ambiguës**, c'est-à-dire qu'il y a plusieurs arbres syntaxiques qui correspondent à la même séquence de jetons. En voici un exemple. Une grammaire simple pour les expressions arithmétiques qui contiennent l'addition et la multiplication :

⟨exp⟩ : := ⟨int⟩ | ⟨exp⟩ ⟨op⟩ ⟨exp⟩
⟨op⟩ : := + | ∗

L'expression `2*3+4` correspond à deux arbres syntaxiques selon la manière de lire les deux occurrences de ⟨exp⟩. La figure 2.3 montre les deux arbres. Dans un arbre, la première ⟨exp⟩ est `2` et la deuxième ⟨exp⟩ est `3+4`. Dans l'autre arbre, elles sont respectivement `2*3` et `4`.

En principe l'ambiguïté est une propriété que l'on veut éviter, sinon on ne sait pas exactement quel programme on écrit. Dans l'expression `2*3+4`, les deux arbres syntaxiques donnent des résultats différents quand l'expression est évaluée : un arbre donne 14 (le résultat de 2*(3+4)) et l'autre donne 10 (le résultat de (2*3)+4). Parfois les règles de grammaire peuvent être changées pour enlever l'ambiguïté, mais cela les rend plus compliquées. Il est plus commode d'ajouter des conditions supplémentaires. Ces conditions mettent des restrictions sur le parseur afin qu'il n'y ait qu'un seul arbre syntaxique possible. Nous disons qu'elles lèvent l'ambiguïté de la grammaire.

L'approche habituelle pour les expressions avec des opérateurs binaires, comme les expressions arithmétiques, est d'ajouter deux conditions, la précédence et l'associativité :

– La **précédence** est une condition sur une expression avec des opérateurs différents, comme `2*3+4`. On donne a chaque opérateur un niveau de précédence. Les opérateurs avec des précédences élevées sont placés plus profondément dans l'arbre syntaxique, c'est-à-dire plus loin de la racine. Si ∗ a une précédence plus

élevée que +, alors l'arbre syntaxique `(2*3)+4` sera choisi au lieu de l'alternative `2*(3+4)`. Si * est plus profond dans l'arbre que +, on dit que * est un opérateur plus étroit que +.

– L'**associativité** est une condition sur une expression avec le même opérateur, comme `2-3-4`. Dans ce cas, la précédence ne suffit pas à lever l'ambiguïté parce que tous les opérateurs ont la même précédence. Nous devons choisir entre les arbres `(2-3)-4` et `2-(3-4)`. L'associativité détermine si l'opérateur de gauche ou de droite est plus étroit. Si l'associativité de - est `gauche`, l'arbre `(2-3)-4` sera choisi. Si l'associativité de - est `droite`, l'autre arbre `2-(3-4)` sera choisi.

La précédence et l'associativité ensemble suffisent pour lever l'ambiguïté de toutes les expressions définies avec des opérateurs. L'annexe B donne la précédence et l'associativité de tous les opérateurs que nous utilisons.

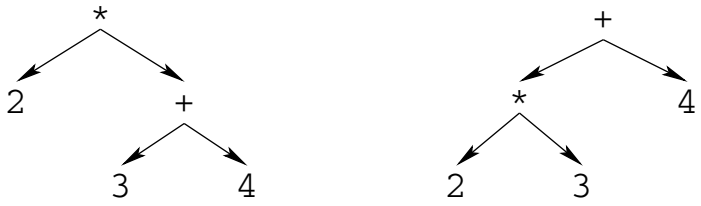


Figure 2.3 L'ambiguïté dans une grammaire hors-contexte.

La notation de syntaxe

Dans ce chapitre et le reste du livre, chaque nouvelle construction du langage sera introduite avec un petit diagramme syntaxique qui montre comment elle s'insère dans le langage tout entier. Le diagramme syntaxique montre des règles de grammaire pour une grammaire hors-contexte simple constituée de jetons. La notation est délibérément conçue pour satisfaire deux principes de base :

– Une règle de grammaire est toujours vraie. Aucune information introduite plus loin n'invalidera une règle de grammaire. En d'autres termes, nous ne montrons jamais une règle erronée.
– Le fait qu'une règle de grammaire soit une définition complète ou une définition partielle d'un symbole non terminal est toujours clair. Une définition partielle termine toujours par trois points « ... ».

Tous les diagrammes de syntaxe sont rassemblés dans l'annexe B. Cette annexe définit aussi la syntaxe lexicale, c'est-à-dire la syntaxe des jetons en termes de caractères. Voici un exemple de diagramme syntaxique avec deux règles de grammaire pour illustrer notre notation :

⟨statement⟩ : := **skip** | ⟨expression⟩ ´=´ ⟨expression⟩ | ...
⟨expression⟩ : := ⟨variable⟩ | ⟨int⟩ | ...

Ces règles donnent des définitions partielles de deux non terminaux, ⟨statement⟩ et ⟨expression⟩. La première règle dit qu'une instruction peut être le mot clé **skip**, deux expressions séparées par le symbole d'égalité ´=´ ou autre chose. La deuxième règle dit qu'une expression peut être une variable, un entier ou autre chose. Un choix entre différentes possibilités dans une règle de grammaire est désigné par une barre verticale |. Pour éviter la confusion avec la syntaxe de la notation elle-même, nous mettons parfois entre guillemets simples un symbole qui apparaît tel quel dans le texte. Par exemple, le symbole d'égalité devient ´=´. On ne met pas les mots clés entre guillemets parce qu'il n'y a pas de confusion possible pour eux.

Voici un deuxième exemple pour présenter le reste de la notation :

⟨statement⟩ : := **if** ⟨expression⟩ **then** ⟨statement⟩
{ **elseif** ⟨expression⟩ **then** ⟨statement⟩ }
[**else** ⟨statement⟩] **end** | ...
⟨expression⟩ : := ´[´ { ⟨expression⟩ }+ ´]´ | ...
⟨label⟩ : := **unit** | **true** | **false** | ⟨variable⟩ | ⟨atom⟩

La première règle définit l'instruction **if**. Il y a une séquence de zéro ou plusieurs clauses **elseif**. Tout nombre d'occurrences est permis y compris zéro, ce qui est noté par les accolades { ... }. Cette séquence est suivie par une clause **else** facultative, c'est-à-dire qu'elle peut apparaître zéro ou une fois, ce qui est désigné par les crochets [...]. La deuxième règle définit la syntaxe des listes explicites. Elles doivent avoir au moins un élément, par exemple `[5 6 7]` est valable mais `[ ]` ne l'est pas (il y a un espace entre le `[` et le `]`). Une séquence avec au moins un élément est désignée par { ... }+. La troisième règle définit la syntaxe des étiquettes des enregistrements. Elle est complète parce qu'il n'y a pas les trois points « ... ». Il y a cinq possibilités et aucune autre ne sera jamais donnée.

2.1.2 La sémantique d'un langage

La sémantique d'un langage définit ce que fait un programme lors de son exécution. Idéalement, la sémantique devrait être définie dans une structure mathématique simple permettant de déduire les propriétés souhaitées du programme (comme l'exactitude, le temps d'exécution et l'utilisation de mémoire) sans introduire des détails compliqués. La technique que nous utilisons, que nous appelons l'**approche du langage noyau**, convient tout à fait.

Les langages de programmation modernes ont évolué pendant presque six décennies pour satisfaire aux exigences de la construction de solutions programmées pour les

problèmes complexes du monde réel.[1] Les programmes modernes peuvent être assez complexes, avec des tailles mesurées en millions de lignes de code source, écrits par de grandes équipes de programmeurs humains et ce, sur plusieurs années. Notre point de vue est que les langages qui permettent d'atteindre ce niveau de complexité réussissent parce qu'ils modélisent certains aspects essentiels de la structure de programmes complexes. Dans ce sens, ces langages ne sont pas que des constructions arbitraires de l'esprit humain. Nous voulons donc les comprendre de façon scientifique, en expliquant leur structure et leur comportement avec un modèle sous-jacent qui est le plus simple possible. C'est la motivation profonde de l'approche du langage noyau.

L'approche du langage noyau

Nous utilisons l'approche du langage noyau pour définir la sémantique des langages de programmation. Dans cette approche, toutes les constructions du langage sont définies par leurs traductions dans un langage simple qui est appelé langage noyau. L'approche comporte deux parties (voir figure 2.4) :

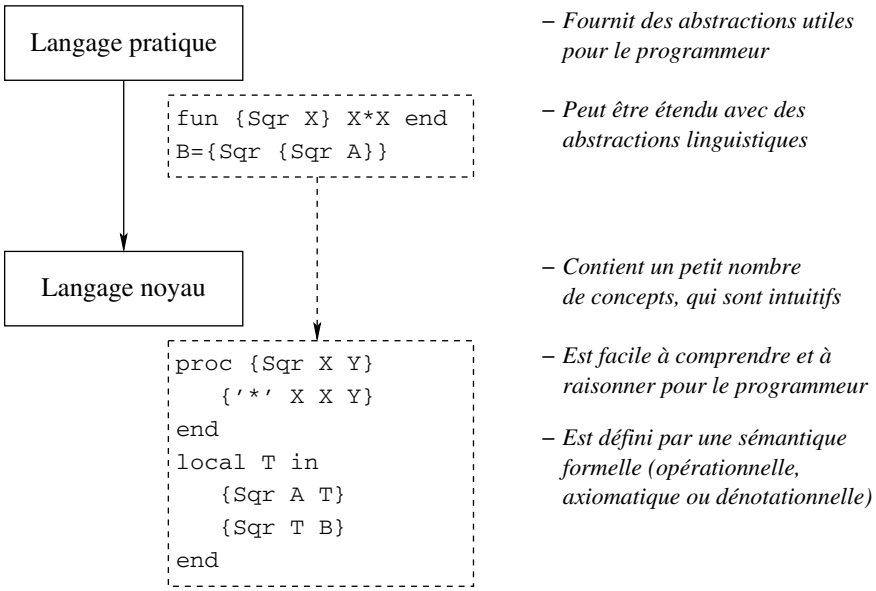


Figure 2.4 L'approche du langage noyau pour la sémantique.

– D'abord, définissez un langage très simple, appelé langage noyau. Ce langage doit être simple pour faciliter le raisonnement et doit être fidèle à l'efficacité en

1. Le chiffre de six décennies est un peu arbitraire. Nous le mesurons par rapport au premier ordinateur à programme enregistré, le Manchester Mark I. Selon les documents du laboratoire, il a exécuté son premier programme le 21 juin 1948 [83].

espace et en temps de l'implémentation. Le langage noyau et les structures de données qu'il manipule sont appelés le modèle de calcul noyau.

– Ensuite, définissez un schéma de traduction du langage pratique complet vers le langage noyau. Chaque construction grammaticale dans le langage complet est traduite dans le langage noyau. La traduction doit être simple. Il y a deux formes de traduction, que nous appelons l'abstraction linguistique et le sucre syntaxique. Nous les expliquons ci-dessous.

Chaque modèle de calcul du livre a son propre langage noyau. On l'obtient simplement en ajoutant un nouveau concept à un langage noyau qui le précède. Le premier, présenté dans ce chapitre, s'appelle le langage noyau déclaratif. Nous en présenterons plusieurs autres plus loin.

La sémantique formelle

Nous avons la liberté de définir la sémantique du langage noyau comme nous le voulons. Il y a quatre approches largement utilisées pour la sémantique des langages de programmation :

– Une **sémantique opérationnelle** montre comment une instruction s'exécute sur une machine abstraite. Cette approche fonctionne toujours bien, parce que tous les langages s'exécutent sur un ordinateur.

– Une **sémantique axiomatique** définit le sens d'une instruction comme une relation entre l'état de l'entrée (avant l'exécution de l'instruction) et l'état de la sortie (après l'exécution de l'instruction). Cette relation est donnée comme une assertion logique. C'est une bonne manière de raisonner sur les séquences d'instructions, parce que la sortie de chaque instruction est l'entrée de l'instruction suivante. Cette sémantique fonctionne bien pour les modèles avec état, parce qu'un état est une séquence de valeurs.

– Une **sémantique dénotationnelle** définit une instruction comme une fonction sur un domaine abstrait. Cette approche fonctionne particulièrement bien pour les modèles déclaratifs, mais elle peut être adaptée pour les autres modèles aussi. L'approche devient compliquée pour les langages concurrents.

– Une **sémantique logique** définit une instruction comme une relation qui est vraie sur un modèle logique et son exécution comme une théorie de preuve. Cette approche fonctionne bien pour les modèles déclaratifs et relationnels, mais est plus difficile à adapter pour les autres.

Une grande partie de la théorie mathématique de ces sémantiques est intéressante principalement pour les mathématiciens et non pour les programmeurs. La sémantique formelle que nous donnons est une sémantique opérationnelle. Nous la définissons pour chaque modèle de calcul. Elle est assez détaillée pour permettre le raisonnement sur

l'exactitude et la complexité mais assez abstraite pour éviter les détails encombrants et sans pertinence. Nous donnons d'abord une sémantique informelle pour chaque nouvelle construction de langage avant de donner la sémantique formelle. Nous raisonnons souvent de manière informelle sur les programmes. Ces raisonnements sont toujours basés sur la sémantique opérationnelle.

Les abstractions linguistiques

Les langages de programmation et les langages naturels peuvent tous les deux évoluer pour satisfaire aux besoins. Quand nous utilisons un langage de programmation, nous pouvons à un moment donné sentir le besoin d'étendre le langage avec une nouvelle construction linguistique. Par exemple, le modèle déclaratif de ce chapitre n'a pas de constructions pour faire des boucles. Néanmoins, on peut définir une construction **`for`** pour exprimer certaines formes de boucles qui sont utiles pour les programmes déclaratifs [97, 35]. La nouvelle construction est à la fois une abstraction et une addition à la syntaxe du langage. Nous l'appelons donc une **abstraction linguistique**. Chaque langage de programmation pratique contient beaucoup d'abstractions linguistiques.

Une abstraction linguistique se définit en deux phases. D'abord, la syntaxe de la nouvelle construction grammaticale. Ensuite, sa traduction en langage noyau. Le langage noyau ne change pas. Nous donnons beaucoup d'exemples d'abstractions linguistiques utiles, comme les fonctions (**`fun`**), les boucles (**`for`**), les classes (**`class`**), les composants logiciels (**`functor`**) et d'autres.[2] Certaines abstractions font partie du système Mozart. D'autres peuvent être ajoutées à Mozart avec l'outil `gump` qui est un parseur-générateur [52]. Pour l'utilisation de cet outil nous vous invitons à consulter la documentation de Mozart [69].

Il y a des langages dans lesquels on peut programmer les abstractions linguistiques directement dans le langage. Un exemple simple mais puissant est le langage Lisp avec ses macros. Une macro Lisp ressemble à une fonction qui génère du code Lisp quand elle est exécutée. Les macros ont eu beaucoup de succès dans Lisp et ses successeurs, en partie à cause de la syntaxe simple de Lisp. Lisp soutient les macros avec des opérations comme le « *quote* » (guillemet simple) (convertir une expression en une structure de données) et « *backquote* » (guillemet arrière) (inclure une structure de données dans une expression « quotée »). Pour une présentation détaillée des macros Lisp et les idées associées, veuillez consulter un des nombreux bons livres sur Lisp [30, 87].

2. Dans [97] il y a en particulier les fonctions paresseuses (**`fun`** `lazy`), les verrous ré-entrants (**`lock`**), les boîtes à messages (**`receive`**), les portes logiques (**`gate`**), les compréhensions de liste et la curryfication (« *currying* », d'après le logicien Haskell B. Curry) comme dans les langages fonctionnels modernes tels que Haskell.

Un exemple simple d'une abstraction linguistique est la fonction. Cette abstraction utilise le mot clé **`fun`**. Sa syntaxe et sa traduction sont expliquées dans la section 2.6.2. Nous avons déjà utilisé des fonctions au chapitre 1. Mais notre langage noyau n'a que des procédures. Nous utilisons des procédures parce que tous les arguments sont explicites et qu'il peut y avoir plusieurs résultats. Il y a d'autres raisons, plus profondes, pour choisir les procédures. Elles sont expliquées plus loin. Mais comme les fonctions sont très utiles, nous les avons ajoutées comme une abstraction linguistique.

Nous définirons une syntaxe pour les définitions et les appels de fonction, et une traduction de cette syntaxe vers le langage noyau. La traduction nous permet de répondre à toutes les questions sur les appels de fonction. Par exemple, que fait `{F1 {F2 X} {F3 Y}}` exactement (quand il y a des appels imbriqués) ? L'ordre des appels est-il défini ? Si oui, quel est cet ordre ? Il y a beaucoup de possibilités. Certains langages prennent la décision de ne pas spécifier l'ordre d'évaluation des arguments d'une fonction. Ils supposent uniquement que les arguments sont évalués avant la fonction elle-même. D'autres langages supposent qu'un argument sera évalué uniquement si on a besoin de son résultat, au moment de ce besoin. Nous voyons qu'une chose aussi simple que les fonctions imbriquées n'a pas forcément une sémantique évidente. Un des rôles de la traduction est de clarifier cette sémantique.

Les abstractions linguistiques sont utiles pour d'autres choses que pour simplement augmenter l'expressivité d'un programme. Elles peuvent aussi améliorer des propriétés comme l'exactitude, la sécurité et l'efficacité. En cachant l'implémentation d'une abstraction du programmeur, le soutien linguistique rend impossible l'utilisation erronée de l'abstraction. Le compilateur peut profiter de ce fait pour générer du code plus efficace.

Le sucre syntaxique

Il est souvent commode d'avoir une notation raccourcie pour certaines expressions idiomatiques. Il s'agit d'une courte séquence d'instructions qui est souvent utilisée. On peut définir une notation plus courte pour une expression idiomatique. Cette notation fera partie de la syntaxe du langage pratique et sa définition fera partie des règles de grammaire. Cette notation s'appelle le **sucre syntaxique**. Le sucre syntaxique est semblable à l'abstraction linguistique dans la mesure où pour les deux, la sémantique est définie par une traduction vers le langage noyau. Mais il ne faut pas les confondre : le sucre syntaxique ne définit pas une nouvelle abstraction, mais réalise simplement une réduction de la taille du programme et une amélioration de sa lisibilité.

Nous donnons un exemple de sucre syntaxique basé sur l'instruction **`local`**. Une variable locale peut à tout moment être définie avec l'instruction **`local`** `X` **`in ... end`**. Quand cette instruction est à l'intérieur d'une autre, il est commode d'avoir un sucre syntaxique qui permet d'omettre les mots clés **`local`** et **`end`**. Au lieu de

```
if N==1 then [1] else local L in ... end end
```

nous pouvons écrire

```
if N==1 then [1] else L in ... end
```

qui est à la fois plus courte et plus lisible que la notation complète. D'autres exemples de sucre syntaxique sont donnés dans la section 2.6.1.

La conception de langages

Les abstractions linguistiques sont un outil de base pour la conception de langages. Elles ont une place naturelle dans le cycle de vie d'une abstraction. Une abstraction a trois phases dans sa vie. Quand elle vient d'être définie, elle n'a pas de soutien linguistique, c'est-à-dire qu'il n'y a pas de syntaxe dans le langage pour faciliter son utilisation. Si le moment arrive où nous pensons qu'elle est particulièrement utile, nous pourrons décider de lui donner un soutien linguistique. À cet instant elle devient une abstraction linguistique. Cette phase est exploratoire : on ne s'engage pas à ce que l'abstraction linguistique fasse partie du langage un jour. Si l'abstraction linguistique a du succès, si elle simplifie les programmes et les réflections des programmeurs, alors elle fera partie du langage.

D'autres approches basées sur la traduction

L'approche du langage noyau est un exemple d'une approche de la sémantique basée sur la traduction d'un langage vers un autre. La figure 2.5 montre les trois formes de cette approche qui ont été utilisées pour la définition des langages de programmation :

- L'approche du langage noyau, que nous utilisons, est ciblée pour le programmeur. Ses concepts correspondent directement aux concepts de programmation.
- L'approche fondamentale est ciblée pour le mathématicien. Quelques exemples de formalismes fondamentaux sont la machine de Turing, le λ calcul (pour la programmation fonctionnelle), la logique de premier ordre (pour la programmation logique) et le π calcul (pour la programmation concurrente). Comme ces formalismes sont ciblés pour une utilisation mathématique, ils contiennent le moins possible de concepts.
- L'approche de la machine abstraite est ciblée pour l'implémenteur. Les programmes sont traduits vers une machine idéalisée, que l'on appelle traditionnellement une **machine abstraite** ou une **machine virtuelle**.[3] Il est relativement facile de traduire le code d'une machine idéalisée vers du code réel.

3. À proprement parler, une machine virtuelle est une émulation d'une machine réelle qui s'exécute sur la machine réelle et qui est presque aussi efficace que la machine réelle. Elle parvient à cette efficacité en exécutant la plupart des instructions virtuelles directement sur la machine comme des instructions réelles. IBM a ouvert la voie à cette approche au début des années 1960 dans le système d'exploitation VM. À cause du succès de Java, qui utilise le terme « machine virtuelle », l'usage moderne utilise ce terme aussi dans le sens de machine abstraite.

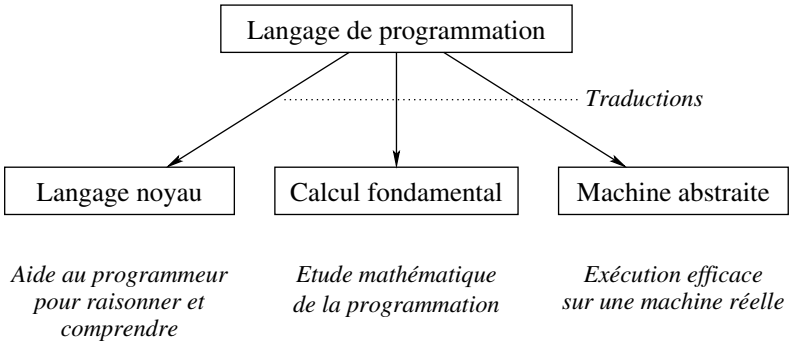


Figure 2.5 Les approches de la sémantique basées sur la traduction.

Comme nous ciblons les techniques de programmation pratiques, nous utiliserons uniquement l'approche du langage noyau. Dans les deux autres approches, tout programme réaliste est encombré de détails techniques sur les mécanismes du langage. L'approche du langage noyau évite cet encombrement par un choix judicieux des concepts.

L'approche basée sur l'interprétation

Une alternative à l'approche basée sur la traduction est l'approche basée sur l'interprétation. La sémantique du langage est définie par un interpréteur. Des extensions au langage sont définies en étendant l'interpréteur. Un interpréteur est un programme écrit en langage L_1 qui accepte comme entrée des programmes écrits en un autre langage L_2 et les exécute. Cette approche est utilisée par Abelson et Sussman [2]. Dans leur cas, l'interpréteur est méta-circulaire : L_1 et L_2 sont le même langage L. L'addition de nouveaux concepts au langage, par exemple pour la concurrence et l'évaluation paresseuse, donne un nouveau langage L' qui est implémenté en étendant l'interpréteur de L.

L'approche basée sur l'interprétation a l'avantage de montrer une implémentation autonome des abstractions linguistiques. Nous n'utilisons pas cette approche parce qu'en général elle ne conserve pas la complexité en temps des programmes (le nombre d'opérations qu'ils font en fonction de la taille de l'entrée). Une deuxième difficulté est que les concepts de base interagissent entre eux dans l'interpréteur, ce qui le rend plus difficile à comprendre. L'approche basée sur la traduction permet de garder les concepts séparés.

2.2 LA MÉMOIRE À AFFECTATION UNIQUE

Nous présentons d'abord les structures de données du modèle déclaratif. Le modèle utilise une mémoire à affectation unique, qui est un ensemble de variables initialement non liées et dont chacune peut être liée à une valeur. La figure 2.6 montre une mémoire avec trois variables non liées x_1, x_2 et x_3. Nous écrivons $\{x_1, x_2, x_3\}$. Les valeurs peuvent être (entre autres) des entiers, des listes et des enregistrements. La figure 2.7 montre la mémoire où x_1 est liée à l'entier 314 et x_2 est liée à la liste `[1 2 3]`.[4] Nous écrivons $\{x_1 = 314, x_2 = \texttt{[1 2 3]}, x_3\}$.

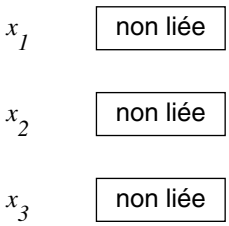


Figure 2.6 Une mémoire à affectation unique avec trois variables non liées.

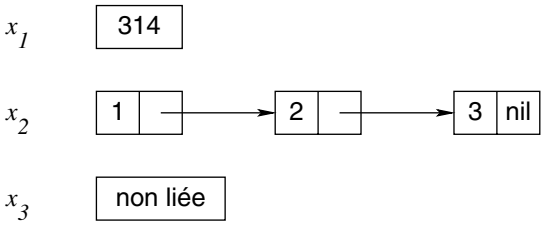


Figure 2.7 Deux variables sont liées aux valeurs.

2.2.1 Les variables déclaratives

Les variables dans la mémoire à affectation unique s'appellent des **variables déclaratives**. Nous utilisons ce terme quand il y a une confusion possible avec d'autres sortes de variables. Plus loin nous les appellerons aussi des **variables dataflow** à cause de leur rôle dans l'exécution dataflow.

Une fois liée, une variable déclarative reste liée dans toute l'exécution et on ne peut pas la distinguer de sa valeur. Elle peut être utilisée dans les calculs comme si

4. En fait, c'est une légère simplification. En réalité, chaque paire de liste et chaque entier a sa propre variable ; nous avons donc $\{x_2 = y_1|x_4, x_4 = y_2|x_5, x_5 = y_3|x_6, x_6 = \texttt{nil}, y_1 = 1, y_2 = 2, y_3 = 3\}$. Pour que les exemples ne soient pas trop difficiles à lire, nous utiliserons souvent cette simplification !

elle était la valeur. L'opération $x + y$ est la même que 11 + 22 si la mémoire contient $\{x = 11, y = 22\}$.

2.2.2 La mémoire à valeurs

Une mémoire où toutes les variables sont liées aux valeurs s'appelle une **mémoire à valeurs**. Cette mémoire est une correspondance immuable des variables aux valeurs. Une valeur est une constante mathématique. Par exemple, l'entier 314 est une valeur. Des valeurs peuvent aussi être des entités composées, à savoir des entités qui contiennent une ou plusieurs autres valeurs. Par exemple, la liste `[1 2 3]` et l'enregistrement `person(name:"George" age:25)` sont des valeurs. La figure 2.8 montre une mémoire à valeurs où x_1 est liée à l'entier 314, x_2 est liée à la liste `[1 2 3]` et x_3 est liée à l'enregistrement `person(name:"George" age:25)`. Les langages fonctionnels tels que Standard ML, Haskell et Scheme utilisent une mémoire à valeurs. (Les langages orientés objet tels que Smalltalk, C++ et Java utilisent une mémoire à cellules, qui est un ensemble de cellules dont le contenu peut être modifié.)

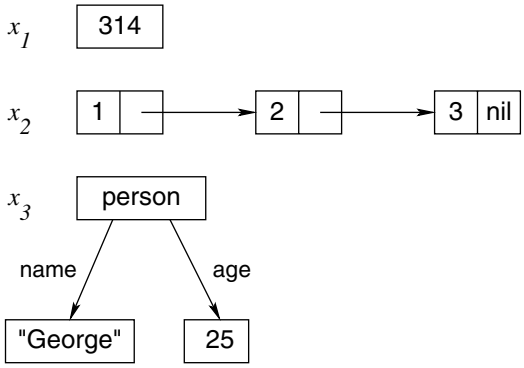


Figure 2.8 Une mémoire à valeurs : toutes les variables sont liées aux valeurs.

À ce point, le lecteur qui a déjà programmé se demandera peut-être pourquoi nous introduisons une mémoire à affectation unique, quand d'autres langages s'en tiennent à une mémoire à valeurs ou à une mémoire à cellules. Il y a beaucoup de raisons. Une première raison est que nous voulons calculer avec des valeurs partielles. Par exemple, une procédure peut renvoyer son résultat en liant une variable non liée donnée comme argument. Une deuxième raison est la concurrence dataflow, qui est traitée au chapitre 4. Elle devient possible à cause de la mémoire à affectation unique. Une troisième raison est que nous avons besoin d'une mémoire à affectation unique à cause de la programmation relationnelle (programmation logique) et de la programmation par contraintes. Ces deux modèles sont expliqués dans les chapitres 9 et 12 de [97]. Il y a encore d'autres raisons ; par exemple, la mémoire à affectation unique permet

d'utiliser beaucoup plus souvent la récursion terminale, ce qui augmente l'efficacité des programmes récursifs.

2.2.3 La création des valeurs

L'opération de base sur une mémoire consiste à lier une variable avec une valeur que l'on vient de créer. Nous l'écrivons x_i=*valeur*. Ici x_i fait référence directement à une variable en mémoire (ce n'est pas le nom textuel de la variable dans un programme !) et *valeur* fait référence à une valeur, comme 314 ou `[1 2 3]`. Par exemple, la figure 2.7 montre la mémoire de la figure 2.6 après les deux liens :

x_1 = 314
x_2 = `[1 2 3]`

L'opération à affectation unique x_i=*valeur* construit *valeur* dans la mémoire et ensuite lie la variable x_i avec cette valeur. Si la variable est déjà liée, l'opération vérifiera la compatibilité des deux valeurs. Si elles ne le sont pas, une erreur sera signalée (avec une exception ; voir section 2.7).

2.2.4 Les identificateurs

Jusqu'ici, nous avons regardé uniquement la mémoire, qui contient des entités (variables et valeurs) avec lesquelles nous pouvons faire des calculs. Il serait bien de pouvoir faire référence à une entité en mémoire à partir de l'extérieur de celle-ci. C'est le rôle des identificateurs. Un identificateur est un nom textuel (une séquence de caractères) qui fait référence à une entité dans la mémoire. La correspondance entre identificateurs et variables en mémoire s'appelle un **environnement**.

Les noms des variables dans le code source d'un programme sont des identificateurs. Par exemple, la figure 2.9 montre un identificateur « `X` » (la lettre majuscule X) qui fait référence à la variable x_1. L'environnement est $\{\texttt{X} \rightarrow x_1\}$. Pour parler d'un identificateur quelconque, nous utiliserons la notation $\langle \mathsf{x} \rangle$. L'environnement $\{\langle \mathsf{x} \rangle \rightarrow x_1\}$ est le même qu'avant si $\langle \mathsf{x} \rangle$ représente `X`. Nous verrons que les identificateurs et leurs variables sont ajoutés à l'environnement par les instructions **`local`** et **`declare`**.

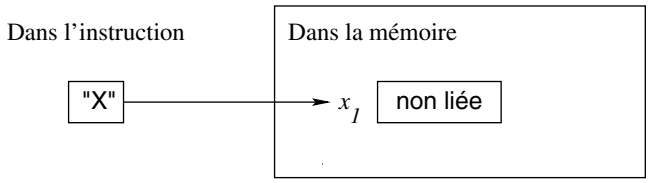


Figure 2.9 Un identificateur qui fait référence à une variable non liée.

2.2.5 La création des valeurs avec les identificateurs

Une fois liée, une variable ne peut pas être distinguée de sa valeur. La figure 2.10 montre ce qui se passe quand x_1 est liée à `[1 2 3]` dans la figure 2.9. Avec l'identificateur `X`, nous pouvons écrire cette opération comme `X=[1 2 3]`. C'est le code qu'un programmeur écrirait pour exprimer ce lien. Nous pouvons aussi utiliser la notation ⟨x⟩`=[1 2 3]` quand nous voulons parler d'un identificateur quelconque. Pour que cette notation devienne légale dans un programme, il faut remplacer ⟨x⟩ par un identificateur.

L'égalité « = » fait référence à l'opération de lien. Quand l'opération se termine, l'identificateur « `X` » fait encore référence à x_1, qui est maintenant liée à `[1 2 3]`. Nous ne distinguons pas cette situation de celle de la figure 2.11, où `X` référence directement `[1 2 3]`. Le système suit les liens des variables liées pour obtenir la valeur, ce qui s'appelle l'opération de **déréférence**. Cette opération est invisible pour le programmeur.

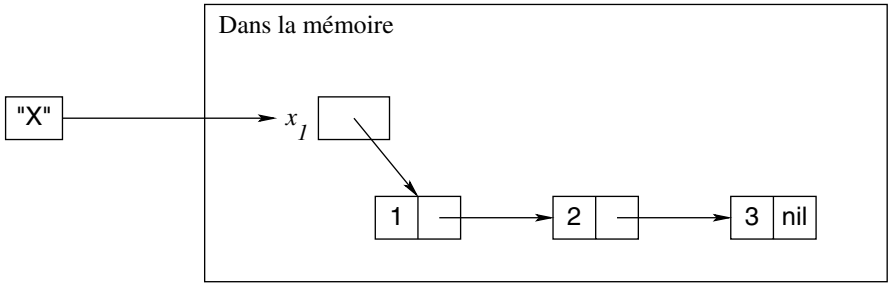


Figure 2.10 Un identificateur qui fait référence à une variable liée.

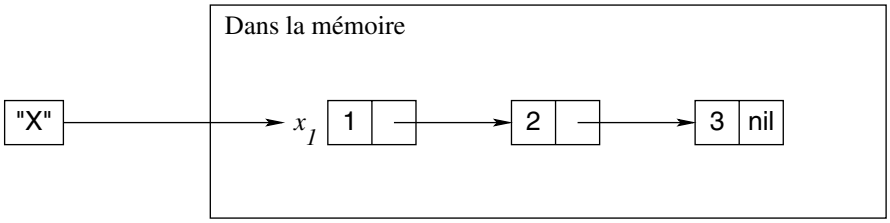


Figure 2.11 Un identificateur qui fait référence à une valeur.

2.2.6 Les valeurs partielles

Une valeur partielle est une structure de données qui peut contenir des variables non liées. La figure 2.12 montre l'enregistrement `person(name:"George" age:`x_2`)`, référencé par l'identificateur `X`. C'est une valeur partielle parce qu'elle contient la variable non liée x_2. L'identificateur `Y` référence x_2. La figure 2.13 montre la situation après le lien de x_2 avec 25 (par l'opération de lien `Y=25`). Maintenant x_1 est une valeur partielle qui ne contient pas de variables non liées, ce que nous appelons une valeur complète. Une variable déclarative peut être liée à plusieurs valeurs partielles, du moment qu'elles sont compatibles entre elles. Nous disons que les valeurs partielles sont compatibles si on peut lier leurs variables non liées pour les rendre égales. Par exemple, `person(age:25)` et `person(age:`x`)` sont compatibles (parce que x peut être liée à 25) mais `person(age:25)` et `person(age:26)` ne le sont pas.

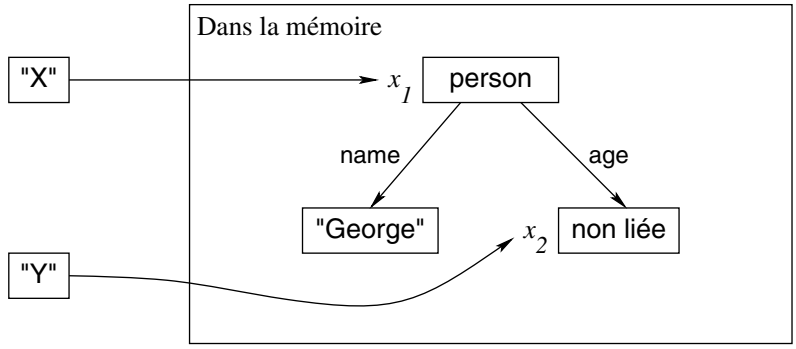


Figure 2.12 Une valeur partielle.

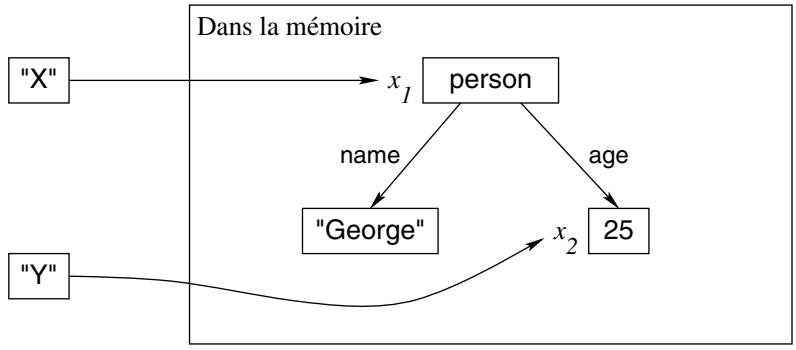


Figure 2.13 Une valeur complète ; c'est une valeur partielle sans variables non liées.

2.2.7 Les liens variable-variable

Une variable peut être liée à une autre variable. Par exemple, prenons les deux variables non liées x_1 et x_2 référencées par les deux identificateurs `X` et `Y`. Après le lien `X=Y`, nous obtenons la situation de la figure 2.14. Les deux variables x_1 et x_2 sont égales. La figure montre cette égalité par un lien de chaque variable vers l'autre. Nous disons que $\{x_1, x_2\}$ forment un ensemble d'équivalence.[5] Nous l'écrivons aussi $x_1 = x_2$. Avec trois variables liées ensemble on écrit $x_1 = x_2 = x_3$ ou $\{x_1, x_2, x_3\}$. Dans une figure, nous dessinons ces variables en chaîne circulaire. Quand une variable dans un ensemble d'équivalence est liée, toutes les variables sont liées à la même valeur. La figure 2.15 montre le résultat du lien `X=[1 2 3]`.

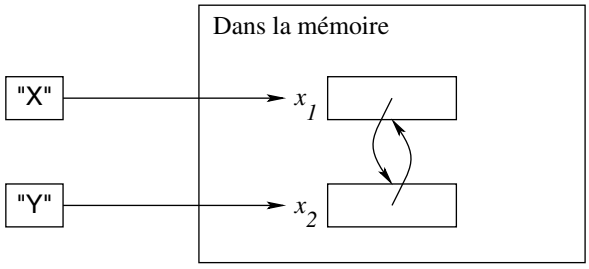


Figure 2.14 Deux variables liées entre eux.

5. D'un point de vue formel, l'opération d'égalité partitionne la mémoire en classes d'équivalence.

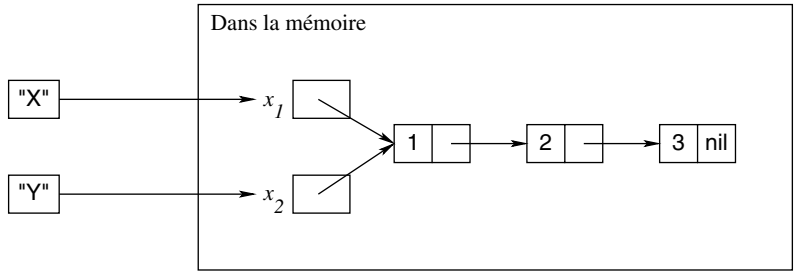


Figure 2.15 La mémoire après le lien d'une des variables.

2.2.8 Les variables dataflow

Dans le modèle déclaratif, la création et le lien d'une variable se font séparément. Que se passe-t-il si on essaie d'utiliser une variable avant qu'elle soit liée ? C'est une erreur d'utilisation de variable. Certains langages font la création et le lien des variables en une fois, ce qui élimine ces erreurs. C'est le cas pour les langages fonctionnels. D'autres langages permettent la séparation de la création et du lien. Voici ce qui se passe dans quelques langages quand il y a une erreur d'utilisation :

1. L'exécution continue sans message d'erreur. Le contenu de la variable n'est pas définie, son contenu est « n'importe quoi » : ce que l'on trouve dans la mémoire à l'adresse de la variable. C'est ce que fait le langage C++.
2. L'exécution continue sans message d'erreur. La variable est initialisée à une valeur par défaut quand elle est déclarée, par exemple, à 0 pour un entier. C'est ce que fait le langage Java pour les champs dans les objets et les structures de données, comme les tableaux (« *arrays* »). La valeur par défaut dépend du type.
3. L'exécution s'arrête avec un message d'erreur (ou une exception est levée). C'est ce que fait le langage Prolog pour les opérations arithmétiques.
4. L'exécution n'est pas possible parce que le compilateur détecte qu'il y a un chemin d'exécution qui mène à l'utilisation de la variable sans l'avoir initialisée. C'est ce que fait Java pour les variables locales.
5. L'exécution attend jusqu'à ce que la variable soit liée et continue ensuite. C'est ce que fait Oz, pour soutenir la programmation dataflow.

Ces cas sont énumérés dans l'ordre du plus mauvais au meilleur. Le premier cas est très mauvais car chaque exécution d'un programme peut donner un résultat différent (le non-déterminisme). Par ailleurs, comme l'existence de l'erreur n'est pas signalée, le programmeur n'est même pas conscient du problème. Le deuxième cas est un peu mieux. Si le programme présente une erreur d'utilisation, au moins il donnera toujours le même résultat, mais le résultat peut être faux. De nouveau, le programmeur n'est pas mis au courant du problème.

Les troisième et quatrième cas peuvent être acceptables dans certaines situations. Dans les deux cas, un programme avec une erreur d'utilisation la signalerait pendant l'exécution ou pendant la compilation. Dans un système séquentiel c'est acceptable parce qu'il y a vraiment une erreur. Le troisième cas n'est pas acceptable dans un système concurrent parce que le résultat devient non-déterministe : selon le hasard de l'ordonnancement du programme pendant l'exécution, une erreur sera parfois signalée et parfois pas.

Dans le cinquième cas, le programme attendra que la variable soit liée et continuera ensuite. Nos modèles de calcul utilisent le cinquième cas. Ce n'est pas acceptable dans un système séquentiel parce que le programme attendra indéfiniment, au contraire d'un système concurrent ; dans le déroulement normal du programme il se peut qu'un autre fil lie la variable. Le cinquième cas introduit une nouvelle forme d'erreur, une suspension (attente) qui attend pour toujours. Par exemple, si un identificateur est mal orthographié, alors la variable correspondante ne sera jamais liée. Un bon débuggeur doit détecter cette situation.

Des variables déclaratives qui font attendre le programme jusqu'à ce qu'elles soient liées s'appellent des **variables dataflow**. Le modèle déclaratif utilise les variables dataflow parce qu'elles sont extrêmement utiles dans la programmation concurrente, c'est-à-dire pour des programmes avec des activités indépendantes. Si nous faisons deux opérations concurrentes, par exemple `A=23` et `B=A+1`, avec le cinquième cas l'exécution sera toujours correcte et donnera toujours la réponse `B=24`. C'est vrai indépendamment de l'ordre dans lequel les deux opérations `A=23` et `B=A+1` sont exécutées. Dans les autres cas, ce n'est en général pas vrai. Cette propriété d'indépendance de l'ordre rend possible le modèle déclaratif du chapitre 4. Elle est la raison principale pour utiliser les variables dataflow dans le modèle.

2.3 LE LANGAGE NOYAU DÉCLARATIF

Le modèle déclaratif définit un langage noyau simple. Tous les programmes du modèle peuvent être exprimés dans ce langage. Nous définissons d'abord la syntaxe et la sémantique du langage noyau. Ensuite nous expliquons comment bâtir un langage pratique au-dessus du langage noyau.

2.3.1 La syntaxe

La syntaxe noyau est définie dans les tableaux 2.1 et 2.2. Elle est soigneusement conçue pour être un sous-ensemble de la syntaxe du langage complet. Toutes les instructions du langage noyau sont des instructions valables du langage complet.

⟨s⟩ ::=		
	skip	Instruction vide
|	⟨s⟩$_1$ ⟨s⟩$_2$	Séquence d'instructions
|	**local** ⟨x⟩ **in** ⟨s⟩ **end**	Création de variable
|	⟨x⟩$_1$=⟨x⟩$_2$	Lien variable-variable
|	⟨x⟩=⟨v⟩	Création de valeur
|	**if** ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Instruction conditionnelle
|	**case** ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Correspondance de formes
|	{ ⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$ }	Application de procédure

Tableau 2.1 Le langage noyau déclaratif.

⟨v⟩	::=	⟨number⟩ | ⟨record⟩ | ⟨procedure⟩
⟨number⟩	::=	⟨int⟩ | ⟨float⟩
⟨record⟩, ⟨pattern⟩	::=	⟨literal⟩
		| ⟨literal⟩(⟨feature⟩$_1$: ⟨x⟩$_1$ ··· ⟨feature⟩$_n$: ⟨x⟩$_n$)
⟨procedure⟩	::=	**proc** { $ ⟨x⟩$_1$ ··· ⟨x⟩$_n$ } ⟨s⟩ **end**
⟨literal⟩	::=	⟨atom⟩ | ⟨bool⟩ | ...
⟨feature⟩	::=	⟨atom⟩ | ⟨bool⟩ | ⟨int⟩ | ...
⟨bool⟩	::=	**true** | **false**

Tableau 2.2 Les valeurs dans le langage noyau déclaratif.

La syntaxe des instructions

Le tableau 2.1 définit la syntaxe de ⟨s⟩, qui dénote une instruction. Il y a huit instructions en tout.

La syntaxe des valeurs

Le tableau 2.2 définit la syntaxe de ⟨v⟩, qui dénote une valeur. Il y a trois sortes d'expressions qui les dénotent : les nombres, les enregistrements et les procédures. Pour les enregistrements (⟨record⟩) et les formes (⟨pattern⟩), les arguments ⟨x⟩$_1$, ..., ⟨x⟩$_n$ doivent tous être des identificateurs différents. Cela garantit que tous les liens entre deux variables seront écrits explicitement.

La syntaxe des identificateurs

Le tableau 2.1 utilise les non terminaux ⟨x⟩ et ⟨y⟩ pour désigner un identificateur. Nous utiliserons aussi ⟨z⟩ pour le désigner. Il y a deux manières de l'écrire :

– Une lettre majuscule suivie par zéro ou plusieurs caractères alphanumériques (lettres, chiffres ou un caractère de soulignement), comme `X`, `X1` ou `VoiciUneVariableLongue_NestCePas`.
– Toute séquence de caractères affichables, mise entre ` (guillemet arrière, « *backquote* »), comme `` `voici une variable qui vaut 25$ !` ``.

Une définition précise de la syntaxe des identificateurs est donnée dans l'annexe B. Toutes les variables fraîchement déclarées sont non liées avant l'exécution des instructions. Tous les identificateurs doivent être déclarés explicitement.

2.3.2 Les valeurs et les types

Un type de données est un ensemble de valeurs avec un ensemble d'opérations sur ces valeurs. Une valeur est d'un type si elle est dans l'ensemble du type. Le modèle déclaratif est typé : il calcule avec un ensemble bien défini de types qui sont appelés types de base ou types primitifs. Par exemple, les programmes peuvent calculer avec des entiers ou des enregistrements, qui sont respectivement tous de type entier ou enregistrement. Toute tentative d'utiliser une opération avec des valeurs d'un type erroné est détectée par le système et provoquera une erreur (voir section 2.7). Le modèle n'a pas d'autres restrictions sur l'utilisation des types.

Parce que toutes les utilisations des types sont vérifiées, il n'est pas possible pour un programme de se comporter en dehors du modèle. Par exemple, de se planter à cause d'opérations non définies sur ces structures de données. Il est toujours possible pour un programme de lever une condition d'erreur. Par exemple, en faisant une division par zéro. Dans le modèle déclaratif, un programme qui lève une condition d'erreur terminera immédiatement. Il n'y a rien dans ce modèle pour traiter les erreurs. Dans la section 2.7 nous étendrons le modèle avec un nouveau concept, les exceptions, afin de traiter les erreurs. Dans le modèle étendu, une erreur de type peut être traitée en restant à l'intérieur du modèle.

En plus des types de base, les programmes peuvent définir leurs propres types. Ces types s'appellent des **types de données abstraits** (« *abstract data types* », ADT). Le chapitre 3 et les chapitres ultérieurs montrent comment définir les ADT. Il y a d'autres formes d'abstractions de données que les ADT. La section 5.4 résume les possibilités.

Les types de base

Les types de base du modèle déclaratif sont les nombres (entiers et flottants), les enregistrements (y compris les atomes, booléens, tuples, listes et chaînes) et les procédures. Le tableau 2.2 définit leur syntaxe. Le non terminal ⟨v⟩ dénote une valeur partiellement construite. Plus loin nous verrons d'autres types, comme les noms, les chunks, les foncteurs, les cellules, les dictionnaires, les tableaux (« *arrays* »), les classes et les objets. Parmi tous les nouveaux types, il n'y en a que quatre nouveaux de base : les

noms, les chunks, les cellules et les ports (les ports sont hors de notre portée). Pour une explication complète, consultez CTM et la documentation de Mozart [97, 23].

Le typage dynamique

Fondamentalement, il y a deux approches : le typage dynamique et le typage statique. Dans le typage statique, les types de toutes les variables sont connus à la compilation. Dans le typage dynamique, le type d'une variable est connu seulement quand la variable est liée. Le modèle déclaratif est dynamiquement typé. Le compilateur tente de vérifier que toutes les opérations utilisent des valeurs du type approprié. Mais à cause du typage dynamique, certains tests du type sont nécessairement faits à l'exécution.

La hiérarchie des types

Les types de base du modèle déclaratif peuvent être classifiés dans une hiérarchie. La figure 2.16 montre cette hiérarchie.

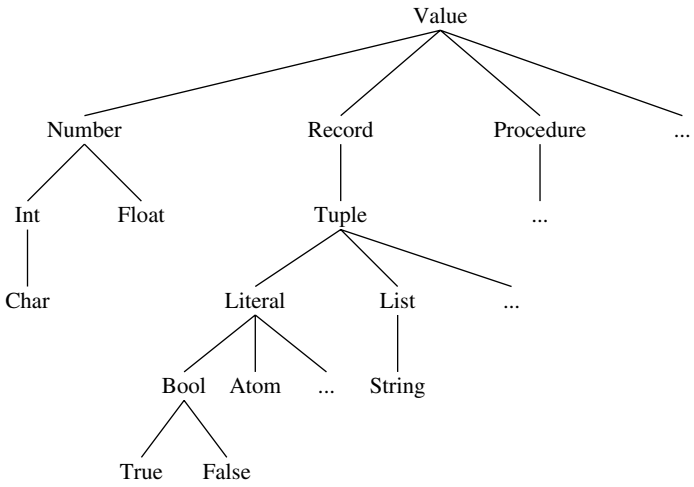


Figure 2.16 La hiérarchie des types du modèle déclaratif.

Chaque nœud représente un type. La hiérarchie est ordonnée par inclusion d'ensembles, c'est-à-dire que toutes les valeurs du type d'un nœud sont aussi des valeurs du type de son nœud parent. Par exemple, tous les tuples sont des enregistrements et toutes les listes sont des tuples. Cela implique que toutes les opérations d'un type sont également possibles pour un sous-type. Toutes les opérations sur les listes sont valables aussi pour les chaînes de caractères. Plus loin nous étendrons cette hiérarchie. Par exemple, un littéral peut être un atome (voir section suivante) ou un autre genre de constante appelée un nom (voir la documentation de Mozart [23]). Les parties incomplètes de la hiérarchie sont indiquées avec « … ».

2.3.3 Les types de base

Nous donnons quelques exemples des types de base et comment les écrire. Pour des informations plus complètes voir la documentation de Mozart [23].

- *Les nombres* (« *Number* »). Les nombres sont des entiers (« *Int* ») ou des nombres à virgule flottante (« *Float* »). Quelques exemples d'entiers : `314`, `0` et `~10` (moins 10). Remarquez que le signe moins est écrit avec un tilde ´~´. Quelques exemples de flottants sont `1.0`, `3.4`, `2.0e2` et `~2.0E~2`.
- *Les atomes* (« *Atom* »). Un atome est une constante symbolique qui peut être utilisé dans les calculs. Il y a plusieurs manières d'écrire un atome. Un atome peut être écrit comme une séquence de caractères qui commence avec une minuscule suivie par un nombre arbitraire de caractères alphanumériques. Un atome peut aussi être écrit comme toute séquence de caractères affichables entre guillemets simples. Quelques exemples d'atomes : `une_personne`, `donkeyKong3` en ´#### bonjour ####´.
- *Les booléens* (« *Bool* »). Un booléen est un des deux symboles **`true`** ou **`false`**. Ces symboles désignent des constantes infalsifiables (des noms, voir la section 3.5.2).
- *Les enregistrements* (« *Record* »). Un enregistrement est une structure de données composée. Il contient une étiquette suivie par un ensemble de champs, où un champ est une paire d'un trait et un identificateur. Un trait (« *feature* » en anglais) peut être un atome, un entier ou un booléen. Quelques exemples d'enregistrements : `person(age:X1 name:X2)` (avec les traits `age` et `name`), `person(1:X1 2:X2)`, ´|´(1:H 2:T), ´#´(1:H 2:T), `nil` et `person`. Un atome est un enregistrement sans traits.
- *Les tuples* (« *Tuple* »). Un tuple est un enregistrement dont les traits sont des entiers consécutifs qui commencent avec 1. Dans ce cas il n'y a pas besoin d'écrire les traits. Quelques exemples de tuples : `person(1:X1 2:X2)` et `person(X1 X2)`, qui représentent le même tuple.
- *Les listes* (« *List* »). Une liste est l'atome `nil` ou le tuple ´|´(H T) (l'étiquette est une barre verticale), avec `T` une liste. Ce tuple est appelé une paire de liste. Il y a du sucre syntaxique pour les listes :
  - L'étiquette ´|´ peut être écrite comme un opérateur infix. Ainsi `H|T` représente le même tuple que ´|´(H T).
  - L'opérateur ´|´ associe à droite : `1|2|3|nil` représente le même tuple que `1|(2|(3|nil))`.
  - Une liste qui termine en `nil` peut être écrite avec des crochets `[ ... ]`. Ainsi `[1 2 3]` représente la même liste que `1|2|3|nil`. Elle est appelée une liste complète.

– *Les chaînes* (« *String* »). Une chaîne est une liste de codes de caractère. Une chaîne peut être écrite entre guillemets doubles. Ainsi `"E=mc^2"` représente la même chaîne que `[69 61 109 99 94 50]`.
– *Les procédures* (« *Procedure* »). Une procédure est une valeur de type procédure (une valeur procédurale). L'instruction

  ⟨x⟩ = **proc** {$ ⟨y⟩$_1$ ··· ⟨y⟩$_n$} ⟨s⟩ **end**

  lie ⟨x⟩ à une nouvelle valeur procédurale. C'est simplement la déclaration d'une nouvelle procédure. Le `$` indique que la valeur procédurale est anonyme, c'est-à-dire créée sans identificateur. Il y a un raccourci syntaxique qui est plus connu :

  proc {⟨x⟩ ⟨y⟩$_1$ ··· ⟨y⟩$_n$} ⟨s⟩ **end**

  Le `$` est remplacé par l'identificateur ⟨x⟩. Cela crée la valeur procédurale et essaie tout de suite de la lier à ⟨x⟩. Le raccourci est souvent plus facile à lire mais il brouille la distinction entre la création d'une valeur et son lien avec un identificateur.

2.3.4 Les enregistrements et les procédures

Nous allons expliquer ici pourquoi nous choisissons les enregistrements et les procédures comme concepts de base dans le langage noyau. Cette section concerne les lecteurs qui ont une certaine expérience en programmation et qui se posent des questions sur la conception du langage noyau.

La puissance des enregistrements

Les enregistrements sont la principale manière de structurer les informations. Ils sont les éléments de base de la plupart des structures de données comme les listes, les arbres, les files, les graphes, etc., comme nous le verrons dans le chapitre 3. Les enregistrements jouent ce rôle à un certain degré dans la plupart des langages de programmation. Nous verrons que leur utilisation peut aller bien au-delà de ce rôle. La puissance des enregistrements apparaît forte ou faible selon la force du soutien donné par le langage. Pour une puissance maximale, le langage doit bien soutenir la création, la décomposition et la manipulation des enregistrements. Dans le modèle déclaratif, un enregistrement est créé simplement en l'écrivant. Un enregistrement est décomposé simplement en écrivant une forme. Enfin, il y a beaucoup d'opérations pour manipuler les enregistrements : pour ajouter, enlever ou sélectionner des champs ; pour faire des conversions entre enregistrements et listes, etc. D'une façon générale, les langages qui offrent ce niveau de soutien aux enregistrements s'appellent des **langages symboliques**.

Quand il y a un soutien fort aux enregistrements, ils peuvent être utilisés pour augmenter l'efficacité de beaucoup d'autres techniques. Nous en considérons trois en

particulier : la programmation orientée objet, la conception des interfaces graphiques intuitives (GUI : « *Graphical User Interface* ») et la programmation par composants. Pour la programmation orientée objet, le chapitre 6 expose comment les enregistrements peuvent représenter des messages et des en-têtes des méthodes, qui sont utilisés par les objets pour communiquer. Pour la conception des GUI, la section 3.7.2 montre comment les enregistrements peuvent représenter des gadgets logiciels (« *widgets* »), les éléments de base d'une interface graphique. Pour la programmation par composants, la section 3.8 décrit comment les enregistrements peuvent représenter des modules de première classe qui regroupent des opérations associées.

Pourquoi les procédures ?

Un programmeur se demanderait peut-être pourquoi notre langage noyau a des procédures comme éléments primitifs. Des admirateurs de la programmation orientée objet se demanderaient pourquoi nous n'utilisons pas des objets, et des admirateurs de la programmation fonctionnelle, pourquoi nous n'utilisons pas des fonctions. Nous aurions pu choisir l'une ou l'autre possibilité, mais nous ne l'avons pas fait. Les raisons sont simples.

Les procédures sont plus appropriées que les objets parce qu'elles sont plus simples. Les objets sont assez compliqués, comme l'explique le chapitre 6. Les procédures sont plus appropriées que les fonctions parce qu'elles ne définissent pas forcément des entités qui se comportent comme des fonctions mathématiques.[6] Par exemple, nous définissons les composants logiciels et les objets comme des abstractions basées sur les procédures. En plus, les procédures sont souples parce qu'elles ne font pas d'hypothèses sur le nombre d'entrées et sorties. Une fonction a toujours une sortie. Une procédure peut avoir un nombre quelconque d'entrées et sorties, y compris zéro. Les procédures montrent toute leur puissance quand on les utilise comme briques de base (ce qui s'appelle la programmation d'ordre supérieur).

2.3.5 Les opérations de base

Le tableau 2.3 contient les opérations de base que nous utiliserons. Les types dans le tableau sont les types de la figure 2.16. La plupart de ces opérations peuvent être écrites comme des expressions en utilisant du sucre syntaxique. Par exemple, `X=A*B` est du sucre syntaxique pour `{Number.´*´ A B X}`, où `Number.´*´` est une

6. D'un point de vue théorique, une procédure est un « processus » dans un calcul concurrent comme le π calcul. Les arguments de la procédure sont des canaux. Dans ce chapitre nous utilisons des processus qui sont composés séquentiellement avec des canaux à envoi unique. Le chapitre 4 fait de la composition concurrente des processus.

procédure associée au type `Number`.[7] Toutes les opérations peuvent être écrites en plus long, comme, `Value.´==´`, `Value.´<´`, `Int.´div´` et `Float.´/´`. Le tableau utilise le sucre syntaxique quand il existe.

Opération	Description	Type des arguments
`A==B`	Comparaison d'égalité	Value
`A\=B`	Comparaison d'inégalité	Value
`{IsProcedure P}`	Test si procédure	Value
`A=<B`	Comp. plus petit ou égal	Number ou Atom
`A<B`	Comparaison plus petit	Number ou Atom
`A>=B`	Comp. plus grand ou égal	Number ou Atom
`A>B`	Comparaison plus grand	Number ou Atom
`A+B`	Addition	Number
`A-B`	Soustraction	Number
`A*B`	Multiplication	Number
`A` **`div`** `B`	Division (entier)	Int
`A` **`mod`** `B`	Modulo	Int
`A/B`	Division (flottant)	Float
`{Arity R}`	Arité	Record
`{Label R}`	Étiquette	Record
`R.F`	Sélection de champ	Record

Tableau 2.3 Quelques opérations de base.

– *L'arithmétique*. Les nombres à virgule flottante ont les quatre opérations de base `+`, `-`, `*` et `/`, avec les définitions standard. Les entiers ont les opérations de base `+`, `-`, `*`, **`div`** et **`mod`**, où **`div`** est la division entière (on laisse tomber la fraction) et **`mod`** est le modulo entier (le reste après une division entière). Par exemple, `10` **`mod`** `3=1`.

– *Les opérations sur les enregistrements*. Les trois opérations de base sur les enregistrements sont `Arity`, `Label` et « `.` » (point, qui signifie sélection de champ). Par exemple, avec

```
X=person(name:"George" age:25)
```

alors `{Arity X}=[age name]`, `{Label X}=person` et `X.age=25`. L'appel à `Arity` renvoie une liste qui contient d'abord les traits entiers en ordre croissant, puis les traits atomiques en ordre lexicographique croissant. Cette liste représente l'arité de l'enregistrement. L'arité d'un enregistrement est l'ensemble des traits de l'enregistrement.

7. Pour être précis, `Number` est un module qui regroupe les opérations du type `Number` et `Number.´*´` sélectionne la procédure de multiplication.

– *Les comparaisons*. Les comparaisons == et \= peuvent comparer toutes les valeurs. Les comparaisons numériques =<, <, >= et > peuvent comparer des entiers, des flottants ou des atomes. Les atomes sont comparés selon l'ordre lexicographique de leurs représentations imprimées. Dans l'exemple suivant, `Z` est liée au maximum de `X` et `Y` :

```
declare X Y Z T in
X=5 Y=10
T=(X>=Y)
if T then Z=X else Z=Y end
```

Il y a un sucre syntaxique pour que l'instruction **if** accepte une expression comme condition. Une autre manière d'écrire cet exemple est donc :

```
declare X Y Z in
X=5 Y=10
if X>=Y then Z=X else Z=Y end
```

– *Les opérations sur les procédures*. Il y a trois opérations de base sur les procédures : la création (avec l'instruction **proc**), l'appel (avec les accolades `{ ··· }`) et le test qui vérifie si une valeur est une procédure avec la fonction booléenne `IsProcedure`. L'appel `{IsProcedure P}` renverra **true** si `P` est une procédure et **false** sinon.

Pour un ensemble plus complet des opérations de base consultez la documentation de Mozart [23].

2.4 LA SÉMANTIQUE DU LANGAGE NOYAU

L'exécution du langage noyau est l'évaluation des fonctions sur les valeurs partielles. Pour voir cela, nous définissons la sémantique du langage noyau dans un modèle opérationnel simple. Le modèle est conçu pour permettre au programmeur un raisonnement simple sur l'exactitude et la complexité calculatoire. C'est une machine abstraite possédant un niveau d'abstraction élevé qui omet des détails tels que les registres et les adresses machine explicites.

2.4.1 Les concepts de base

Avant de montrer la sémantique formelle, voici quelques exemples pour renforcer l'intuition sur l'exécution du langage noyau. Ceux-ci motiveront la sémantique et faciliteront sa compréhension.

Une exécution simple

Les instructions sont exécutées une par une selon leur ordre textuel dans le programme. Nous verrons que cet ordre est réalisé dans la sémantique par une instruction appelée composition séquentielle. Voici une exécution simple :

```
local A B C D in A=11 B=2 C=A+B D=C*C end
```

Elle va lier `D` à 169. Examinons-la de près pour voir ce qu'elle fait exactement. L'instruction **`local`** crée quatre nouvelles variables en mémoire et les fait référencer par les quatre identificateurs `A`, `B`, `C` et `D`. (Pour la facilité nous étendons légèrement l'instruction **`local`** du tableau 2.1.) Ensuite il y a deux liens, `A=11` et `B=2`. L'addition `C=A+B` additionne les valeurs `A` et `B` et lie `C` au résultat 13. La multiplication `D=C*C` multiplie la valeur de `C` par elle-même et lie `D` au résultat 169.

Les identificateurs et la portée statique

Nous avons vu que l'instruction **`local`** fait deux choses : elle crée une nouvelle variable et elle fait une référence d'un identificateur vers cette variable. L'identificateur ne référence la variable qu'à l'intérieur de l'instruction **`local`**, entre le **`local`** et le **`end`**. La région du programme dans laquelle un identificateur référence une variable particulière s'appelle la **portée** de l'identificateur. En dehors de la portée, l'identificateur n'a pas la même référence. Regardons de près ce que cela implique. Considérez le fragment suivant :

```
local X in
   X=1
   local X in X=2 {Browse X} end
   {Browse X}
end
```

Qu'affiche-t-il ? Il affiche d'abord 2 et ensuite 1. Il n'y a qu'un identificateur, `X`, mais à différents moments pendant l'exécution, il référence des variables différentes.

Pour résumer, le sens d'un identificateur comme `X` est déterminé par l'instruction **`local`** la plus proche de `X` qui déclare `X`. La partie du programme où `X` garde ce sens s'appelle la portée de `X`. Nous pouvons déterminer la portée d'un identificateur par une simple inspection du texte du programme. Une action plus compliquée comme une exécution ou une analyse du programme n'est pas nécessaire. Cette règle de portée s'appelle la portée lexicale ou la portée statique. Plus loin nous verrons une autre règle de portée, la portée dynamique, qui est parfois utile. Mais la portée lexicale est de loin la règle la plus importante. Une des raisons en est qu'elle est localisée : le sens d'un identificateur peut être déterminé en regardant une petite partie du programme. Nous verrons une autre raison ci-dessous.

Les procédures

Les procédures sont un des plus importants éléments primitifs du langage. Nous donnons un exemple simple qui montre comment définir et appeler une procédure. Voici une procédure `Max` qui lie `Z` au maximum de `X` et `Y` :

```
proc {Max X Y ?Z}
   if X>=Y then Z=X else Z=Y end
end
```

Pour faciliter la lecture de cette définition, nous marquons l'argument de sortie avec un point d'interrogation « `?` ». Cette marque n'a absolument aucun effet sur l'exécution ; c'est uniquement un commentaire. L'exécution de `{Max 3 5 C}` lie `C` à 5. Comment la procédure fonctionne-t-elle, exactement ? À l'appel de `Max`, `X` correspond à 3, `Y` à 5 et `Z` à la variable non liée référencée par `C`. Quand `Max` lie `Z`, alors cette variable sera liée. Cette manière de passer les paramètres s'appelle le **passage par référence**. Nous utilisons surtout l'appel par référence, à la fois pour les variables dataflow et pour les variables affectables. La section 5.4.3 explique d'autres mécanismes de passage de paramètres.

Les procédures avec des références externes

Examinons le corps de `Max`. C'est une instruction **`if`** :

```
if X>=Y then Z=X else Z=Y end
```

Cette instruction ne peut pas être exécutée car elle ne définit pas les identificateurs `X`, `Y` et `Z` ! Ces identificateurs non définis s'appellent des **identificateurs libres**. (Parfois on les appelle des variables libres, mais strictement parlant ils ne sont pas des variables.) Si on met l'instruction **`if`** à l'intérieur de la procédure `Max`, elle pourra être exécutée parce que tous les identificateurs libres seront déclarés comme arguments de la procédure.

Que se passe-t-il si on définit une procédure qui déclare seulement une partie des identificateurs libres comme arguments ? Par exemple, définissons la procédure `LB` avec le même corps que `Max`, mais seulement deux arguments :

```
proc {LB X ?Z}
   if X>=Y then Z=X else Z=Y end
end
```

Que fait cette procédure ? Apparemment, elle prend un nombre `X` et lie `Z` à `X` si `X>=Y`, et à `Y` sinon. En fait, `Z` aura toujours une valeur au moins `Y`. Mais quelle est la valeur de `Y` ? Elle n'est pas un argument de la procédure. Il faut regarder la valeur de `Y` au moment où la procédure est définie. Ce comportement est une conséquence de la portée statique. Si `Y=9` quand on définit la procédure, l'exécution de `{LB 3 Z}` liera `Z` à 9. Regardons le fragment suivant :

```
local Y LB in
   Y=10
   proc {LB X ?Z}
      if X>=Y then Z=X else Z=Y end
   end
   local Y=15 Z in
      {LB 5 Z}
   end
end
```

Quand on fait l'appel `{LB 5 Z}`, à quoi `Z` sera-t-elle liée ? Elle sera liée à 10. Le lien `Y=15` à l'appel de `LB` est ignoré : c'est le lien `Y=10` visible dans la définition qui est important.

La portée dynamique versus la portée statique

Regardons le fragment suivant :

```
local P Q in
   proc {Q X} {Browse stat(X)} end
   proc {P X} {Q X} end
   local Q in
      proc {Q X} {Browse dyn(X)} end
      {P hello}
   end
end
```

Est-ce qu'il affiche `stat(hello)` ou `dyn(hello)` ? La portée statique dit qu'il affiche `stat(hello)`. En d'autres termes, `P` utilise la version de `Q` qui existe à la définition de `P`. Mais il y a une autre possibilité : `P` pourrait utiliser la version de `Q` qui existe à l'appel de `P`. Cette possibilité s'appelle la portée dynamique.

Les deux formes, la portée statique et la portée dynamique, ont été utilisées comme le défaut dans les langages de programmation. Pour les comparer, mettons d'abord leurs définitions côte à côte :

- **La portée statique**. La variable qui correspond à une occurrence d'un identificateur est celle définie dans la déclaration qui contient l'occurrence et qui est la plus proche de l'occurrence dans le texte du programme.
- **La portée dynamique**. La variable qui correspond à une occurrence d'un identificateur est celle définie dans la déclaration qui contient l'occurrence et qui est la plus récente pendant l'exécution qui mène jusqu'à l'instruction qui contient l'occurrence.

À l'origine, le langage Lisp avait une portée dynamique. Les langages Common Lisp et Scheme, qui sont des descendants de Lisp, ont une portée statique par défaut. Common

Lisp permet la déclaration des variables à portée dynamique, qu'il appelle variables spéciales [87]. Quel est le meilleur défaut pour la portée ? Le meilleur défaut pour une valeur procédurale est la portée statique car une procédure qui fonctionne bien à sa définition continuera à bien fonctionner indépendamment de l'environnement dans lequel elle est appelée. C'est une propriété importante en génie logiciel.

La portée dynamique reste utile dans certains domaines. Par exemple, le cas d'une procédure dont le code est transféré d'un ordinateur à un autre via le réseau. Certaines de ses références externes, comme les appels à des opérations standard de bibliothèque, peuvent utiliser la portée dynamique. De cette façon, la procédure utilisera les définitions locales pour ces opérations au lieu des définitions à distance. C'est bien plus efficace.[8]

L'abstraction procédurale

Récapitulons ce que nous avons appris de `Max` et `LB`. Il y a trois concepts :

1. L'abstraction procédurale. Toute instruction peut devenir une procédure simplement en la mettant dans une déclaration de procédure. Cela s'appelle l'**abstraction procédurale**. En d'autres termes, nous utilisons l'abstraction procédurale pour obtenir une procédure.
2. Les identificateurs libres. Un identificateur libre dans une instruction est un identificateur qui n'est pas défini dans cette instruction. Il peut être défini dans une autre instruction qui l'enveloppe.
3. La portée statique. Une procédure peut avoir des références externes. Ce sont des identificateurs libres dans le corps de la procédure qui ne sont pas des arguments. `LB` a une référence externe. `Max` n'en a pas. La valeur d'une référence externe est sa valeur lors de la définition de la procédure. C'est une conséquence de la portée statique.

Ensemble, ces trois concepts forment un des outils les plus puissants présenté dans ce livre. Dans la sémantique, nous verrons que leur implémentation est simple.

Le comportement dataflow

Dans la mémoire à affectation unique, les variables peuvent être non liées. Mais certaines instructions ont besoin de variables liées pour s'exécuter. Par exemple, que se passe-t-il quand on exécute :

8. Pourtant, il n'y a pas de garantie que l'opération se comportera de la même manière sur l'ordinateur cible. Alors le défaut devrait être la portée statique même pour les programmes répartis.

```
local X Y Z in
   X=10
   if X>=Y then Z=X else Z=Y end
end
```

La comparaison `X>=Y` renvoie **`true`** ou **`false`** si elle peut décider de la relation entre `X` et `Y`. Si `Y` est non liée, elle ne pourra pas décider. Que fait-elle ? Continuer avec **`true`** ou **`false`** serait erroné. Lever une erreur serait une mesure trop forte, parce que le programme n'a pas fait quelque chose d'incorrect (il n'a pas fait quelque chose de correct non plus). Le programme arrêtera tout simplement son exécution sans signaler d'erreur. Si une autre activité (à déterminer ultérieurement) lie `Y`, alors l'exécution arrêtée continuera comme si rien n'avait perturbé son déroulement normal. Cela s'appelle un comportement dataflow. Le comportement dataflow est à la base d'un deuxième outil puissant, à savoir la concurrence. Dans la sémantique nous verrons que le comportement dataflow a aussi une implémentation simple.

2.4.2 La machine abstraite

Nous définissons la sémantique du langage noyau comme une sémantique opérationnelle : elle définit le sens du langage noyau à travers son exécution sur une machine abstraite. Nous définissons d'abord les concepts de base de cette machine : l'environnement, l'instruction sémantique, la pile sémantique, l'état d'exécution et le calcul. Puis nous montrons comment exécuter un programme. Enfin, nous expliquons comment faire des calculs avec les environnements, ce qui est une opération fréquente dans la sémantique.

La définition des concepts de base

Un programme en exécution est défini par un calcul, qui est une séquence d'états d'exécution. Voici une définition précise de ce que cela veut dire. Nous avons besoin des concepts suivants :

- Une **mémoire à affectation unique** σ est un ensemble de variables. Ces variables sont partitionnées en deux groupes : (1) les ensembles de variables qui sont égales mais non liées et (2) les variables qui sont liées à un nombre, un enregistrement ou une procédure. Par exemple, dans la mémoire $\{x_1, x_2 = x_3, x_4 = \mathtt{a}\,|\,x_2\}$, x_1 est non liée, x_2 et x_3 sont égales et non liées et x_4 est liée à la valeur partielle $\mathtt{a}\,|\,x_2$. Une variable en mémoire qui est liée à une valeur est identifiée à cette valeur. C'est pourquoi une variable en mémoire est parfois appelée une entité en mémoire.
- Un **environnement** E est une correspondance des identificateurs vers des entités dans σ. Cela est expliqué dans la section 2.2. Nous écrivons E comme un ensemble de paires, par exemple $\{\mathtt{X} \rightarrow x, \mathtt{Y} \rightarrow y\}$, où `X`, `Y` sont des identificateurs et x, y référencent des entités en mémoire.

– Une **instruction sémantique** est une paire $(\langle s \rangle, E)$ où $\langle s \rangle$ est une instruction et E est un environnement. L'instruction sémantique montre la relation entre une instruction et ce qu'elle référence en mémoire. Les instructions possibles sont énumérées dans la section 2.3.
– Un **état d'exécution** est une paire (ST, σ) où ST est une pile d'instructions sémantiques et σ est une mémoire à affectation unique. La figure 2.17 montre un état d'exécution.
– Un **calcul** est une séquence d'états d'exécution qui commence par un état initial : $(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow \cdots$.

Chaque transition dans un calcul s'appelle un **pas d'exécution**. Il est atomique : aucun état intermédiaire n'est visible. C'est comme si le pas était fait « d'un seul coup ». Dans ce chapitre, tous les calculs sont séquentiels, c'est-à-dire que l'état d'exécution contient une seule pile d'instructions qui est transformée successivement par une séquence de pas d'exécution.

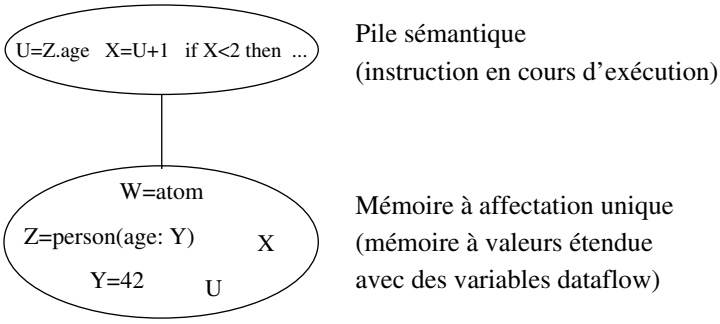


Figure 2.17 Le modèle de calcul déclaratif.

L'exécution d'un programme

Nous exécutons un programme avec cette sémantique. Un programme est une instruction $\langle s \rangle$ sans références externes. Voici comment l'exécuter :

– L'état d'exécution initial est :

$$([(\langle s \rangle, \phi)], \phi)$$

La mémoire initiale est vide (pas de variables, σ est l'ensemble vide ϕ) et il y a une seule instruction sémantique $(\langle s \rangle, \phi)$ sur la pile ST. L'instruction sémantique contient $\langle s \rangle$ et un environnement vide (ϕ). Nous utilisons des crochets [...] pour représenter la pile.
– À chaque pas, le premier élément de ST est dépilé et l'exécution continue selon sa forme.

– L'état final d'exécution (s'il y en a un) est un état dans lequel la pile sémantique est vide.

Une pile sémantique *ST* a trois états possibles pendant l'exécution :

– Exécutable : *ST* peut faire un pas d'exécution.
– Terminée : *ST* est vide.
– Suspendue : *ST* n'est pas vide, mais elle ne peut faire aucun pas d'exécution.

Calculer avec les environnements

L'exécution d'un programme fait souvent des calculs avec des environnements. Un environnement E est une fonction qui prend un identificateur $\langle \mathsf{x} \rangle$ et qui renvoie une entité en mémoire (une variable non liée ou une valeur). La notation $E(\langle \mathsf{x} \rangle)$ désigne l'entité associée avec l'identificateur $\langle \mathsf{x} \rangle$. Pour définir la sémantique des instructions de la machine abstraite, nous avons besoin de deux opérations sur les environnements, l'adjonction et la restriction.

L'**adjonction** définit un nouvel environnement en ajoutant une paire à un environnement existant. La notation

$$E + \{\langle \mathsf{x} \rangle \rightarrow x\}$$

désigne un nouvel environnement E' construit en prenant E et en ajoutant la paire $\{\langle \mathsf{x} \rangle \rightarrow x\}$. Cette paire aura la priorité si E contient une autre paire avec le même identificateur $\langle \mathsf{x} \rangle$. C'est-à-dire, $E'(\langle \mathsf{x} \rangle)$ est égal à x, et $E'(\langle \mathsf{y} \rangle)$ est égal à $E(\langle \mathsf{y} \rangle)$ pour tous les identificateurs $\langle \mathsf{y} \rangle$ différents de $\langle \mathsf{x} \rangle$. Pour ajouter plusieurs paires en même temps, nous écrivons $E + \{\langle \mathsf{x} \rangle_1 \rightarrow x_1, \ldots, \langle \mathsf{x} \rangle_n \rightarrow x_n\}$.

La **restriction** définit un nouvel environnement dont le domaine est un sous-ensemble du domaine d'un environnement existant. La notation

$$E|_{\{\langle \mathsf{x} \rangle_1, \ldots, \langle \mathsf{x} \rangle_n\}}$$

désigne un nouvel environnement E' tel que $\mathrm{dom}(E') = \mathrm{dom}(E) \cap \{\langle \mathsf{x} \rangle_1, \ldots, \langle \mathsf{x} \rangle_n\}$ et $E'(\langle \mathsf{x} \rangle) = E(\langle \mathsf{x} \rangle)$ pour tous $\langle \mathsf{x} \rangle \in \mathrm{dom}(E')$. L'environnement E' contient seulement des identificateurs qui font partie de $\mathrm{dom}(E')$.

2.4.3 Les instructions qui ne suspendent pas

Nous donnons d'abord la sémantique des instructions qui ne suspendent jamais.

L'instruction **skip**

L'instruction sémantique est :

(**skip**, E)

L'exécution dépile simplement l'instruction. L'exécution est alors complète.

La composition séquentielle

L'instruction sémantique est :

($\langle \mathsf{s} \rangle_1$ $\langle \mathsf{s} \rangle_2$, E)

L'exécution fait les actions suivantes (dans cet ordre) :

- Empiler ($\langle \mathsf{s} \rangle_2$, E).
- Empiler ($\langle \mathsf{s} \rangle_1$, E).

L'instruction $\langle \mathsf{s} \rangle_1$, qui est au sommet de la pile, sera donc exécutée avant $\langle \mathsf{s} \rangle_2$. La composition séquentielle permet donc d'exécuter les instructions selon leur ordre dans le programme.

La déclaration de variable (l'instruction **local***)*

L'instruction sémantique est :

(**local** $\langle \mathsf{x} \rangle$ **in** $\langle \mathsf{s} \rangle$ **end**, E)

L'exécution fait les actions suivantes :

- Créer une nouvelle variable x en mémoire.
- Calculer $E' = E + \{\langle \mathsf{x} \rangle \rightarrow x\}$. E' est le même que E sauf qu'il ajoute une correspondance de $\langle \mathsf{x} \rangle$ vers x.
- Empiler ($\langle \mathsf{s} \rangle$, E').

Le lien variable-variable

L'instruction sémantique est :

($\langle \mathsf{x} \rangle_1 = \langle \mathsf{x} \rangle_2$, E)

L'exécution fait les actions suivantes :

- Lier $E(\langle \mathsf{x} \rangle_1)$ et $E(\langle \mathsf{x} \rangle_2)$ dans la mémoire.

La création de valeur

L'instruction sémantique est :

$(\langle x \rangle = \langle v \rangle, E)$

où $\langle v \rangle$ est une valeur partiellement construite qui est un enregistrement, un nombre ou une procédure. L'exécution fait les actions suivantes :

- Créer une nouvelle variable y en mémoire.
- Construiser la valeur représentée par $\langle v \rangle$ dans la mémoire et lier y à cette valeur. Tous les identificateurs dans $\langle v \rangle$ sont remplacés par leur contenu en mémoire donné par E.
- Lier $E(\langle x \rangle)$ et y en mémoire.

Nous avons vu comment construire des valeurs qui sont des enregistrements et des nombres, mais comment construire une valeur procédurale ? Pour l'expliquer, nous devons d'abord définir ce qu'est une occurrence libre d'un identificateur.

Les occurrences libres et liées des identificateurs

Une instruction $\langle s \rangle$ peut contenir beaucoup d'occurrences d'identificateurs. Pour chaque occurrence, nous nous posons la question : où cet identificateur a-t-il été déclaré ? Si la déclaration est dans une instruction (une partie de $\langle s \rangle$ ou pas) qui entoure textuellement (qui contient) l'occurrence, alors nous dirons que la déclaration obéit à la portée lexicale. Comme cette portée est déterminée par le texte du code source, nous l'appelons aussi la portée statique.

Les occurrences d'un identificateur dans une instruction peuvent être liées ou libres par rapport à cette instruction. Une occurrence `X` est liée par rapport à une instruction $\langle s \rangle$ si elle est déclarée dans $\langle s \rangle$. Il y a trois instructions qui font la déclaration des identificateurs : l'instruction **`local`**, l'instruction **`case`** et la déclaration de procédure. Une occurrence d'un identificateur qui n'est pas liée est libre. Les occurrences libres peuvent exister seulement dans les fragments de programme incomplets, c'est-à-dire dans les instructions que l'on ne peut pas exécuter. Dans un programme que l'on peut exécuter, il est toujours vrai que chaque occurrence d'identificateur est liée.

Les occurrences d'identificateurs liées et les variables liées

Il ne faut pas confondre une occurrence d'identificateur liée et une variable liée ! Une occurrence d'identificateur liée n'existe pas à l'exécution ; c'est le nom textuel d'une variable qui apparaît dans une instruction qui la déclare. Une variable liée existe à l'exécution ; c'est une variable dataflow qui est liée à une valeur partielle.

Voici un fragment de programme avec des occurrences libres et liées :

```
local Arg1 Arg2 in
   Arg1=111*111
   Arg2=999*999
   Res=Arg1+Arg2
end
```

Dans cette instruction, tous les identificateurs sont déclarés avec la portée lexicale. Toutes les occurrences des identificateurs `Arg1` et `Arg2` sont liées et l'unique occurrence de `Res` est libre. Cette instruction ne peut pas s'exécuter. Pour la rendre exécutable, elle doit faire partie d'une instruction plus grande qui déclare `Res`. Voici une instruction qui peut s'exécuter :

```
local Res in
   local Arg1 Arg2 in
      Arg1=111*111
      Arg2=999*999
      Res=Arg1+Arg2
   end
   {Browse Res}
end
```

Cette instruction peut s'exécuter parce qu'elle ne contient pas d'occurrences libres. (La seule occurrence libre, `Browse`, est définie dans l'environnement global de l'interface interactive.)

Les valeurs procédurales (fermetures)

Regardons comment construire une valeur procédurale en mémoire. Ce n'est pas aussi simple que l'on pourrait imaginer parce que les procédures peuvent avoir des références externes. Par exemple :

```
proc {LowerBound X ?Z}
   if X>=Y then Z=X else Z=Y end
end
```

Dans cet exemple, l'instruction **if** a trois identificateurs libres, `X`, `Y` et `Z`. Deux d'entre eux, `X` et `Z`, sont aussi des arguments formels de `LowerBound`. Le troisième, `Y`, n'est pas un paramètre formel. Il doit être déclaré dans l'environnement dans lequel la procédure elle-même est déclarée. La valeur procédurale de `LowerBound` doit avoir une référence pour `Y`. Sinon, nous ne pourrions pas appeler la procédure parce que `Y` serait une sorte de pointeur détaché.

Regardons ce qui se passe dans le cas général. Une expression peut créer une valeur procédurale :

proc { $ ⟨y⟩$_1$ ··· ⟨y⟩$_n$ } ⟨s⟩ **end**

L'instruction ⟨s⟩ peut contenir des identificateurs libres. Il y a deux possibilités pour chaque identificateur libre : il est un argument formel ou il ne l'est pas. Les premiers sont définis à nouveau chaque fois que la procédure est appelée. Ils font partie des paramètres formels $\{\langle y\rangle_1, \ldots, \langle y\rangle_n\}$. Les autres sont définis une fois pour toutes quand la procédure est déclarée. Nous les appelons les références externes de la procédure. Nous les écrivons $\{\langle z\rangle_1, \ldots, \langle z\rangle_k\}$. La valeur procédurale est donc une paire :

(**proc** { $ $\langle y\rangle_1 \cdots \langle y\rangle_n$ } ⟨s⟩ **end**, CE)

Cette paire est stockée en mémoire comme toute autre valeur. Ici l'environnement contextuel $CE = E|_{\{\langle z\rangle_1,\ldots,\langle z\rangle_n\}}$, où E est l'environnement quand la procédure est déclarée.

Comme elle contient un environnement et une définition de procédure, une valeur procédurale est souvent appelée une fermeture ou une fermeture à portée lexicale, car elle « ferme » (emballe) l'environnement qui existe au moment de la définition de la procédure. Cette opération est aussi appelée une capture d'environnement. Quand la procédure est appelée, l'environnement contextuel est utilisé pour construire l'environnement du corps de la procédure qui s'exécute.

2.4.4 Les instructions à suspension

Trois instructions du langage noyau restent à définir :

⟨s⟩ ::= ...
| **if** ⟨x⟩ **then** $\langle s\rangle_1$ **else** $\langle s\rangle_2$ **end**
| **case** ⟨x⟩ **of** ⟨pattern⟩ **then** $\langle s\rangle_1$ **else** $\langle s\rangle_2$ **end**
| ´{´ ⟨x⟩ $\langle y\rangle_1 \cdots \langle y\rangle_n$ ´}´

Que font ces instructions quand ⟨x⟩ est non liée ? D'après la section 2.2.8, nous savons ce qui doit arriver. Les instructions doivent simplement attendre jusqu'à ce que ⟨x⟩ soit liée. Nous disons qu'elles sont des instructions à suspension. Elles ont une **condition d'activation** que nous définissons comme une condition qui doit être vraie pour que l'exécution continue. La condition est que $E(\langle x\rangle)$ doit être déterminée, c'est-à-dire liée à un nombre, un enregistrement ou une procédure.

Dans le modèle déclaratif de ce chapitre, une fois qu'une instruction suspend elle ne continue jamais. Le programme arrête simplement son exécution car il n'y a pas d'autre exécution qui pourrait rendre vraie la condition d'activation. Dans le chapitre 4, où nous introduirons la programmation concurrente, nous aurons des exécutions à plusieurs piles sémantiques. Une pile suspendue *ST* pourra devenir exécutable de nouveau si une autre pile fait une opération qui rend vraie la condition d'activation de *ST*. C'est la base de l'exécution dataflow. Pour l'instant, nous explorons la programmation séquentielle, qui ne contient qu'une seule pile sémantique.

L'instruction conditionnelle (l'instruction **if***)*

L'instruction sémantique est :

(**if** ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**, E)

L'exécution fait les actions suivantes :

- Si la condition d'activation est vraie ($E(\langle x \rangle)$ est déterminée), faire les actions suivantes :
  - Si $E(\langle x \rangle)$ n'est pas un booléen (**true** ou **false**), lever une condition d'erreur.
  - Si $E(\langle x \rangle)$ est **true**, empiler $(\langle s \rangle_1, E)$.
  - Si $E(\langle x \rangle)$ est **false**, empiler $(\langle s \rangle_2, E)$.
- Si la condition d'activation est fausse, ne pas continuer l'exécution. L'état d'exécution reste inchangé. Nous disons que l'exécution suspend. L'arrêt peut être temporaire. Si une autre activité dans le système rend vraie la condition d'activation, l'exécution pourra reprendre.

L'appel de procédure

L'instruction sémantique est :

$(\{\langle x \rangle\ \langle y \rangle_1 \cdots\ \langle y \rangle_n\}, E)$

L'exécution fait les actions suivantes :

- Si la condition d'activation est vraie ($E(\langle x \rangle)$ est déterminée), faire les actions suivantes :
  - Si $E(\langle x \rangle)$ n'est pas une valeur procédurale ou est une procédure avec un nombre d'arguments différent de n, lever une condition d'erreur.
  - Si $E(\langle x \rangle)$ a la forme (**proc** { $ ⟨z⟩$_1$ ··· ⟨z⟩$_n$ } ⟨s⟩ **end**, CE), empiler $(\langle s \rangle, CE + \{\langle z \rangle_1 \rightarrow E(\langle y \rangle_1), \ldots, \langle z \rangle_n \rightarrow E(\langle y \rangle_n)\})$.
- Si la condition d'activation est fausse, suspendre l'exécution.

Dans cette sémantique, les arguments formels sont ⟨z⟩$_1$, ..., ⟨z⟩$_n$ et les arguments effectifs sont ⟨y⟩$_1$, ..., ⟨y⟩$_n$.

La correspondance de formes (l'instruction **case***)*

L'instruction sémantique est :

(**case** ⟨x⟩ **of** ⟨lit⟩(⟨feat⟩$_1$: ⟨x⟩$_1$ ··· ⟨feat⟩$_n$: ⟨x⟩$_n$) **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**, E)

(Ici ⟨lit⟩ et ⟨feat⟩ sont des synonymes pour ⟨literal⟩ et ⟨feature⟩.) L'exécution fait les actions suivantes :

– Si la condition d'activation est vraie ($E(\langle x \rangle)$ est déterminée), faire les actions suivantes :
  - Si l'étiquette de $E(\langle x \rangle)$ est ⟨lit⟩ et son arité est $\{\langle feat \rangle_1, \ldots, \langle feat \rangle_n\}$, empiler $(\langle s \rangle_1, E + \{\langle x \rangle_1 \rightarrow E(\langle x \rangle).\langle feat \rangle_1, \ldots, \langle x \rangle_n \rightarrow E(\langle x \rangle).\langle feat \rangle_n\})$.
  - Sinon empiler $(\langle s \rangle_2, E)$.
– Si la condition d'activation est fausse, suspendre l'exécution.

2.4.5 Les concepts de base revisités

Maintenant que nous avons défini la sémantique noyau, nous pouvons jeter un deuxième regard sur les exemples de la section 2.4.1 pour voir exactement ce qu'ils font. Nous regarderons trois exemples ; nous vous recommandons les autres comme exercices.

Les identificateurs et la portée statique

Nous avons vu que l'instruction suivante ⟨s⟩ affiche d'abord 2 et ensuite 1 :

$$\langle s \rangle \equiv \left\{ \begin{array}{l} \texttt{local X in} \\ \quad \texttt{X=1} \\ \quad \langle s \rangle_1 \equiv \left\{ \begin{array}{l} \texttt{local X in} \\ \quad \texttt{X=2} \\ \quad \texttt{\{Browse X\}} \\ \texttt{end} \end{array} \right. \\ \quad \langle s \rangle_2 \equiv \texttt{\{Browse X\}} \\ \texttt{end} \end{array} \right.$$

Le même identificateur X fait d'abord référence à 2 et ensuite il fait référence à 1. Nous pouvons comprendre ce qui se passe en suivant l'exécution de ⟨s⟩ dans la machine abstraite.

1. L'état initial est :

$$([(\langle s \rangle, \phi)],\ \phi)$$

   L'environnement et la mémoire sont tous les deux vides ($E = \phi$ et $\sigma = \phi$).

2. Après l'exécution de la première instruction **local** et le lien X=1, nous avons :

$$\begin{array}{l} ([(\langle s \rangle_1 \langle s \rangle_2, \{X \rightarrow x\})], \\ \ \{y = 1, x = y\}) \end{array}$$

Nous simplifions $\{y = 1, x = y\}$ pour obtenir $\{x = 1\}$. L'identificateur `X` référence la variable x, qui est liée à 1. L'instruction suivante est la composition séquentielle $\langle \mathsf{s} \rangle_1 \langle \mathsf{s} \rangle_2$.

3. Après l'exécution de la composition séquentielle, nous avons :

$$\begin{array}{l}(\, [(\langle \mathsf{s} \rangle_1, \{\texttt{X} \rightarrow x\}), \, (\langle \mathsf{s} \rangle_2, \{\texttt{X} \rightarrow x\})], \\ \quad \{x = 1\} \,)\end{array}$$

Chaque instruction $\langle \mathsf{s} \rangle_1$ et $\langle \mathsf{s} \rangle_2$ a son propre environnement. Les deux environnements ont la même valeur.

4. Nous commençons maintenant l'exécution de $\langle \mathsf{s} \rangle_1$. La première instruction dans $\langle \mathsf{s} \rangle_1$ est une instruction **`local`**. Son exécution donne

$$\begin{array}{l}(\, [(\texttt{X=2 \ \{Browse X\}}, \{\texttt{X} \rightarrow x'\}), \, (\langle \mathsf{s} \rangle_2, \{\texttt{X} \rightarrow x\})], \\ \quad \{x', x = 1\} \,)\end{array}$$

Cela crée la nouvelle variable x' et calcule le nouvel environnement $\{\texttt{X} \rightarrow x\} + \{\texttt{X} \rightarrow x'\}$, qui donne $\{\texttt{X} \rightarrow x'\}$. La nouvelle correspondance de `X` supplante la première.

5. Après le lien `X=2` nous obtenons :

$$\begin{array}{l}(\, [(\texttt{\{Browse X\}}, \{\texttt{X} \rightarrow x'\}), \, (\texttt{\{Browse X\}}, \{\texttt{X} \rightarrow x\})], \\ \quad \{x' = 2, x = 1\} \,)\end{array}$$

(Souvenez-vous que $\langle \mathsf{s} \rangle_2$ est un `Browse`.) Maintenant la raison pour laquelle les deux appels à `Browse` affichent des valeurs différentes est claire : c'est parce qu'ils ont des environnements différents. L'instruction **`local`** de l'intérieur a son propre environnement, dans lequel `X` référence une autre variable. L'instruction **`local`** de l'extérieur garde son environnement indépendamment de ce qui se passe ailleurs.

La définition d'une procédure et son appel

L'exemple suivant définit et appelle la procédure `Max` qui calcule le maximum de deux nombres. Avec la sémantique nous pouvons voir exactement ce qui se passe pendant la définition et l'exécution de `Max`. Voici le code de l'exemple avec quelques raccourcis syntaxiques :

```
local Max C in
   proc {Max X Y ?Z}
      if X>=Y then Z=X else Z=Y end
   end
   {Max 3 5 C}
end
```

Voici une traduction en langage noyau en regroupant les **`local`** pour la clarté :

$$\langle \mathsf{s} \rangle \equiv \left\{ \begin{array}{l} \texttt{local Max in} \\ \quad \texttt{local A in} \\ \quad\quad \texttt{local B in} \\ \quad\quad\quad \texttt{local C in} \\ \quad\quad\quad \langle \mathsf{s} \rangle_1 \equiv \left\{ \begin{array}{l} \texttt{Max=proc \{\$ X Y Z\}} \\ \quad \langle \mathsf{s} \rangle_3 \equiv \left\{ \begin{array}{l} \texttt{local T in} \\ \quad \texttt{T=(X>=Y)} \\ \quad \langle \mathsf{s} \rangle_4 \equiv \texttt{if T then Z=X else Z=Y end} \\ \texttt{end} \end{array} \right. \\ \texttt{end} \\ \texttt{A=3} \\ \texttt{B=5} \\ \langle \mathsf{s} \rangle_2 \equiv \texttt{\{Max A B C\}} \end{array} \right. \\ \quad\quad\quad \texttt{end} \\ \quad\quad \texttt{end} \\ \quad \texttt{end} \\ \texttt{end} \end{array} \right.$$

Vous remarquez que la syntaxe du langage noyau est assez encombrante à cause de sa simplicité. La simplicité du langage noyau est importante parce qu'elle nous permet d'avoir une sémantique simple. Le code original a trois raccourcis pour la lisibilité :

- La déclaration de plusieurs variables dans une instruction **`local`**.
- L'utilisation des valeurs directement dans les expressions au lieu des identificateurs, comme `{P 3}` qui est un raccourci pour **`local`** `X` **`in`** `X=3 {P X}` **`end`**.
- L'utilisation d'opérations imbriquées, comme de remplacer le booléen de l'instruction **`if`** par l'opération `X>=Y`.

À partir de maintenant nous utiliserons ces trois raccourcis dans tous les exemples.

Maintenant on peut exécuter l'instruction $\langle \mathsf{s} \rangle$. Pour la clarté, nous omettons quelques pas intermédiaires.

1. L'état initial est :

$$(\, [(\langle \mathsf{s} \rangle, \phi)], \ \phi \,)$$

   L'environnement et la mémoire sont tous les deux vides ($E = \phi$ et $\sigma = \phi$).

2. Après l'exécution des quatre déclarations **`local`**, nous avons :

$$\begin{array}{l} (\, [(\langle \mathsf{s} \rangle_1, \{\texttt{Max} \rightarrow m, \texttt{A} \rightarrow a, \texttt{B} \rightarrow b, \texttt{C} \rightarrow c\})], \\ \{m, a, b, c\} \,) \end{array}$$

La mémoire contient les quatre variables m, a, b et c. L'environnement de $\langle \mathtt{s} \rangle_1$ a des correspondances pour chacune de ses variables.

3. Après l'exécution des liens de `Max`, `A` et `B`, nous obtenons :

$$([(\{\texttt{Max A B C}\}, \{\texttt{Max} \rightarrow m, \texttt{A} \rightarrow a, \texttt{B} \rightarrow b, \texttt{C} \rightarrow c\})], \\ \{m = (\textbf{proc } \texttt{\{\$ X Y Z\}} \langle \mathtt{s} \rangle_3 \textbf{ end}, \phi), a = 3, b = 5, c\})$$

Les variables m, a et b sont maintenant liées aux valeurs. La procédure est prête à être appelée. Remarquez que l'environnement contextuel de `Max` est vide parce qu'elle n'a pas d'identificateurs libres.

4. Après l'exécution de l'appel de procédure, nous obtenons :

$$([(\langle \mathtt{s} \rangle_3, \{\texttt{X} \rightarrow a, \texttt{Y} \rightarrow b, \texttt{Z} \rightarrow c\})], \\ \{m = (\textbf{proc } \texttt{\{\$ X Y Z\}} \langle \mathtt{s} \rangle_3 \textbf{ end}, \phi), a = 3, b = 5, c\})$$

L'environnement de $\langle \mathtt{s} \rangle_3$ a maintenant des correspondances pour les nouveaux identificateurs `X`, `Y` et `Z`.

5. Après l'exécution de la comparaison `X>=Y`, nous obtenons :

$$([(\langle \mathtt{s} \rangle_4, \{\texttt{X} \rightarrow a, \texttt{Y} \rightarrow b, \texttt{Z} \rightarrow c, \texttt{T} \rightarrow t\})], \\ \{m = (\textbf{proc } \texttt{\{\$ X Y Z\}} \langle \mathtt{s} \rangle_3 \textbf{ end}, \phi), a = 3, b = 5, c, t = \textbf{false}\})$$

Il y a le nouvel identificateur `T` et sa variable t liée à **`false`**.

6. L'exécution est terminée après l'instruction $\langle \mathtt{s} \rangle_4$ (l'instruction conditionnelle) :

$$([], \{m = (\textbf{proc } \texttt{\{\$ X Y Z\}} \langle \mathtt{s} \rangle_3 \textbf{ end}, \phi), a = 3, b = 5, c = 5, t = \textbf{false}\})$$

La pile des instructions est vide et c est liée à 5.

Une procédure avec des références externes (première partie)

Notre troisième exemple définit et appelle la procédure `LowerBound`, qui garantit qu'un nombre ne sera jamais plus petit qu'une borne inférieure donnée. Cet exemple est intéressant parce que `LowerBound` a une référence externe. Regardons l'exécution du code suivant :

```
local LowerBound Y C in
   Y=5
   proc {LowerBound X ?Z}
      if X>=Y then Z=X else Z=Y end
   end
   {LowerBound 3 C}
end
```

C'est très similaire à l'exemple `Max`. Le corps de `LowerBound` est identique au corps de `Max`. La seule différence est que `LowerBound` a une référence externe. La valeur procédurale est

(**proc** `{$ X Z}` **if** `X>=Y` **then** `Z=X` **else** `Z=Y` **end end**, $\{\texttt{Y} \rightarrow y\}$)

avec la mémoire suivante :

$y = 5$

Quand la procédure est définie, c'est-à-dire quand la valeur procédurale est créée, l'environnement doit contenir une correspondance pour `Y`. On fait maintenant l'appel `{LowerBound A C}` avec `A` liée à 3. Juste avant l'appel nous avons :

($[(\{\texttt{LowerBound A C}\}, \{\texttt{Y} \rightarrow y, \texttt{LowerBound} \rightarrow lb, \texttt{A} \rightarrow a, \texttt{C} \rightarrow c\})]$,
$\{ lb = ($**proc** `{$ X Z}` **if** `X>=Y` **then** `Z=X` **else** `Z=Y` **end end**,
$\{\texttt{Y} \rightarrow y\}), y = 5, a = 3, c\}$)

Après l'instruction de l'appel, quand l'exécution du corps de la procédure commence, nous avons :

($[($**if** `X>=Y` **then** `Z=X` **else** `Z=Y` **end**$, \{\texttt{Y} \rightarrow y, \texttt{X} \rightarrow a, \texttt{Z} \rightarrow c\})]$,
$\{ lb = ($**proc** `{$ X Z}` **if** `X>=Y` **then** `Z=X` **else** `Z=Y` **end end**,
$\{\texttt{Y} \rightarrow y\}), y = 5, a = 3, c\}$)

Le nouvel environnement est calculé en commençant avec l'environnement contextuel ($\{\texttt{Y} \rightarrow y\}$ dans la valeur procédurale) en ajoutant les correspondances des arguments formels `X` et `Z` aux arguments effectifs a et c.

Une procédure avec des références externes (deuxième partie)

Dans l'exemple précédent, l'identificateur `Y` référence la même variable y dans l'environnement contextuel de `LowerBound` et dans l'environnement à l'appel. Comment l'exécution changerait-elle si l'instruction suivante était exécutée au lieu de `{LowerBound 3 C}` ? :

```
local Y in
   Y=10
   {LowerBound 3 C}
end
```

`Y` ne référence plus y dans l'environnement à l'appel ! Avant de regarder la réponse, veuillez poser le livre, prendre une feuille de papier, et faire l'exécution. Juste avant l'appel nous avons presque la même situation :

([({LowerBound A C}, {Y → y', LowerBound → lb, A → a, C → c})],
{ lb = (**proc** {$ X Z} **if** X>=Y **then** Z=X **else** Z=Y **end end**,
{Y → y}), $y' = 10, y = 5, a = 3, c$})

L'environnement à l'appel a un peu changé : Y référence une nouvelle variable y', qui est liée à 10. Quand on fait l'appel, le nouvel environnement est calculé exactement comme avant, en commençant avec l'environnement contextuel et en ajoutant les arguments formels. La variable y' est ignorée ! Nous obtenons exactement la même situation qu'avant sur la pile sémantique :

([(**if** X>=Y **then** Z=X **else** Z=Y **end**, {Y → y, X → a, Z → c})],
{ lb = (**proc** {$ X Z} **if** X>=Y **then** Z=X **else** Z=Y **end end**,
{Y → y}), $y' = 10, y = 5, a = 3, c$})

La mémoire contient toujours le lien $y' = 10$. Mais comme y' n'est pas référencée par la pile sémantique, ce lien n'a aucun effet sur l'exécution.

2.5 LA GESTION DE MÉMOIRE

La machine abstraite que nous avons définie dans la section précédente est un outil puissant pour analyser les propriétés des calculs. Nous allons faire une première exploration pour regarder le comportement en mémoire, pour voir comment les tailles de la pile sémantique et la mémoire évoluent quand le calcul progresse. Nous verrons le principe de l'optimisation terminale et nous l'expliquerons au moyen de la machine abstraite. Cela nous mènera aux concepts de cycle de vie mémoire et de ramassage de miettes (« *garbage collection* »).

2.5.1 L'optimisation terminale

Regardons une procédure récursive avec un seul appel récursif qui est le dernier appel dans le corps de la procédure. Nous appelons une telle procédure récursive-terminale. Nous montrons que la machine abstraite exécute une procédure récursive-terminale avec une taille de pile constante. Cette propriété s'appelle l'**optimisation du dernier appel** ou l'**optimisation terminale** (« *last call optimization* »). Le terme de récursion terminale est parfois employé, mais il est moins précis parce que l'optimisation fonctionne pour tout dernier appel, pas seulement pour les appels récursifs (voir exercices, section 2.8). Voici la procédure que nous investiguons :

```
proc {Loop10 I}
   if I==10 then skip
   else {Browse I} {Loop10 I+1} end
end
```

L'appel `{Loop10 0}` affiche les entiers successifs de 0 jusqu'à 9. Voici l'exécution de cet appel.

– L'état initial est :

$$\begin{array}{l}([(\{\texttt{Loop10 0}\}, E_0)], \\ \sigma\ )\end{array}$$

où E_0 est l'environnement à l'appel et σ la mémoire initiale.

– Après l'exécution de l'instruction **`if`**, l'état devient :

$$\begin{array}{l}([(\{\texttt{Browse I}\}, \{\texttt{I} \rightarrow i_0\})\ (\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_0\})], \\ \{i_0 = 0\} \cup \sigma\ )\end{array}$$

– Après l'exécution du `Browse`, nous arrivons au premier appel récursif :

$$\begin{array}{l}([(\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_0\})], \\ \{i_0 = 0\} \cup \sigma\ )\end{array}$$

– Après l'exécution de l'instruction **`if`** dans l'appel récursif, nous avons :

$$\begin{array}{l}([(\{\texttt{Browse I}\}, \{\texttt{I} \rightarrow i_1\})\ (\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_1\})], \\ \{i_0 = 0, i_1 = 1\} \cup \sigma\ )\end{array}$$

– Après l'exécution du `Browse` à nouveau, nous arrivons au deuxième appel récursif :

$$\begin{array}{l}([(\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_1\})], \\ \{i_0 = 0, i_1 = 1\} \cup \sigma\ )\end{array}$$

Il est clair que la pile au k-ième appel récursif a toujours la forme :

$[(\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_{k-1}\})]$

Il n'y a qu'une instruction sémantique et son environnement a une taille constante. Cela s'appelle l'optimisation terminale. La manière efficace de programmer une boucle dans le modèle déclaratif est de la programmer comme une procédure récursive-terminale. Nous pouvons aussi voir que les tailles de la pile sémantique et la mémoire ont des évolutions différentes. La pile sémantique est bornée par une taille constante. Mais la mémoire grandit à chaque appel. Au k-ième appel récursif, la mémoire a le contenu :

$$\{i_0 = 0, i_1 = 1, \ldots, i_{k-1} = k-1\} \cup \sigma$$

La taille de la mémoire est proportionnelle au nombre d'appels récursifs. Nous verrons que cette croissance n'est pas un problème en pratique. Regardez bien la pile sémantique du k-ième appel récursif. Elle n'a pas besoin des variables $\{i_0, i_1, \ldots, i_{k-2}\}$. La seule variable dont elle a besoin est i_{k-1}. Nous pouvons donc enlever de la mémoire les variables dont nous n'avons pas besoin sans changer les résultats du calcul. La mémoire devient alors plus petite :

$$\{i_{k-1} = k - 1\} \cup \sigma$$

Cette mémoire plus petite a une taille constante. Si nous pouvions garantir que les variables dont nous n'avons pas besoin sont toujours enlevées, alors le programme pourrait s'exécuter indéfiniment avec une taille mémoire constante.

Pour cet exemple nous pouvons résoudre le problème en empilant les variables au lieu de les garder en mémoire. C'est possible parce que les variables sont liées aux petits entiers qui tiennent dans un mot du processeur (par exemple, 32 bits). La pile peut donc stocker les entiers directement au lieu de stocker des références aux variables en mémoire.[9] Cet exemple est atypique ; la plupart des programmes réalistes ont de grandes quantités de données à longue vie ou partagées qui ne peuvent pas être stockées facilement sur la pile. Nous devons donc toujours résoudre le problème général d'enlever les variables dont nous n'avons pas besoin. Dans les sections suivantes nous verrons comment le faire.

2.5.2 Le cycle de vie mémoire

Une conséquence de la sémantique de la machine abstraite est qu'un programme qui s'exécute n'a besoin que des informations de la pile sémantique et de la partie de la mémoire qui est accessible à partir de la pile sémantique. Une valeur partielle est accessible si elle est référencée par une instruction sur la pile sémantique ou par une autre valeur partielle accessible. La pile sémantique et la partie accessible de la mémoire s'appellent la **mémoire active**. Toute la mémoire qui n'est pas active peut être récupérée sans risque, c'est-à-dire qu'elle peut être réutilisée par la machine abstraite. Nous avons vu que la taille de la mémoire active de l'exemple `Loop10` est bornée par une constante. L'exemple peut donc s'exécuter indéfiniment sans épuiser la mémoire du système.

Nous pouvons introduire maintenant le concept de cycle de vie mémoire. Les programmes s'exécutent dans la mémoire centrale, qui contient une séquence de mots. En règle générale, les ordinateurs personnels ont des mots de 32 bits et les ordinateurs personnels haut de gamme en ont de 64 bits ou plus. La séquence de mots est partitionnée en blocs, où un bloc est une séquence d'un ou plusieurs mots qui est utilisé pour stocker une partie d'un état d'exécution. Le bloc est l'unité de base de l'allocation mémoire. La figure 2.18 montre le cycle de vie d'un bloc de mémoire. Chaque bloc prend successivement les trois états, actif, inactif et libre, en cycle continu. La fonction de la gestion de mémoire est d'assurer que les blocs passent correctement de chaque état au suivant. Un programme en exécution qui a besoin

9. Une optimisation supplémentaire qui est souvent faite est de stocker une partie de la pile dans les registres du processeur. Ceci est important car les registres sont beaucoup plus rapides pour la lecture et l'écriture que la mémoire centrale.

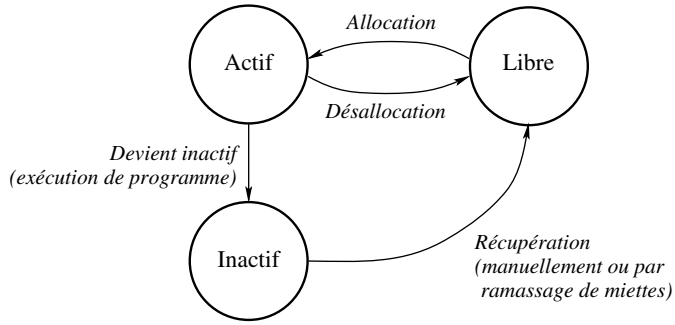


Figure 2.18 Le cycle de vie d'un bloc de mémoire.

d'un bloc l'allouera à partir d'un réservoir de blocs libres. Le bloc devient alors actif. Pendant son exécution, le programme n'a plus besoin de certains blocs alloués :

- S'il peut déterminer cela directement, il désalloue ces blocs. Les blocs deviennent alors libres immédiatement. C'est ce qui se passe avec la pile sémantique dans l'exemple `Loop10`.
- S'il ne peut pas déterminer cela directement, alors les blocs deviennent inactifs. Ils ne sont plus accessibles à partir du programme, mais comme le programme ne le sait pas, il ne peut pas libérer les blocs. C'est ce qui se passe avec la mémoire dans l'exemple `Loop10`.

Dans la plupart des cas, les blocs utilisés pour gérer le contrôle (la pile sémantique) peuvent être désalloués et les blocs utilisés pour les structures de données (la mémoire) deviennent inactifs.

La mémoire inactive doit tôt ou tard être récupérée. Le système doit reconnaître qu'elle est inactive et l'ajouter au réservoir de mémoire libre. Sinon, le système a une fuite de mémoire et n'aura finalement plus de mémoire. La récupération de la mémoire inactive est la partie la plus difficile de la gestion de mémoire, parce que reconnaître que la mémoire est inaccessible est une condition globale. Elle dépend de tout l'état d'exécution du programme. Les langages de bas niveau comme le C ou le C++ laissent souvent cette tâche au programmeur, ce qui est une source majeure d'erreurs. Il y a deux sortes d'erreurs :

- Un **pointeur détaché**. Cette erreur arrive quand un bloc est récupéré bien qu'il soit toujours accessible. Le système va tôt ou tard réutiliser le bloc. Il y aura donc des structures de données qui seront corrompues de façon imprévisible et le programme se plantera ou fera n'importe quoi. Cette erreur est particulièrement pernicieuse parce que l'effet (le système se plante) est généralement très loin

de la cause (la récupération erronée). Le débogage des pointeurs détachés est difficile.

– Une **fuite de mémoire**. Cette erreur arrive quand un bloc n'est pas récupéré bien qu'il ne soit plus accessible. La taille de la mémoire allouée augmentera continuellement jusqu'à l'épuisement de la mémoire du système. Les fuites de mémoire sont moins dangereuses que les pointeurs détachés parce que les programmes peuvent continuer à s'exécuter longtemps avant que l'erreur ne les force à s'arrêter. Les programmes à longue vie, comme les systèmes d'exploitation et les serveurs, ne doivent pas avoir de fuites de mémoire.

2.5.3 Le ramassage de miettes

Beaucoup de langages de haut niveau, comme Erlang, Haskell, Java, Lisp, Prolog, Smalltalk, Oz, etc., font de la récupération automatique. La récupération est faite par le système indépendamment du programme. Les pointeurs détachés sont complètement éliminés et les fuites de mémoire grandement réduites. Le programmeur est délesté du fardeau de la gestion manuelle de mémoire. La récupération automatique s'appelle le **ramassage de miettes** (« *garbage collection* » en anglais). C'est une technique connue qui a été utilisée pendant longtemps, depuis les années 1960 pour les premiers systèmes Lisp. Jusqu'aux années 1990, les langages courants ne l'utilisaient pas, parce qu'il était jugé (à tort) comme trop inefficace. Il est enfin devenu acceptable dans les langages courants à cause de la popularité du langage Java.

Un ramasse-miettes a typiquement deux phases. Dans la première phase, il détermine quelle partie de la mémoire est active. Il le fait en cherchant toutes les structures de données qui sont accessibles depuis un ensemble de pointeurs initiaux appelés les racines. Dans un système d'exploitation, le terme « pointeur » désigne une adresse dans l'espace mémoire d'un processus. Dans le contexte de notre machine abstraite, un pointeur est une variable en mémoire. L'ensemble des racines est l'ensemble de pointeurs dont le programme a toujours besoin. Pour la machine abstraite de la section précédente, l'ensemble des racines contient toutes les variables de la pile sémantique. En général, l'ensemble des racines contient tous les pointeurs dans les fils prêts. Nous verrons cela quand nous étendrons la machine abstraite dans les chapitres ultérieurs. L'ensemble des racines contient aussi les pointeurs à distance utilisés dans la programmation répartie.[10]

Dans la deuxième phase, le ramasse-miettes compacte la mémoire. Il rassemble tous les blocs de mémoire actifs dans un bloc contigu (un bloc sans trous) et tous les blocs de mémoire libres dans un autre bloc contigu.

10. Voir chapitre 11 de [97].

Les algorithmes modernes de ramassage de miettes sont suffisamment efficaces pour que la plupart des applications puissent les utiliser avec seulement une petite dégradation des performances en temps et en espace [47]. Pour certains systèmes, le ramasse-miettes est même plus efficace que la gestion manuelle de mémoire. Les ramasse-miettes les plus largement répandus s'exécutent dans un mode de fonctionnement par lots (« *batch* »). Ils sont inactifs la plupart du temps et ils ne s'exécutent que quand la quantité totale de mémoire active et inactive atteint un seuil prédéfini. Quand le ramasse-miettes s'exécute, il y a une pause dans l'exécution du programme. Généralement, la pause est suffisamment petite pour ne pas gêner l'utilisation du programme.

Il existe des algorithmes de ramassage de miettes, appelés ramasse-miettes temps-réels, qui peuvent s'exécuter continuellement, entrelacés avec l'exécution du programme. Ils peuvent être utilisés dans les cas pour lesquels les pauses sont inacceptables, comme le traitement en temps réel dur.

2.5.4 Le ramassage de miettes n'est pas magique

Le ramassage de miettes allège le fardeau de la gestion de mémoire pour le développeur, mais il ne l'élimine pas complètement. Il y a deux cas qui restent la responsabilité du développeur : éviter les fuites de mémoire et gérer les ressources externes.

Éviter les fuites de mémoire

Le programmeur a toujours une responsabilité par rapport aux fuites de mémoire. Si le programme continue de référencer une structure de données dont il n'a plus besoin, alors la mémoire de cette structure de données ne sera jamais récupérée. Le programme doit être attentif à éliminer toutes les références aux structures de données dont il n'a plus besoin.

Par exemple, prenons une fonction récursive qui traverse une liste. Si la tête de la liste est passée à l'appel récursif, alors la mémoire de la liste ne sera pas récupérée pendant l'exécution de la fonction. Voici un exemple :

```
L=[1 2 3 ... 1000000]
fun {Sum X L1 L}
   case L1 of Y|L2 then {Sum X+Y L2 L} else X end
end
{Browse {Sum 0 L L}}
```

`Sum` additionne les éléments d'une liste. Mais il garde aussi une référence à `L`, la liste originale, quoiqu'il n'ait pas besoin de `L`. Cela veut dire que `L` restera en mémoire pendant toute l'exécution de `Sum`. Une meilleure définition est la suivante :

```
fun {Sum X L1}
   case L1 of Y|L2 then {Sum X+Y L2} else X end
end
{Browse {Sum 0 L}}
```

Ici la référence à `L` est perdue immédiatement. Cet exemple est trivial. Mais les choses peuvent être plus subtiles. Par exemple, considérez une structure de données active `S` qui contient une liste d'autres structures de données `D1`, `D2`, ..., `Dn`. Si une de celles-là, disons `Di`, n'est plus utilisée par le programme, alors elle devra être enlevée de la liste. Sinon sa mémoire ne sera jamais récupérée.

Un programme bien écrit doit donc faire un peu de « nettoyage » de temps en temps, pour être sûr qu'il ne référence plus les structures de données dont il n'a plus besoin. Ce nettoyage est possible dans le modèle déclaratif, mais il est encombrant à faire pour le programmeur.[11]

Gérer les ressources externes

Un programme Mozart a souvent besoin de structures de données qui sont externes à son processus dans le système d'exploitation. Nous appelons une telle structure de données une ressource externe. Les ressources externes influent sur la gestion de mémoire en deux manières. Une structure de données interne à Mozart peut référencer une ressource externe et vice versa. Les deux possibilités demandent une intervention du programmeur. Considérons chaque cas séparément.

Premier cas : une structure de données Mozart référence une ressource externe. Par exemple, un enregistrement peut correspondre à une entité graphique sur un affichage graphique ou un fichier ouvert dans un système de fichiers. Si l'enregistrement n'est plus utilisé, alors l'entité graphique devra être enlevée ou le fichier devra être fermé. Sinon, l'affichage graphique ou le système de fichiers aura une fuite de mémoire. Le nettoyage est fait avec une technique qui s'appelle la **finalisation**, qui définit les actions à faire quand les structures de données disparaissent. La finalisation est hors de notre propos.[12]

Deuxième cas : une ressource externe a besoin d'une structure de données Mozart. C'est parfois facile à traiter. Par exemple, considérons un scénario où le programme Mozart implémente un serveur de base de données qui est appelé par des clients externes. Ce scénario a une solution simple : ne jamais faire de la récupération automatique de la mémoire de la base de données. D'autres scénarios ne sont pas toujours aussi simples. Une solution générale est de mettre de côté une partie du programme Mozart pour représenter la ressource externe. Cette partie doit être active (c'est-à-dire avoir son propre fil) pour qu'elle ne soit pas récupérée au hasard. Elle peut être vue

11. Il est plus simple de le faire avec l'état explicite (voir chapitre 5).
12. La finalisation est expliquée dans [97].

comme un « mandataire » (« *proxy* ») pour la ressource. Le mandataire garde une référence à la structure de données Mozart aussi longtemps que la ressource en a besoin. La ressource informe le mandataire quand elle n'a plus besoin de la structure de données.

2.5.5 Le ramasse-miettes de Mozart

Le système Mozart fait de la gestion de mémoire automatique. Il a un ramasse-miettes local et un ramasse-miettes réparti. Ce dernier est utilisé pour la programmation répartie.[13] Le ramasse-miettes local utilise un algorithme de copie à double espace. Il divise la mémoire en deux espaces égaux. À tout moment, le programme s'exécute complètement dans un des deux espaces. Le ramassage de miettes est déclenché quand il n'y a plus de mémoire libre dans cet espace. Il trouve toutes les structures de données qui sont accessibles depuis une racine et les copie vers l'autre espace. Comme la copie se fait vers un bloc contigu, cette opération fait le compactage en même temps.

L'avantage d'un ramassage de miettes à copie est que son temps d'exécution est proportionnel à la taille de la mémoire active, pas à la taille de toute la mémoire disponible. Des petits programmes feront leur ramassage rapidement, même s'ils s'exécutent dans un grand espace mémoire. Les deux inconvénients d'un ramassage de miettes à copie sont que la moitié de la mémoire est inutilisable et que les structures de données à longue vie (comme les structures du système) doivent être copiées à chaque ramassage. Nous pouvons corriger cela. La copie des données à longue vie peut être évitée en utilisant un algorithme modifié qui s'appelle un **ramassage de miettes à générations**. Cet algorithme partitionne la mémoire en générations. Les structures à longue vie sont mises dans les générations plus anciennes, qui sont ramassées moins souvent.

La division par deux de la mémoire utilisable ne sera significative que si la taille de la mémoire active approche la taille maximale qui peut être adressée par l'architecture du processeur. La technologie des ordinateurs personnels courants est actuellement en transition d'un adressage 32 bits vers un adressage 64 bits. Dans un ordinateur avec des adresses de 32 bits, la limite est typiquement atteinte quand la taille de la mémoire active est de 2048 Mo ou plus. (La limite n'est généralement pas tout à fait 2^{32} octets ou 4096 Mo, à cause des limitations du système d'exploitation.) La limite est actuellement atteinte par de grands programmes dans les ordinateurs personnels haut de gamme. Pour de tels programmes, nous recommandons un ordinateur avec des adresses à 64 bits, qui n'a pas ce problème.

13. Voir chapitre 11 de [97].

2.6 DU LANGAGE NOYAU AU LANGAGE PRATIQUE

Le langage noyau contient tous les concepts nécessaires pour la programmation déclarative, mais il est trop minimaliste pour la programmation pratique. Ses programmes sont simplement trop verbeux. La plupart de cette verbosité peut être éliminée avec l'addition judicieuse du sucre syntaxique et des abstractions linguistiques.

- La section 2.6.1 définit un ensemble de commodités syntaxiques pour obtenir une syntaxe plus succincte et lisible.
- La section 2.6.2 définit une abstraction linguistique importante, la fonction, qui est utile pour la programmation succincte et lisible.
- La section 2.6.3 associe les interfaces interactives du *Labo interactif* et du système Mozart au modèle déclaratif. Cela introduit l'instruction **`declare`**, qui est une variation de l'instruction **`local`** ciblée pour l'utilisation interactive.

Le langage résultant est utilisé dans le chapitre 3 pour l'étude des techniques de programmation du modèle déclaratif.

2.6.1 Les commodités syntaxiques

Le langage noyau définit une syntaxe simple pour toutes ses propres constructions et types. Le langage complet ajoute du sucre syntaxique pour faciliter l'usage :

- Les valeurs partielles imbriquées peuvent être écrites de manière concise.
- Les variables peuvent être déclarées et initialisées en une fois.
- Les expressions peuvent être écrites de manière concise.
- Les instructions **`if`** et **`case`** peuvent être imbriquées de manière concise.
- Les opérateurs **`andthen`** et **`orelse`** sont définis pour les instructions **`if`** imbriquées.
- Les instructions peuvent devenir des expressions avec une balise d'insertion « $ ».

Les symboles non terminaux utilisés dans la syntaxe et la sémantique noyau correspondent à la syntaxe complète selon le tableau suivant :

Syntaxe noyau	Syntaxe complète
⟨x⟩, ⟨y⟩, ⟨z⟩	⟨variable⟩
⟨s⟩	⟨statement⟩, ⟨stmt⟩

Les valeurs partielles imbriquées

Dans le tableau 2.2, la syntaxe des enregistrements et formes implique que leurs arguments sont des variables. En pratique, beaucoup de valeurs partielles sont imbriquées plus profondément. Étant donné que les valeurs imbriquées sont très fréquentes, nous

leur donnons du sucre syntaxique. Par exemple, nous étendons la syntaxe pour permettre l'écriture de `person(name:"George" age:25)` au lieu de la version plus encombrante :

```
local A B in A="George" B=25 X=person(name:A age:B) end
```

où `X` est liée à l'enregistrement imbriqué.

L'initialisation implicite des variables

Pour rendre les programmes plus courts et plus faciles à lire, on utilise du sucre syntaxique pour lier une variable tout de suite à sa déclaration. L'idée est de mettre une opération de lien entre **`local`** et **`in`**. Au lieu de **`local`** `X` **`in`** `X=10 {Browse X}` **`end`**, dans laquelle `X` est mentionnée trois fois, le raccourci permet d'écrire **`local`** `X=10` **`in`** `{Browse X}` **`end`**, qui mentionne `X` seulement deux fois. Voici un cas simple :

`local` `X=`⟨expression⟩ **`in`** ⟨statement⟩ **`end`**

Cette instruction déclare `X` et la lie au résultat de ⟨expression⟩. Le cas général est :

`local` ⟨pattern⟩`=`⟨expression⟩ **`in`** ⟨statement⟩ **`end`**

où ⟨pattern⟩ est n'importe quelle valeur partielle (toutefois avec tous les identificateurs distincts). On déclare d'abord toutes les variables dans ⟨pattern⟩ et ensuite on lie ⟨pattern⟩ au résultat de ⟨expression⟩. La règle générale dans les deux exemples est que les identificateurs à gauche de l'égalité « = », c'est-à-dire `X` ou les identificateurs dans ⟨pattern⟩, sont ceux qui sont déclarés. Les identificateurs à droite ne sont pas déclarés.

L'initialisation implicite des variables est commode pour la création d'une structure de données complexe quand nous avons besoin des variables à l'intérieur de la structure. Par exemple, si `T` n'est pas liée, alors cette instruction :

```
local tree(key:A left:B right:C value:D)=T in
   ⟨statement⟩
end
```

construira l'enregistrement `tree`, le liera à `T` et déclarera `A`, `B`, `C` et `D` en tant que parties de `T`. C'est strictement équivalent à :

```
local A B C D in
   T=tree(key:A left:B right:C value:D) ⟨statement⟩
end
```

Il est intéressant de comparer l'initialisation implicite des variables avec l'instruction **`case`**. Dans les deux cas, il y a des formes et une déclaration implicite de variables. Le premier cas les utilise pour construire des structures de données et le deuxième cas les utilise pour décomposer des structures de données.[14]

Les expressions

Une expression est un sucre syntaxique pour une séquence d'opérations qui renvoie une valeur. Elle est différente d'une instruction, qui est aussi une séquence d'opérations mais qui ne renvoie pas de valeur. Une expression peut être utilisée à l'intérieur d'une instruction à la place d'une valeur. Par exemple, `11*11` est une expression et `X=11*11` est une instruction. Une expression est définie par une traduction en langage noyau. Alors `X=11*11` est traduite en `{Mul 11 11 X}`, où `Mul` est une procédure à trois arguments qui fait la multiplication.[15]

Le tableau 2.4 montre la syntaxe des expressions qui calculent avec des nombres. Plus loin nous verrons des expressions pour calculer avec d'autres types de données. Les expressions sont construites hiérarchiquement, en commençant avec des expressions de base (identificateurs et nombres). Il y a deux manières de combiner des expressions pour en faire d'autres : avec des opérateurs (comme l'addition `1+2+3+4`) ou avec des appels de fonction (comme la racine carrée `{Sqrt 5.0}`).

⟨expression⟩	::=	⟨variable⟩ | ⟨int⟩ | ⟨float⟩
	|	⟨unaryOp⟩ ⟨expression⟩
	|	⟨expression⟩ ⟨evalBinOp⟩ ⟨expression⟩
	|	´(´ ⟨expression⟩ ´)´
	|	´{´ ⟨expression⟩ { ⟨expression⟩ } ´}´
	|	...
⟨unaryOp⟩	::=	´~´ | ...
⟨evalBinOp⟩	::=	´+´ | ´-´ | ´*´ | ´/´ | **div** | **mod**
	|	´==´ | ´\=´ | ´<´ | ´=<´ | ´>´ | ´>=´ | ...

Tableau 2.4 Les expressions pour calculer avec des nombres.

14. L'initialisation implicite des variables peut aussi décomposer des structures de données. Si `T` est déjà liée à un enregistrement `tree`, alors ses quatre champs seront liés à `A`, `B`, `C` et `D`. Cela fonctionne parce que l'opération de lien fait de l'unification, qui est symétrique (voir l'explication dans [97]). Nous ne recommandons pas cette utilisation.

15. Son vrai nom est `Number.´*´` et elle fait partie du module `Number`.

Les instructions **if** *et* **case** *imbriquées*

Nous ajoutons du sucre syntaxique pour faciliter l'écriture des instructions **if** et **case** avec plusieurs alternatives et des conditions compliquées. Le tableau 2.5 montre la syntaxe complète de l'instruction **if**. Le tableau 2.6 montre la syntaxe complète de l'instruction **case** et ses formes. (Certains non terminaux dans ces tableaux sont définis en annexe B.) Ces instructions sont traduites en instructions **if** et **case** primitives du langage noyau. Voici un exemple d'une instruction **case** complète :

```
case Xs#Ys of nil#Ys then ⟨s⟩1
[] Xs#nil then ⟨s⟩2
[] (X|Xr)#(Y|Yr) andthen X=<Y then ⟨s⟩3
else ⟨s⟩4 end
```

Elle contient une série d'alternatives séparées avec le symbole « `[]` ». Les alternatives sont souvent appelées des clauses. Voici la traduction en syntaxe noyau :

```
case Xs of nil then ⟨s⟩1 else
   case Ys of nil then ⟨s⟩2 else
      case Xs of X|Xr then
         case Ys of Y|Yr then
            if X=<Y then ⟨s⟩3 else ⟨s⟩4 end
         else ⟨s⟩4 end
      else ⟨s⟩4 end
   end
end
```

Cette traduction montre une propriété importante de l'instruction **case** complète : les clauses sont essayées en séquence en partant de la première clause. L'exécution continue au-delà d'une clause seulement si la forme de la clause est incompatible avec l'argument d'entrée.

Les formes imbriquées sont traitées en essayant d'abord la forme à l'extérieur et ensuite les formes plus à l'intérieur. La forme imbriquée `(X|Xr)#(Y|Yr)` a une forme extérieure avec la structure `A#B` et deux formes intérieures avec la structure `A|B`. Les trois formes sont des tuples écrits avec une syntaxe infixée, avec les opérateurs infixés `´#´` et `´|´`. Elles auraient pu être écrites avec la syntaxe habituelle comme `´#´(A B)` et `´|´(A B)`. Chaque forme intérieure `(X|Xr)` et `(Y|Yr)` est mise dans sa propre instruction **case** de base. La forme extérieure avec `´#´` disparaît de la traduction parce qu'elle apparaît aussi dans l'entrée du **case**. Dans cet exemple la correspondance avec `´#´` peut donc être faite lors de la compilation.

```
⟨statement⟩ : :=
        if ⟨expression⟩ then ⟨inStatement⟩
        { elseif ⟨expression⟩ then ⟨inStatement⟩ }
        [ else ⟨inStatement⟩ ] end
    |   ...
⟨inStatement⟩ : :=
        [ { ⟨declPart⟩ }+ in ] ⟨statement⟩
```

Tableau 2.5 L'instruction **`if`**.

```
⟨statement⟩ : :=
      case ⟨expression⟩
      of ⟨pattern⟩ [ andthen ⟨expression⟩ ] then ⟨inStatement⟩
      { ´ [ ] ´ ⟨pattern⟩ [ andthen ⟨expression⟩ ] then ⟨inStatement⟩ }
      [ else ⟨inStatement⟩ ] end
  |   ...
⟨pattern⟩ : :=
      ⟨variable⟩ | ⟨atom⟩ | ⟨int⟩ | ⟨float⟩
  |   ⟨string⟩ | unit | true | false
  |   ⟨label⟩ ´ ( ´ { [ ⟨feature⟩ ':' ] ⟨pattern⟩ } [ ´ . . . ´ ] ´ ) ´
  |   ⟨pattern⟩ ⟨consBinOp⟩ ⟨pattern⟩
  |   ´ [ ´ { ⟨pattern⟩ }+ ´ ] ´
⟨consBinOp⟩ : := ´ # ´ | ´ | ´
```

Tableau 2.6 L'instruction **`case`**.

Les opérateurs **`andthen`** *et* **`orelse`**

Les opérateurs **`andthen`** et **`orelse`** sont utilisés dans les calculs avec les valeurs booléennes. L'expression

⟨expression⟩$_1$ **`andthen`** ⟨expression⟩$_2$

est traduite en

`if` ⟨expression⟩$_1$ **`then`** ⟨expression⟩$_2$ **`else false end`**

L'avantage de **`andthen`** est que ⟨expression⟩$_2$ ne sera pas évaluée si ⟨expression⟩$_1$ est **`false`**. Il y a un opérateur **`orelse`** avec un comportement analogue. L'expression

⟨expression⟩$_1$ **`orelse`** ⟨expression⟩$_2$

est traduite en

if ⟨expression⟩$_1$ **then true else** ⟨expression⟩$_2$ **end**

c'est-à-dire ⟨expression⟩$_2$ n'est pas évaluée si ⟨expression⟩$_1$ est **true**.

Les balises d'insertion

La balise d'insertion « `$` » change toute instruction en une expression. La valeur de l'expression est ce qui est à la position indiquée par la balise. Par exemple, l'instruction `{P X1 X2 X3}` peut être écrite `{P X1 $ X3}`, qui est une expression avec la valeur `X2`. Le code source peut donc être plus concis, parce que l'on évite de déclarer et d'utiliser l'identificateur `X2`. La variable qui correspond à `X2` est cachée du code source.

Les balises d'insertion peuvent rendre le code source plus lisible pour un programmeur averti, quoiqu'il soit plus difficile pour un débutant de voir comment le code est traduit en langage noyau. Nous les utiliserons uniquement quand elles augmentent considérablement la lisibilité. Par exemple, au lieu d'écrire

```
local X in {Obj get(X)} {Browse X} end
```

nous écrivons `{Browse {Obj get($)}}`. Une fois que vous êtes habitué aux balises d'insertions, elles deviennent concises et claires. Notez que la syntaxe des valeurs procédurales expliquée dans la section 2.3.3 est cohérente avec la syntaxe des balises d'insertion.

2.6.2 Les fonctions (l'instruction `fun`)

Le modèle déclaratif fournit une abstraction linguistique pour programmer avec des fonctions. C'est notre premier exemple d'une abstraction linguistique comme nous l'avons définie en section 2.1.2. Nous définissons la nouvelle syntaxe des définitions et appels de fonction et montrons comment les traduire en langage noyau.

La définition d'une fonction

Une définition de fonction est différente d'une définition de procédure de deux points de vue : elle commence avec le mot clé **fun** et le corps doit terminer avec une expression. Voici un exemple d'une définition simple :

fun `{F X1 ... XN}` ⟨statement⟩ ⟨expression⟩ **end**

Cela devient la définition de procédure suivante :

proc `{F X1 ... XN ?R}` ⟨statement⟩ `R=`⟨expression⟩ **end**

L'argument supplémentaire `R` est lié à l'expression dans le corps de la procédure. Si le corps de la fonction est une instruction **`if`**, alors chaque alternative de l'instruction devra terminer par une expression :

```
fun {Max X Y}
   if X>=Y then X else Y end
end
```

Cela est traduit en :

```
proc {Max X Y ?R}
   R = if X>=Y then X else Y end
end
```

Nous pouvons continuer la traduction en transformant le **`if`** d'une expression en une instruction. Cela donne le résultat final :

```
proc {Max X Y ?R}
   if X>=Y then R=X else R=Y end
end
```

Des règles similaires s'appliquent pour les instructions **`local`** et **`case`**, et pour les autres instructions que nous verrons plus loin. Chaque instruction peut être utilisée comme une expression. Quand une composition séquentielle dans une procédure se termine par une instruction, la séquence correspondante dans une fonction se termine par une expression. Le tableau 2.7 donne la syntaxe complète des expressions en appliquant cette règle. Ce tableau montre toutes les instructions que nous avons vues jusqu'à maintenant avec leur syntaxe en tant qu'expression. En particulier, il y a également les valeurs fonctionnelles, qui sont simplement des valeurs procédurales écrites en syntaxe fonctionnelle.

L'appel de fonction

Un appel de fonction `{F X1 ... XN}` est traduit en un appel de procédure `{F X1 ... XN R}`, où `R` remplace l'appel de `F` à son emplacement. Par exemple, l'appel imbriqué de `F` suivant :

```
{Q {F X1 ... XN} ...}
```

est traduit en :

```
local R in
   {F X1 ... XN R}
   {Q R ...}
end
```

En général, les appels de fonction imbriqués sont évalués avant l'appel de la fonction dans laquelle ils sont imbriqués. S'il y en a plusieurs, ils sont évalués dans l'ordre de leur apparition dans le programme.

```
⟨statement⟩ : :=
      fun '{' ⟨variable⟩ { ⟨pattern⟩ } '}' ⟨inExpression⟩ end
   |  ...
⟨expression⟩ : :=
      fun '{' '$' { ⟨pattern⟩ } '}' ⟨inExpression⟩ end
   |  proc '{' '$' { ⟨pattern⟩ } '}' ⟨inStatement⟩ end
   |  '{' ⟨expression⟩ { ⟨expression⟩ } '}'
   |  local { ⟨declPart⟩ }+ in ⟨expression⟩ end
   |  if ⟨expression⟩ then ⟨inExpression⟩
      { elseif ⟨expression⟩ then ⟨inExpression⟩ }
      [ else ⟨inExpression⟩ ] end
   |  case ⟨expression⟩
      of ⟨pattern⟩ [ andthen ⟨expression⟩ ] then ⟨inExpression⟩
      { '[]' ⟨pattern⟩ [ andthen ⟨expression⟩ ] then ⟨inExpression⟩ }
      [ else ⟨inExpression⟩ ] end
   |  ...
⟨inStatement⟩ : :=
      [ { ⟨declPart⟩ }+ in ] ⟨statement⟩
⟨inExpression⟩ : :=
      [ { ⟨declPart⟩ }+ in ] [ ⟨statement⟩ ] ⟨expression⟩
```

Tableau 2.7 La syntaxe des fonctions.

L'appel de fonction dans une structure de données

Il y a une règle supplémentaire pour les appels de fonction. Il s'agit d'un appel dans une structure de données (enregistrement, tuple ou liste). Voici un exemple :

```
Ys={F X}|{Map Xr F}
```

Dans ce cas, la traduction met les appels imbriqués après l'opération de lien :

```
local Y Yr in
   Ys=Y|Yr
   {F X Y}
   {Map Xr F Yr}
end
```

Cela permet souvent à l'appel récursif d'être le dernier appel. La section 2.5.1 explique pourquoi c'est important pour l'efficacité de l'exécution. La fonction `Map` complète est définie comme ceci :

```
fun {Map Xs F}
   case Xs of nil then nil
   [] X|Xr then {F X}|{Map Xr F} end
end
```

`Map` applique la fonction `F` à tous les éléments d'une liste et renvoie le résultat. Voici un exemple d'appel :

```
{Browse {Map [1 2 3 4] fun {$ X} X*X end}}
```

Il affiche `[1 4 9 16]`. Voici la traduction de `Map` en langage noyau :

```
proc {Map Xs F ?Ys}
   case Xs of nil then Ys=nil
   else case Xs of X|Xr then
      local Y Yr in
         Ys=Y|Yr {F X Y} {Map Xr F Yr}
      end
   end end
end
```

La variable dataflow `Yr` est utilisée à la place du résultat dans l'appel récursif `{Map Xr F Yr}`. L'appel récursif peut alors être le dernier appel. On obtient alors la récursion terminale qui s'exécute avec la même efficacité en temps et en espace qu'une construction itérative comme une boucle **`while`**.

2.6.3 L'interface interactive (l'instruction `declare`)

Le *Labo interactif* et le système Mozart ont tous les deux une interface interactive qui permet l'introduction des fragments de programme et leur exécution immédiate. Les fragments doivent respecter la syntaxe des instructions interactives qui est donnée dans le tableau 2.8.

⟨interStmt⟩	::=	⟨statement⟩
	|	**declare** { ⟨declPart⟩ }+ [⟨interStmt⟩]
	|	**declare** { ⟨declPart⟩ }+ **in** ⟨interStmt⟩
⟨declPart⟩	::=	⟨variable⟩ | ⟨pattern⟩ ´=´ ⟨expression⟩ | ⟨statement⟩

Tableau 2.8 La syntaxe des instructions interactives.

Une instruction interactive (⟨interStmt⟩) peut être toute instruction légale ou la nouvelle instruction **`declare`**. Nous supposons que l'utilisateur donne des instructions interactives au système une par une. (Dans les exemples du livre, l'instruction

declare est souvent omise. Elle devra être ajoutée si l'exemple déclare de nouvelles variables.)

L'interface interactive a toute la fonctionnalité dont on a besoin pour le développement de logiciel. L'annexe A résume une partie de cette fonctionnalité. Pour l'instant, nous supposons que l'utilisateur sait comment introduire des instructions.

L'interface interactive a un seul environnement global. L'instruction **declare** ajoute des correspondances à cet environnement. Il s'ensuit que **declare** ne peut pas être utilisée dans une application autonome mais seulement de façon interactive. Introduire la déclaration suivante :

```
declare X Y
```

crée deux nouvelles variables en mémoire, x_1 et x_2, et ajoute des correspondances à partir de `X` et `Y`. Parce que les correspondances sont dans l'environnement global, nous disons que `X` et `Y` sont des variables globales ou variables interactives. Introduire la même déclaration une deuxième fois fera correspondre `X` et `Y` à deux nouvelles variables, x_3 et x_4. La figure 2.19 montre ce qui se passe. Les variables originales, x_1 et x_2, sont toujours en mémoire, mais elles ne sont plus référencées par `X` et `Y`. Dans la figure, `Browse` correspond à une valeur procédurale qui implémente le Browser. L'instruction **declare** ajoute de nouvelles variables et correspondances, mais laisse les variables existantes inchangées.

L'addition d'une nouvelle correspondance à un identificateur qui correspond déjà à une variable pourra rendre cette dernière inaccessible si elle n'a pas d'autres références. Si la variable fait partie d'un calcul, elle restera accessible à l'intérieur du calcul. Par exemple :

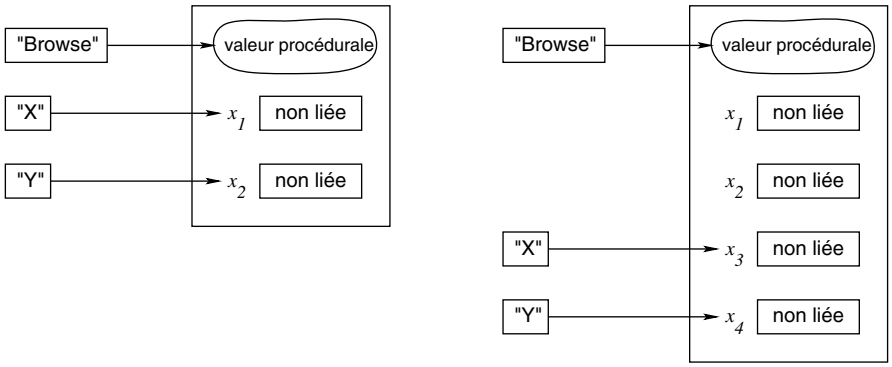
```
declare X Y
X=25
declare A
A=person(age:X)
declare X Y
```

Juste après le lien `X=25`, `X` correspond à 25. Mais après la deuxième « **declare** `X Y` », il correspond à une nouvelle variable non liée. Le 25 est toujours accessible à travers la variable globale `A` qui est liée à l'enregistrement `person(age:25)`. L'enregistrement contient 25 parce que `X` correspondait à 25 au moment du lien `A=person(age:X)`. La deuxième « **declare** `X Y` » change la correspondance de `X` mais pas l'enregistrement `person(age:25)` parce que l'enregistrement existe déjà en mémoire. Ce comportement de **declare** est conçu pour soutenir un style de programmation modulaire. L'exécution d'un fragment de programme ne changera pas les résultats des fragments qui ont été exécutés auparavant.

Il y a une variante de **declare** avec le mot clé **in**. L'exemple :

```
declare X Y in ⟨stmt⟩
```

déclare deux variables globales et exécute ensuite ⟨stmt⟩. La différence avec la première variante est que ⟨stmt⟩ ne déclare pas de variables globales (sauf si elle contient une autre **declare**).



Résultat de la première `declare X Y` Résultat de la deuxième `declare X Y`

Figure 2.19 La déclaration des variables globales.

Le Browser

Le *Labo interactif* et le système Mozart contiennent un outil qui s'appelle le **Browser** (« fureteur ») qui permet de regarder dans la mémoire. Cet outil est accessible au programmeur comme une procédure qui s'appelle `Browse`. La procédure `Browse` a un argument. Elle est appelée comme `{Browse` ⟨expr⟩`}`, où ⟨expr⟩ est une expression. Le Browser affiche des valeurs partielles. Il met à jour l'affichage automatiquement chaque fois qu'une variable non liée est liée à une valeur. Le fragment suivant :

```
{Browse 1}
```

affiche l'entier 1. Le fragment :

```
declare Y in
{Browse Y}
```

affiche uniquement le nom de la variable, donc `Y`. Aucune valeur n'est affichée. Cela signifie que `Y` est actuellement non liée. La figure 2.20 montre la fenêtre du Browser après ces deux opérations. Si `Y` est liée, par exemple en exécutant `Y=2`, alors le Browser mettra à jour son affichage pour montrer ce lien.

Figure 2.20 Le Browser.

L'exécution dataflow

Nous avons vu précédemment que les variables déclaratives soutiennent l'exécution dataflow, c'est-à-dire qu'une opération attend que tous ces arguments soient liés avant de continuer. Pour les programmes séquentiels ce n'est pas très utile car le programme attendra pour toujours. Par contre, c'est utile pour les programmes concurrents, dans lesquels il y a simultanément plusieurs séquences d'instructions en exécution. Une séquence d'instructions qui s'exécute indépendamment s'appelle un **fil**. La programmation avec plusieurs fils s'appelle la **programmation concurrente** : elle est expliquée dans le chapitre 4.

Chaque fragment de programme exécuté dans l'interface interactive s'exécute dans son propre fil. Cela nous permet de montrer des exemples simples d'exécution dataflow. Par exemple, le fragment suivant :

```
declare A B C in
C=A+B
{Browse C}
```

Le Browser n'affiche rien car l'instruction `C=A+B` bloque (les deux arguments sont non liés). Maintenant, donnez le fragment suivant :

```
A=10
```

`A` sera liée, mais l'instruction `C=A+B` bloquera toujours parce que `B` est toujours non liée. Enfin, donnez le fragment suivant :

```
B=200
```

Maintenant `210` est affiché dans le Browser. Toute opération, pas uniquement l'addition, bloquera si elle n'a pas assez d'informations pour calculer son résultat. Par exemple, les comparaisons peuvent bloquer. La comparaison d'égalité `X==Y` bloquera si elle ne peut pas décider si `X` est égal à ou différent de `Y`. Cela arrivera, par exemple, si l'une ou l'autre des variables est non liée.

Des erreurs de programmation donnent souvent lieu à des suspensions dataflow. Si vous exécutez une instruction qui devrait afficher un résultat et que rien n'est affiché, la cause probable du problème est une opération bloquée. Il faut soigneusement vérifier que les arguments de toutes les opérations sont liés. Idéalement, le débuggeur du système doit détecter quand un programme a des opérations bloquées qui ne peuvent pas continuer.

2.7 LES EXCEPTIONS

Trouvons d'abord la règle, ensuite nous tenterons d'expliquer les exceptions.
– Le Nom de la Rose, Umberto Eco (1932-)

Comment traiter une situation exceptionnelle dans un programme ? Par exemple, une division par zéro, l'ouverture d'un fichier ou la sélection d'un champ d'un enregistrement qui n'existe pas ? Comme ces situations n'arrivent pas dans un programme correct, elles ne doivent pas encombrer le style de programmation normal. Néanmoins, elles arrivent parfois. Il doit être possible pour un programme de les gérer simplement. Le modèle déclaratif ne peut pas faire cela sans l'ajout de tests encombrants partout dans le programme. Une manière plus élégante est d'étendre le modèle avec un mécanisme de traitement d'exceptions. Cette section montre comment on peut le faire. Nous donnons la syntaxe et la sémantique du modèle étendu et nous expliquons comment les exceptions apparaissent dans le langage complet.

2.7.1 La motivation et les concepts de base

Dans la sémantique de la section 2.4, nous parlons de « lever une erreur » quand une instruction ne peut pas continuer correctement. Par exemple, une instruction conditionnelle lève une erreur quand son argument est une valeur non booléenne. Nous avons été délibérément vague sur ce qui se passe ensuite. Maintenant, nous pouvons être plus précis. Nous définissons une erreur comme une différence entre le comportement d'un programme et son comportement souhaité. Il y a beaucoup de sources d'erreurs. Elles peuvent être internes ou externes au programme. Une erreur interne peut résulter de l'invocation d'une opération avec un argument de type ou de valeur illégal. Une erreur externe peut résulter de l'ouverture d'un fichier non-existant.

Nous voudrions pouvoir détecter les erreurs et les traiter à l'intérieur d'un programme pendant son exécution. Le programme ne devrait pas s'arrêter quand elles arrivent. Au contraire, il devrait transférer l'exécution vers une autre partie qui s'appelle le **gestionnaire d'exceptions**, et passer à ce gestionnaire une valeur qui décrit l'erreur.

À quoi le mécanisme de traitement d'exceptions ressemblerait-il ? Nous pouvons faire deux observations. D'abord, il doit pouvoir confiner l'erreur, la mettre en « quarantaine » pour qu'elle ne contamine pas tout le programme. Nous appelons ceci le principe d'endiguement d'erreurs. Supposons que le programme est fait des « composants » organisés de façon hiérarchique. Chaque composant est fait de composants plus petits. Nous mettons « composant » entre guillemets parce que le langage ne doit pas forcément avoir un concept de composant. Il doit seulement être compositionnel, c'est-à-dire que les programmes sont construits en couches. Le principe d'endiguement d'erreurs dit alors qu'une erreur dans un composant devrait être attrapée à la frontière du composant. En dehors du composant, l'erreur est invisible ou signalée proprement.

Le mécanisme fait alors un « saut » de l'intérieur du composant vers sa frontière. La deuxième observation est que ce saut devrait être une opération. Le mécanisme devrait pouvoir, en une opération, sortir d'un nombre quelconque de niveaux de contexte imbriqués. La figure 2.21 montre ce qui se passe. Dans notre sémantique, nous pouvons définir un contexte comme une entrée sur la pile sémantique, c'est-à-dire une instruction qui sera exécutée plus tard. Les contextes imbriqués sont créés par les appels de procédure et les compositions séquentielles.

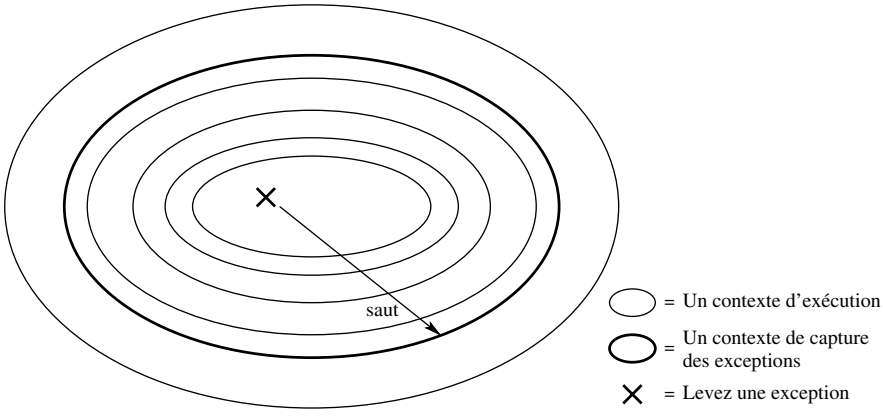


Figure 2.21 Le traitement d'exceptions.

Le modèle déclaratif ne peut pas faire le saut en une seule opération. Le saut doit être codé explicitement comme une série de petits sauts, un par contexte, avec des

variables booléennes et des instructions conditionnelles. Cela fait des programmes plus encombrants, surtout parce que le code supplémentaire doit être ajouté partout où une erreur est possible. On peut montrer formellement que la seule manière de garder la simplicité des programmes est d'étendre le modèle [51, 54].

Nous proposons une extension simple au modèle qui satisfait ces conditions. Nous ajoutons deux instructions : l'instruction **try** et l'instruction **raise**. L'instruction **try** crée un contexte de capture d'exceptions avec un traitement d'exceptions. L'instruction **raise** lève une exception : elle fait un saut jusqu'à la frontière du contexte de capture d'exceptions le plus imbriqué et elle appelle ensuite le traitement d'exceptions de ce contexte. Des instructions **try** imbriquées créent des contextes imbriqués. L'exécution de **try** ⟨s⟩ **catch** ⟨x⟩ **then** $\langle s \rangle_1$ **end** est équivalente à l'exécution de ⟨s⟩, si ⟨s⟩ ne lève pas une exception. Par contre, si ⟨s⟩ lève une exception, par l'exécution d'une instruction **raise**, alors l'exécution (en cours) de ⟨s⟩ est annulée. Toutes les informations associées à ⟨s⟩ sont enlevées de la pile sémantique. Le contrôle est transféré à $\langle s \rangle_1$ et une référence à l'exception est donnée à ⟨x⟩.

Toute valeur partielle peut être une exception. Le mécanisme de traitement d'exceptions est donc extensible par le programmeur. De nouvelles exceptions peuvent être définies au besoin par le programme. Le programmeur peut prévoir de nouvelles situations exceptionnelles. Parce qu'une exception peut être une variable non liée, la lever et la déterminer peuvent être faits de façon concurrente. En d'autres termes, une exception peut être levée (et capturée) avant de savoir de quelle exception il s'agit ! C'est tout à fait raisonnable dans un langage avec des variables dataflow : il est possible que nous sachions l'existence d'un problème sans savoir la nature exacte du problème.

Un exemple

Nous donnons un petit exemple de traitement d'exceptions. Considérez la fonction suivante, qui évalue des expressions arithmétiques simples et renvoie le résultat :

```
fun {Eval E}
   if {IsNumber E} then E else
      case E of plus(X Y) then {Eval X}+{Eval Y}
      []       times(X Y) then {Eval X}*{Eval Y}
      else raise illFormedExpr(E) end
      end
   end
end
```

Dans cet exemple, nous disons qu'une expression est mal formée si elle n'est pas reconnue par `Eval`, c'est-à-dire si elle contient d'autres valeurs que des nombres, des `plus` et des `times`. Essayer l'évaluation d'une expression `E` qui est mal formée

lèvera une exception. L'exception est un tuple, `illFormedExpr(E)`, qui contient l'expression mal formée. Voici quelques appels d'`Eval` qui montrent comment gérer l'exception :

```
try
   {Browse {Eval plus(plus(5 5) 10)}}
   {Browse {Eval times(6 11)}}
   {Browse {Eval minus(7 10)}}
catch illFormedExpr(E) then
   {Browse ´*** Illegal expression ´#E#´ ***´}
end
```

Si un appel de `Eval` lève une exception, le contrôle sera transféré à la clause **`catch`** qui affichera un message d'erreur.

2.7.2 Le modèle déclaratif avec exceptions

Nous étendons le modèle déclaratif avec les exceptions. Le tableau 2.9 donne la syntaxe du langage noyau étendu. Les programmes peuvent utiliser deux nouvelles instructions, **`try`** et **`raise`**. Il y a aussi une troisième instruction, **`catch`** ⟨x⟩ **`then`** ⟨s⟩ **`end`**, qui est utilisée pour définir la sémantique mais qui n'est pas utilisable par le programmeur. L'instruction **`catch`** est une « balise » sur la pile sémantique qui définit la frontière d'un contexte de capture d'exceptions. Voici la sémantique de ces trois instructions.

⟨s⟩ ::=	
`skip`	Instruction vide
⟨s⟩$_1$ ⟨s⟩$_2$	Séquence d'instructions
`local` ⟨x⟩ **`in`** ⟨s⟩ **`end`**	Création de variable
⟨x⟩$_1$=⟨x⟩$_2$	Lien variable-variable
⟨x⟩=⟨v⟩	Création de valeur
`if` ⟨x⟩ **`then`** ⟨s⟩$_1$ **`else`** ⟨s⟩$_2$ **`end`**	Instruction conditionnelle
`case` ⟨x⟩ **`of`** ⟨pattern⟩ **`then`** ⟨s⟩$_1$ **`else`** ⟨s⟩$_2$ **`end`**	Correspondance de formes
{⟨x⟩ ⟨y⟩$_1$ ⋯ ⟨y⟩$_n$}	Application de procédure
`try` ⟨s⟩$_1$ **`catch`** ⟨x⟩ **`then`** ⟨s⟩$_2$ **`end`**	**Contexte d'exception**
`raise` ⟨x⟩ **`end`**	**Lève exception**

Tableau 2.9 Le langage noyau déclaratif avec exceptions.

L'instruction **try**

L'instruction sémantique est :

(**try** ⟨s⟩$_1$ **catch** ⟨x⟩ **then** ⟨s⟩$_2$ **end**, E)

L'exécution fait les actions suivantes :

- Empiler l'instruction sémantique (**catch** ⟨x⟩ **then** ⟨s⟩$_2$ **end**, E).
- Empiler (⟨s⟩$_1$, E).

L'instruction **raise**

L'instruction sémantique est :

(**raise** ⟨x⟩ **end**, E)

L'exécution fait les actions suivantes :

- Dépiler les éléments de la pile sémantique à la recherche d'une instruction **catch**.
  - Si une instruction **catch** est trouvée, la dépiler.
  - Si la pile est vidée et aucune **catch** n'est trouvée, arrêter l'exécution avec le message d'erreur « `Uncaught exception` » (« `Exception non capturée` »).
- Soit (**catch** ⟨y⟩ **then** ⟨s⟩ **end**, E_c) l'instruction **catch** qui est trouvée. Empiler alors (⟨s⟩, $E_c + \{⟨y⟩ \rightarrow E(⟨x⟩)\}$).

Voici comment le système Mozart traite une exception non capturée. Pour l'exécution interactive, un message d'erreur est affiché dans la fenêtre de l'émulateur Oz. Pour une application, l'application termine et un message d'erreur est envoyé sur `stderr`, la sortie standard d'erreur du processus. Il est possible de changer ce comportement vers quelque chose de plus adapté à l'application en utilisant le module `Property`.

L'instruction **catch**

L'instruction sémantique est :

(**catch** ⟨x⟩ **then** ⟨s⟩ **end**, E)

L'exécution dépile simplement cette instruction. L'instruction **catch** ne fait rien, tout comme **skip**.

2.7.3 La syntaxe complète

Le tableau 2.10 donne la syntaxe de l'instruction **try** dans le langage complet. Il y a une clause **finally** qui est facultative. La clause **catch** peut prendre une série de formes, comme une instruction **case**. Nous verrons comment ces extensions sont définies.

⟨statement⟩	::=	**try** ⟨inStatement⟩
		[**catch** ⟨pattern⟩ **then** ⟨inStatement⟩
		{ ´[]´ ⟨pattern⟩ **then** ⟨inStatement⟩ }]
		[**finally** ⟨inStatement⟩] **end**
	|	**raise** ⟨inExpression⟩ **end**
	|	...
⟨inStatement⟩	::=	[{ ⟨declPart⟩ }+ **in**] ⟨statement⟩
⟨inExpression⟩	::=	[{ ⟨declPart⟩ }+ **in**] [⟨statement⟩] ⟨expression⟩

Tableau 2.10 La syntaxe des exceptions.

La clause **finally**

Une instruction **try** peut contenir une clause **finally** qui est toujours exécutée, que l'instruction lève une exception ou pas. La nouvelle syntaxe :

try ⟨s⟩$_1$ **finally** ⟨s⟩$_2$ **end**

est traduite en langage noyau comme :

try ⟨s⟩$_1$ **catch** X **then** ⟨s⟩$_2$ **raise** X **end end** ⟨s⟩$_2$

(avec un nouvel identificateur X qui n'est pas libre dans ⟨s⟩$_2$). Il est possible de définir une traduction où ⟨s⟩$_2$ n'apparaisse qu'une fois ; nous laissons cette possibilité aux exercices.

La clause **finally** est utile pour gérer des entités qui sont à l'extérieur du modèle de calcul. Avec **finally**, nous pouvons garantir qu'une action de « nettoyage » sera toujours effectuée sur l'entité qu'une exception ait lieu ou pas. Un exemple typique est la lecture d'un fichier. Supposons que F soit un fichier ouvert.[16] La procédure ProcessFile traite le fichier et la procédure CloseFile le ferme. Le programme suivant garantit que F sera toujours fermé après l'exécution de ProcessFile, qu'une exception soit levée ou pas :

```
try {ProcessFile F} finally {CloseFile F} end
```

16. Nous verrons plus tard comment traiter l'entrée/sortie des fichiers (voir section 3.7).

Remarquez que cette instruction **try** ne capture pas l'exception ; elle exécute simplement `CloseFile` après `ProcessFile`. Nous pouvons combiner la capture de l'exception et l'exécution d'une instruction terminale :

```
try {ProcessFile F}
catch X then
   {Browse ´*** Exception ´#X#´ avec le fichier ***´}
finally {CloseFile F} end
```

Cette instruction se comporte comme deux instructions **try** imbriquées : la plus intérieure avec seulement une clause **catch** et la plus extérieure avec seulement une clause **finally**.

La correspondance de formes

Une instruction **try** peut utiliser la correspondance de formes pour capturer uniquement les exceptions qui correspondent à une forme donnée. Les autres sont passées à l'instruction **try** suivante (qui entoure la première). La nouvelle syntaxe :

```
try ⟨s⟩
catch ⟨p⟩1 then ⟨s⟩1
   [] ⟨p⟩2 then ⟨s⟩2
   ...
   [] ⟨p⟩n then ⟨s⟩n
end
```

est traduite en langage noyau comme :

```
try ⟨s⟩ catch X then
   case X
   of ⟨p⟩1 then ⟨s⟩1
   [] ⟨p⟩2 then ⟨s⟩2
   ...
   [] ⟨p⟩n then ⟨s⟩n
   else raise X end end
end
```

Si l'exception ne correspond à aucune des formes, elle sera simplement levée de nouveau.

2.7.4 Les exceptions système

Le système Mozart lui-même lève certaines exceptions qui s'appellent des exceptions système. Elles sont des enregistrements avec une des trois étiquettes `failure` (« échec »), `error` (« erreur ») ou `system` (« système ») :

- `failure` : indique une tentative de réalisation d'un lien incompatible (par exemple, `1=2`) en mémoire. On l'appelle aussi un **échec d'unification**.
- `error` : indique une erreur dans l'exécution du programme, C'est une situation qui ne devrait pas se produire pendant une opération normale. Ce sont des erreurs de type ou des erreurs de domaine. Une erreur de type se produit quand une opération est invoquée avec un argument de type erroné, par exemple essayer d'appeler une non procédure (`{foo 1}`, où `foo` est un atome), ou d'additionner un entier avec un atome (`X=1+a`). Une erreur de domaine se produit quand une opération est invoquée avec un argument en dehors de son domaine d'application (même si le type est correct), par exemple la racine carrée d'un entier négatif, la division par zéro ou la sélection d'un champ d'enregistrement qui n'existe pas.
- `system` : indique une erreur qui arrive dans l'environnement du processus Mozart du système d'exploitation, par exemple une situation imprévisible comme un fichier ou une fenêtre qui sont fermés de l'extérieur d'une application ou l'échec d'une tentative d'ouverture d'une connexion entre deux processus Mozart dans la programmation répartie.[17]

Les informations dans l'enregistrement de l'exception dépendent de la version de Mozart. Le programmeur ne devrait donc se fier qu'à l'étiquette. Par exemple :

```
fun {One} 1 end
fun {Two} 2 end
try {One}={Two}
catch
   failure(...) then {Browse caughtFailure}
end
```

La forme `failure(...)` correspond à tout enregistrement dont l'étiquette est `failure`.

2.8 EXERCICES

➤ **Exercice 1** — *Les identificateurs libres et liés*

Prenez l'instruction suivante :

```
proc {P X}
   if X>0 then {P X-1} end
end
```

17. Voir chapitre 11 de [97].

La deuxième occurrence de l'identificateur `P` est-elle libre ou liée ? Justifiez votre réponse.

Indice : La réponse sera évidente si vous traduisez d'abord en langage noyau.

➤ **Exercice 2** — *L'environnement contextuel*

La section 2.4 explique comment un appel de procédure est exécuté. Prenez la procédure `MulByN` avec la définition suivante :

```
declare MulByN N in
N=3
proc {MulByN X ?Y}
   Y=N*X
end
```

et l'appel `{MulByN A B}`. Supposez que l'environnement à l'appel contient (entre autres) $\{\texttt{A} \rightarrow 10, \texttt{B} \rightarrow x_1\}$. Quand le corps de la procédure est exécuté, un nouvel environnement est calculé. La correspondance $\texttt{N} \rightarrow 3$ est ajoutée à ce nouvel environnement. Pourquoi est-ce nécessaire ? En particulier, $\texttt{N} \rightarrow 3$ n'existerait-elle pas déjà quelque part dans l'environnement à l'appel ? Cela ne serait-il pas suffisant pour s'assurer que l'identificateur `N` correspond déjà à `3` pendant l'exécution de la procédure ? Donnez un exemple où `N` n'existe pas dans l'environnement à l'appel. Donnez alors un deuxième exemple où `N` existe à l'appel mais correspond à une autre valeur que `3`.

➤ **Exercice 3** — *Les fonctions et les procédures*

Si le corps d'une fonction a une instruction **`if`** avec une clause **`else`** qui manque, une exception sera levée quand la condition du **`if`** sera fausse. Expliquez pourquoi ce comportement est correct. Cette situation n'arrive pas avec les procédures. Expliquez pourquoi.

➤ **Exercice 4** — *Les instructions* **`if`** *et* **`case`**

Cet exercice explore la relation entre les instructions **`if`** et **`case`**.

a) Définissez l'instruction **`if`** en utilisant l'instruction **`case`**. Cela montre que l'instruction conditionnelle n'ajoute pas d'expressivité par rapport à la correspondance de formes. Il aurait pu être ajouté comme une abstraction linguistique.

b) Définissez l'instruction **`case`** avec l'instruction **`if`**, en utilisant les opérations `Label`, `Arity` et `´.´` (sélection de champ).

Cela montre que l'instruction **`if`** est essentiellement une version plus primitive de l'instruction **`case`**.

➤ **Exercice 5** — *L'instruction* **`case`**

Cet exercice teste vos connaissances de l'instruction **`case`** complète. Prenez la procédure suivante :

```
proc {Test X}
   case X
   of a|Z then {Browse ´case´(1)}
   [] f(a) then {Browse ´case´(2)}
   [] Y|Z andthen Y==Z then {Browse ´case´(3)}
   [] Y|Z then {Browse ´case´(4)}
   [] f(Y) then {Browse ´case´(5)}
   else {Browse ´case´(6)} end
end
```

Sans exécuter l'exemple dans le *Labo interactif* ou le système Mozart, pronostiquez ce qui se passe quand on exécute les appels suivants : `{Test [b c a]}`, `{Test f(b(3))}`, `{Test f(a)}`, `{Test f(a(3))}`, `{Test f(d)}`, `{Test [a b c]}`, `{Test [c a b]}`, `{Test a|a}` et `{Test ´|´(a b c)}`. Utilisez la traduction en langage noyau et la sémantique si nécessaire pour faire vos prédictions. Après les prédictions, vérifiez vos connaissances par l'exécution des exemples.

➤ **Exercice 6** — *Encore l'instruction* **case**

Prenez la procédure suivante :

```
proc {Test X}
   case X of f(a Y c) then {Browse ´case´(1)}
   else {Browse ´case´(2)} end
end
```

De la même manière que l'exercice précédent, pronostiquez ce qui se passe quand on exécute :

```
declare X Y {Test f(X b Y)}
```

La même chose pour :

```
declare X Y {Test f(a Y d)}
```

La même chose pour :

```
declare X Y {Test f(X Y d)}
```

Utilisez la traduction en langage noyau et la sémantique si nécessaire pour faire les prédictions. Après les prédictions, vérifiez vos connaissances par l'exécution des exemples.

➤ **Exercice 7** — *Les fermetures à portée lexicale*

Prenez le code suivant :

```
declare Max3 Max5
proc {SpecialMax Value ?SMax}
   fun {SMax X}
      if X>Value then X else Value end
   end
end
{SpecialMax 3 Max3}
{SpecialMax 5 Max5}
```

De la même manière que l'exercice précédent, pronostiquez ce qui se passe quand on exécute :

```
{Browse [{Max3 4} {Max5 4}]}
```

Vérifiez vos connaissances par l'exécution de cet exemple.

➤ **Exercice 8** — *La récursion terminale*

Cet exercice examine l'importance de la récursion terminale dans la lumière de notre sémantique. Prenez les deux fonctions suivantes :

```
fun {Sum1 N}
   if N==0 then 0 else N+{Sum1 N-1} end
end
fun {Sum2 N S}
   if N==0 then S else {Sum2 N-1 N+S} end
end
```

Maintenant faites les choses suivantes :

a) Traduisez les deux définitions en syntaxe noyau. Il doit être clair que `Sum2` est récursive terminale et `Sum1` ne l'est pas.

b) Exécutez les deux appels `{Sum1 10}` et `{Sum2 10 0}` à la main, en utilisant la sémantique de ce chapitre pour suivre ce qui se passe sur la pile et dans la mémoire. Quelle est la taille maximale de la pile dans chaque cas ?

c) Que se passerait-il pendant l'exécution si vous appeliez `{Sum1 100000000}` ou `{Sum2 100000000 0}` ? Lequel des deux fonctionnera vraisemblablement ? Lequel ne fonctionnera pas ? Exécutez les deux pour vérifier votre raisonnement.

➤ **Exercice 9** — *La traduction en langage noyau*
Prenez la fonction `SMerge` suivante qui fusionne deux listes triées :

```
fun {SMerge Xs Ys}
   case Xs#Ys of nil#Ys then Ys
   [] Xs#nil then Xs
   [] (X|Xr)#(Y|Yr) then
      if X=<Y then X|{SMerge Xr Ys}
      else Y|{SMerge Xs Yr} end
   end
end
```

Traduisez `SMerge` en syntaxe noyau. Remarquez que `X#Y` est un tuple de deux arguments qui peut être écrit `´#´(X Y)`. La procédure qui résulte devrait être récursive terminale si vous suivez correctement les règles de la section 2.6.2.

➤ **Exercice 10** — *La récursion mutuelle*
L'optimisation terminale est importante pour bien plus que les appels récursifs. Prenez cette définition mutuellement récursive des fonctions `IsOdd` et `IsEven` :

```
fun {IsEven X}
   if X==0 then true else {IsOdd X-1} end
end

fun {IsOdd X}
   if X==0 then false else {IsEven X-1} end
end
```

On dit que ces fonctions sont mutuellement récursives parce que chaque fonction appelle l'autre. La récursion mutuelle peut être généralisée à un nombre quelconque de fonctions. Un ensemble de fonctions sera mutuellement récursif si on peut les mettre dans une séquence telle que chaque fonction appelle la suivante et la dernière appelle la première. Pour cet exercice, montrez que les appels `{IsOdd N}` et `{IsEven N}` s'exécutent avec une taille constante de la pile et ce pour tous les `N` zéro ou positifs. En général, si chaque fonction dans un ensemble mutuellement récursif n'a qu'un appel de fonction dans son corps et que cet appel est un dernier appel, alors toutes les fonctions dans l'ensemble s'exécuteront avec la taille de leur pile bornée par une constante.

➤ **Exercice 11** — *Les exceptions avec une clause* **`finally`**
La section 2.7 montre comment définir l'instruction **`try`**/**`finally`** par une traduction en une instruction **`try`**/**`catch`**. Pour cet exercice, définissez une autre traduction de

`try` $\langle s \rangle_1$ **`finally`** $\langle s \rangle_2$ **`end`**

dans laquelle $\langle s \rangle_1$ et $\langle s \rangle_2$ n'apparaissent qu'une seule fois.
Indice : Il faut une variable booléenne.

Chapitre **3**

Techniques de programmation déclarative

> *Le côté agréable de la programmation déclarative est que vous pouvez écrire une spécification et l'exécuter comme un programme. Le côté désagréable de la programmation déclarative est que certaines spécifications claires font des programmes incroyablement mauvais. L'espoir de la programmation déclarative est que vous pourrez progresser d'une spécification vers un programme raisonnable sans quitter le langage.*
>
> – *The Craft of Prolog*, Richard O'Keefe (1990)

Considérez un programme qui prend des entrées et renvoie des sorties. Nous disons que l'opération calculée par ce programme est déclarative si, quand on l'appelle avec les mêmes entrées, elle renvoie les mêmes résultats indépendamment de son contexte. La figure 3.1 illustre le concept. Une opération déclarative a trois propriétés : elle est **indépendante** (ne dépend pas d'un état d'exécution en dehors d'elle-même), **sans état** (n'a pas d'état d'exécution interne qui est gardé entre les appels) et **déterministe** (calcule toujours les mêmes résultats avec les mêmes entrées). Nous montrerons que tous les programmes écrits avec le modèle du chapitre précédent sont déclaratifs.

Pourquoi la programmation déclarative est importante

La programmation déclarative est importante à cause de deux propriétés :

– *Les programmes sont compositionnels*. Un programme déclaratif est fait de composants. Chaque composant peut être écrit, testé et prouvé correct indépendamment des autres composants et de sa propre histoire (les appels précédents).

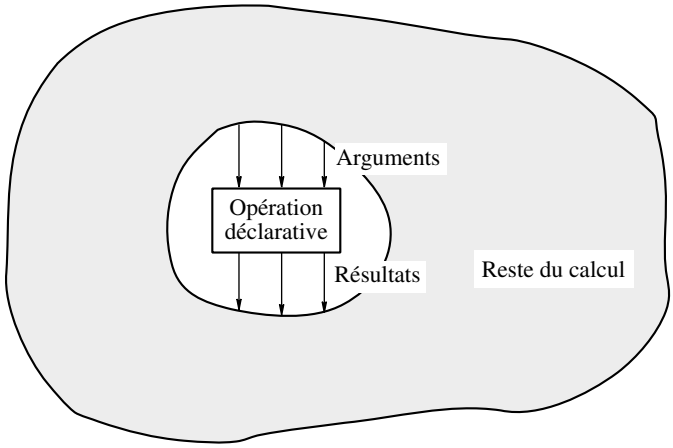


Figure 3.1 Une opération déclarative à l'intérieur d'un contexte de calcul.

– *Le raisonnement est facile pour les programmes*. Il est plus facile à raisonner sur les programmes déclaratifs que sur les programmes écrits dans les modèles plus expressifs. Comme les programmes déclaratifs ne calculent qu'avec des valeurs, des techniques simples de raisonnement algébrique et logique peuvent être utilisées.

Ces deux propriétés sont importantes pour la programmation à grande échelle et la programmation à petite échelle. Il serait vraiment agréable que tous les programmes puissent être écrits facilement dans le modèle déclaratif. Malheureusement, ce n'est pas le cas. Le modèle déclaratif est une bonne approche pour certains genres de programmes et une mauvaise approche pour d'autres. Ce chapitre et le suivant étudient les techniques de programmation du modèle déclaratif et expliquent les genres de programmes pour lesquels ce modèle est approprié.

Commençons par regarder de plus près la première propriété. Nous définissons un **composant** comme étant un fragment de programme avec une frontière précise entre l'extérieur et l'intérieur, ainsi que des entrées et des sorties bien définies. Un composant peut être construit en utilisant des composants plus simples. Par exemple, dans le modèle déclaratif, une procédure est une sorte de composant et une procédure peut appeler des procédures. Le programme principal est le composant le plus haut dans une hiérarchie de composants. La hiérarchie s'arrête vers le bas avec les composants primitifs qui sont fournis par le système.

Dans un programme déclaratif, l'interaction entre composants est déterminée exclusivement par les entrées et les sorties de chaque composant. Un composant déclaratif peut être compris tout seul, sans besoin de comprendre le reste du programme. L'effort

nécessaire pour comprendre tout le programme est donc la somme de l'effort pour comprendre le composant déclaratif et de l'effort pour comprendre le reste.

S'il y avait des interactions plus rapprochées entre le composant et le reste du programme, alors ils ne pourraient pas être compris indépendamment. Ils devraient être compris ensemble et l'effort nécessaire serait beaucoup plus grand. Par exemple, il pourrait être proportionnel (grosso modo) au produit des efforts nécessaires pour chaque partie. Pour un programme avec beaucoup de composants qui interagissent souvent, l'effort total serait énorme (exponentiel dans le nombre de composants), ce qui rend la compréhension difficile ou impossible. Un exemple d'un programme avec des interactions rapprochées est un programme concurrent dont les fils se partagent un état.[1]

Des interactions plus rapprochées sont souvent nécessaires. Elles ne peuvent pas être éliminées « par décret » en programmant dans un modèle qui ne les soutient pas. Mais un principe important est qu'elles devraient être utilisées uniquement quand elles sont nécessaires et pas autrement. Pour soutenir ce principe, autant de composants que possible devraient être déclaratifs.

Le développement des programmes déclaratifs

La façon la plus simple d'écrire un programme déclaratif est d'utiliser le modèle du chapitre 2. Toutes les opérations de base sur les types de base sont déclaratives. Il est possible de combiner les opérations déclaratives pour faire de nouvelles opérations déclaratives si on respecte certaines règles. La combinaison des opérations déclaratives selon les opérations du modèle déclaratif donne toujours une nouvelle opération déclarative (voir la section 3.1.3).

La règle standard en algèbre qui dit que « nous pouvons remplacer des égaux par des égaux » est un autre exemple d'une combinaison déclarative. Dans les langages de programmation, cette propriété s'appelle la **transparence référentielle**. Elle simplifie énormément le raisonnement sur les programmes. Par exemple, si nous savons que $f(a) = a^2$, nous pourrons remplacer $f(a)$ par a^2 partout où elle apparaît. L'équation $b = 7f(a)^2$ devient alors $b = 7a^4$. C'est possible parce que $f(a)$ est déclarative : elle dépend uniquement de ses arguments et pas d'un autre état du calcul.

La technique de base pour écrire les programmes déclaratifs est de considérer le programme comme un ensemble de fonctions récursives, en utilisant la programmation d'ordre supérieur pour simplifier la structure. Une fonction récursive est une fonction dont le corps de la définition fait référence à la même fonction, directement ou indirectement. La récursion directe veut dire que la fonction elle-même est utilisée dans le corps. La récursion indirecte veut dire que la fonction référence une autre

1. Voir chapitre 8 de [97].

fonction qui fait référence directement ou indirectement à la fonction originale. La programmation d'ordre supérieur veut dire que les fonctions peuvent avoir d'autres fonctions comme arguments et comme résultats. Cette capacité sous-tend toutes les techniques pour construire les abstractions que nous montrerons dans ce livre. L'ordre supérieur peut compenser partiellement le manque d'expressivité du modèle déclaratif. Il facilite la programmation des formes limitées de concurrence et d'état au sein du modèle déclaratif.

La structure du chapitre

Ce chapitre explique comment écrire des programmes déclaratifs pratiques. Il est divisé en huit parties qui couvrent les sujets montrés dans la figure 3.2. La section 3.1 définit le concept de « déclarativité ». Les sections 3.2, 3.3 et 3.4 font un survol des techniques de programmation (les calculs récursifs et itératifs). Les sections 3.5 et 3.6 complètent ce survol (les abstractions et l'efficacité). La section 3.7 explique comment un programme déclaratif interagit avec le monde extérieur (les besoins non déclaratifs). La section 3.8 explique la programmation à petite échelle : comment organiser la conception de programmes. Enfin, nous donnons quelques conclusions sur les limitations du modèle déclaratif (voir section 4.4 dans le chapitre 4).

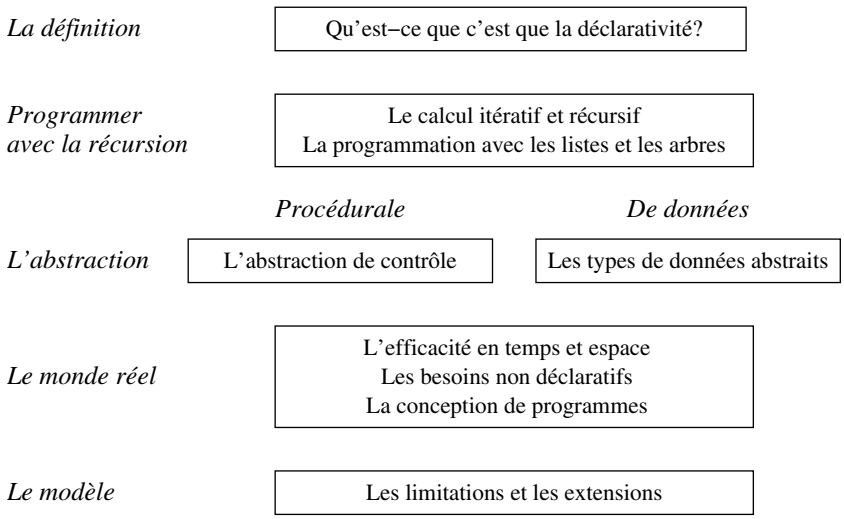


Figure 3.2 Les sujets couverts par ce chapitre.

3.1 LA DÉCLARATIVITÉ C'EST QUOI ?

Le modèle déclaratif du chapitre 2 est particulièrement approprié pour écrire des programmes déclaratifs parce que tous les programmes écrits dans ce modèle seront déclaratifs de ce fait même. Mais il y a de nombreuses autres façons de faire de la programmation déclarative. Avant d'expliquer comment programmer dans le modèle déclaratif, nous le situons par rapport aux autres manières d'être déclaratif. Nous expliquons aussi pourquoi les programmes écrits dans ce modèle sont toujours déclaratifs.

3.1.1 Une classification de la programmation déclarative

À l'origine, la programmation déclarative c'est programmer en définissant le *pourquoi* (les résultats que nous voulons) sans expliquer le *comment* (les algorithmes, etc., nécessaires pour les calculer). Cette intuition vague couvre beaucoup d'idées différentes. La figure 3.3 montre une classification. Le premier niveau de la classification est basé sur l'expressivité. Il y a deux possibilités :

– Une déclarativité descriptive. C'est le modèle le moins expressif. Le « programme » déclaratif définit simplement une structure de données. Le tableau 3.1 définit un langage à ce niveau. Ce langage peut seulement créer des enregistrements ! Il ne contient que les cinq premières instructions du langage noyau du tableau 2.1. La section 3.7.2 montre comment utiliser ce langage pour définir des interfaces graphiques. D'autres exemples sont des langages de formattage comme le HTML (« *Hypertext Markup Language* »), qui donne la structure d'un document sans dire comment faire le formattage, et des langages d'échange d'informations comme le XML (« *Extensible Markup Language* ») qui définit un format ouvert facilement lisible par tous. Mais le niveau descriptif est trop faible pour écrire des programmes généraux. Alors pourquoi est-il intéressant ? Parce qu'il est fait des structures de données que l'on peut facilement manipuler. Les enregistrements du tableau 3.1, les documents HTML et XML, et les interfaces graphiques déclaratives de la section 3.7.2 peuvent tous être créés et transformés facilement par un programme.
– Une déclarativité programmable. C'est aussi expressif qu'une machine de Turing.[2] Par exemple, le tableau 2.1 définit un langage à ce niveau. (L'introduction du chapitre 5 contient plus d'informations sur la relation entre les niveaux descriptif et programmable.)

2. Une machine de Turing est un modèle de calcul simple, défini pour la première fois par Alan Turing, qui est aussi puissant que tout ordinateur que l'on puisse construire, selon l'état des connaissances actuel en informatique. Tout calcul que l'on puisse programmer sur un ordinateur quelconque peut être programmé sur une machine de Turing.

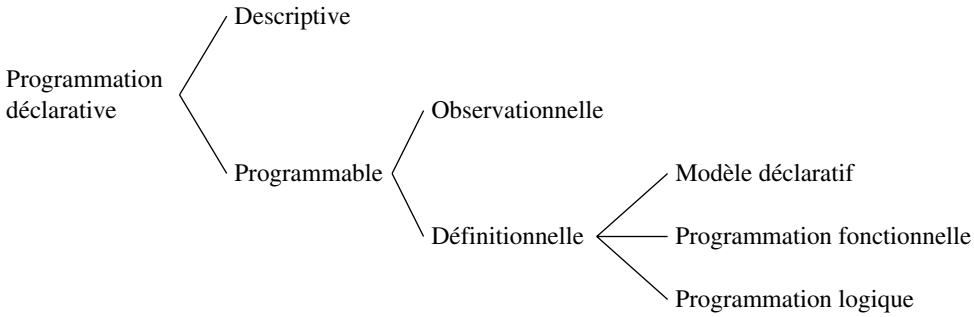


Figure 3.3 Une classification de la programmation déclarative.

⟨s⟩	::=	**skip**	Instruction vide
	|	⟨s⟩$_1$ ⟨s⟩$_2$	Séquence d'instructions
	|	**local** ⟨x⟩ **in** ⟨s⟩ **end**	Création de variable
	|	⟨x⟩$_1$=⟨x⟩$_2$	Lien variable-variable
	|	⟨x⟩=⟨v⟩	Création de valeur

Tableau 3.1 Le langage noyau déclaratif descriptif.

Il y a deux façons fondamentalement différentes de considérer la déclarativité programmable :

- Une vue définitionnelle, où la déclarativité est une propriété de l'implémentation des composants. Par exemple, les programmes écrits dans le modèle déclaratif sont toujours déclaratifs, à cause des propriétés du modèle.
- Une vue observationnelle, où la déclarativité est une propriété de l'interface des composants. La vue observationnelle respecte le principe d'abstraction : pour utiliser un composant il suffit de connaître sa spécification sans connaître son implémentation. Le composant doit simplement se comporter de façon déclarative, être indépendant, sans mémoire et déterministe, sans nécessairement être écrit dans un modèle déclaratif.

Ce livre utilise les deux vues, définitionnelle et observationnelle. Quand nous nous intéressons à l'intérieur d'un composant, nous utilisons la vue définitionnelle. Quand nous nous intéressons au comportement d'un composant, nous utilisons la vue observationnelle. Il y a deux styles de programmation déclarative définitionnelle qui sont relativement populaires : le style fonctionnel et le style logique. Dans le style fonctionnel, nous disons qu'un composant défini comme une fonction mathématique est déclaratif. Les langages fonctionnels tels que Haskell et Standard ML suivent cette approche. Dans le style logique, nous disons qu'un composant défini comme une relation logique est déclaratif. Les langages logiques tels que Prolog et Mercury suivent cette approche.

Il est plus difficile de manipuler les programmes fonctionnels ou logiques que les programmes descriptifs, mais ils respectent toujours des lois algébriques simples.[3] Notre modèle déclaratif couvre les deux styles, fonctionnel et logique.

La vue observationnelle nous permet d'utiliser des composants déclaratifs dans un programme déclaratif même s'ils sont écrits dans un modèle non déclaratif. Par exemple, une interface à une base de données peut être une extension valable à un langage déclaratif. Cependant, l'implémentation de cette interface n'est probablement pas logique ou fonctionnelle. Là n'est pas la question. Il suffit qu'elle ait pu être définie déclarativement. Parfois un composant déclaratif sera écrit dans un style fonctionnel ou logique. Dans les chapitres ultérieurs nous construisons des composants déclaratifs dans des modèles non déclaratifs. Nous ne serons pas dogmatiques à ce sujet ; nous considérerons un composant comme déclaratif s'il se comporte de façon déclarative.

3.1.2 Les langages de spécification

Les partisans de la programmation déclarative prétendent parfois qu'elle permet de se passer de l'implémentation parce qu'il n'y a que la spécification. Il disent que la spécification est un programme. C'est vrai en théorie, mais pas en pratique. En pratique, les programmes déclaratifs sont très semblables à d'autres programmes : ils ont besoin d'algorithmes, des structures de données et d'un raisonnement sur les opérations. Cette ressemblance existe parce que les langages déclaratifs doivent se restreindre aux mathématiques avec une implémentation efficace. Il y a un compromis entre l'expressivité et l'efficacité. Les programmes déclaratifs sont généralement beaucoup plus longs par rapport à ce que pourrait être une spécification. Nous concluons que la distinction entre spécification et implémentation a toujours un sens, même pour les programmes déclaratifs.

Il est possible de définir un langage déclaratif qui soit bien plus expressif que celui que nous utilisons. Un tel langage s'appelle un **langage de spécification**. Il est généralement impossible de faire une implémentation efficace d'un langage de spécification. Cela ne veut pas dire qu'un tel langage n'est pas pratique. Au contraire, c'est un outil important pour réfléchir sur les programmes. On peut l'utiliser avec un prouveur de théorèmes, qui est un programme qui peut faire certaines formes de raisonnement mathématique. Les prouveurs de théorèmes pratiques ne sont pas complètement automatiques ; ils ont besoin d'un coup de main humain. Mais ils peuvent prendre sur eux une grande partie de la corvée du raisonnement sur les programmes : la manipulation fastidieuse des formules mathématiques. Avec l'aide d'un prouveur de théorèmes, un développeur peut souvent prouver des propriétés très fortes de son programme. Une telle utilisation d'un prouveur de théorèmes s'appelle l'**ingénierie des preuves**. Pour l'instant, l'ingénierie des preuves n'est pratique que

3. Si on n'utilise pas les possibilités non déclaratives de ces langages !

pour de petits programmes. Mais cela suffit quand la sûreté a une importance capitale, par exemple quand il y a des vies en jeu comme dans les appareils médicaux ou les transports publics.

Les langages de spécification dépassent la portée de cet ouvrage.

3.1.3 L'implémentation des composants

Combiner des instructions déclaratives selon les constructions du modèle déclaratif donne une autre instruction déclarative. Expliquons d'abord ce qu'est une instruction déclarative. Prenez une instruction dans le modèle déclaratif. Partitionnez ses identificateurs libres en entrées et sorties. Les entrées sont liées aux valeurs partielles et les sorties sont liées aux variables non liées. Exécutez l'instruction. Cela donnera une des trois possibilités suivantes : (1) une exécution complète avec des liens pour les variables non liées, (2) une suspension (l'exécution n'est pas complète) ou (3) une exception. Dans les cas (2) et (3), nous ne regardons pas les liens des variables. Si l'instruction est déclarative, pour les mêmes entrées le résultat sera toujours le même.

Par exemple, considérez l'instruction `Z=X`. Supposez que `X` est l'entrée et `Z` la sortie. Pour tout lien de `X` à une valeur partielle, l'exécution de cette instruction liera `Z` à la même valeur partielle. L'instruction est donc déclarative.

Nous pouvons utiliser ce résultat pour prouver que

```
if X>Y then Z=X else Z=Y end
```

est déclarative. Partitionnez les trois identificateurs libres, `X`, `Y` et `Z`, en deux entrées `X` et `Y` et une sortie `Z`. Alors, si `X` et `Y` sont liées à deux valeurs partielles quelconques, l'exécution de l'instruction suspendra ou liera `Z` à une de ces deux valeurs, déterminée par la comparaison `X>Y`. L'instruction est donc déclarative.

Nous pouvons faire ce raisonnement pour toutes les opérations du modèle déclaratif :

- D'abord, toutes les opérations de base sont déclaratives. Ces opérations sont expliquées dans la section 2.3.5.
- Ensuite, combiner les opérations déclaratives avec les constructions du modèle déclaratif donne d'autres opérations déclaratives. Il y a cinq instructions composées dans le modèle déclaratif :
  - La composition séquentielle.
  - L'instruction **`local`**.
  - L'instruction **`if`**.
  - L'instruction **`case`**.
  - La déclaration de procédure, l'instruction ⟨x⟩=⟨v⟩ quand ⟨v⟩ est une valeur procédurale.

Ces opérations permettent de construire des instructions en utilisant d'autres instructions. Toutes ces manières de combiner des instructions sont déterministes (si les instructions qui les composent sont déterministes, elles le sont aussi) et elles ne dépendent pas d'un quelconque contexte.

3.2 LE CALCUL ITÉRATIF

Nous commençons la présentation des techniques de programmation par une technique simple, le calcul itératif. C'est une boucle dont la taille de la pile est bornée par une constante, indépendamment du nombre d'itérations. Le calcul itératif est un outil de base. Comme il n'est pas toujours évident de savoir quand un programme est itératif, nous allons donner un schéma général avec lequel nous pouvons construire des calculs itératifs.

3.2.1 Un schéma général

Une classe importante de calculs itératifs commence avec un état initial S_0 et transforme cet état en pas successifs jusqu'à l'état final S_{final} :

$$S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_{\text{final}}$$

Un calcul itératif de cette classe peut être fait avec le schéma suivant :

```
fun {Iterate Si}
   if {IsDone Si} then Si
   else Si+1 in
      Si+1={Transform Si}
      {Iterate Si+1}
   end
end
```

Pour utiliser ce schéma il faut donner des définitions pour les fonctions *IsDone* et *Transform*. Nous prouvons que tout programme qui suit ce schéma est itératif. Il suffit de démontrer que la taille de la pile ne grandit pas pendant l'exécution de `Iterate`. Pour la clarté, nous donnons simplement les instructions sur la pile sémantique et nous omettons les environnements et la mémoire :

- Prenons la pile sémantique initiale [`R={Iterate` S_0`}`].
- Supposons que `{`*IsDone* S_0`}` renvoie **false**. Juste après l'exécution du **if**, la pile sémantique est [S_1`={`*Transform* S_0`}`, `R={Iterate` S_1`}`].
- Après l'exécution de `{`*Transform* S_0`}`, la pile sémantique est [`R={Iterate` S_1`}`].

Il est clair que la pile sémantique n'a qu'un élément juste avant chaque appel récursif, l'instruction `R={Iterate` S_{i+1}`}`. C'est le même raisonnement que nous avons utilisé pour expliquer la récursion terminale dans la section 2.5.1.

3.2.2 L'itération avec des nombres

Un bon exemple du calcul itératif est la **méthode de Newton** pour calculer la racine carrée d'un nombre réel positif x. L'idée est de commencer avec une estimation g de la racine carrée et d'améliorer cette estimation itérativement jusqu'à ce qu'elle soit suffisamment précise. Dans chaque itération, on calcule une estimation améliorée g' en prenant la moyenne de g et x/g :

$$g' = (g + x/g)/2.$$

Pour voir que l'estimation g' est meilleure, nous utilisons la différence ε entre l'estimation et $\sqrt{x}$:

$$\varepsilon = g - \sqrt{x}$$

Nous pouvons calculer la différence entre g' et $\sqrt{x}$ comme

$$\varepsilon' = g' - \sqrt{x} = (g + x/g)/2 - \sqrt{x} = \varepsilon^2/2g$$

Pour la convergence, ε' doit être plus petit que ε. Quelle condition cela impose-t-il sur x et g ? La condition $\varepsilon' < \varepsilon$ est la même que $\varepsilon^2/2g < \varepsilon$, qui est la même que $\varepsilon < 2g$. (En supposant que $\varepsilon > 0$; sinon nous commençons avec ε', qui est toujours plus grand que 0.) En substituant la définition de ε, nous obtenons la condition $\sqrt{x} + g > 0$. Si $x > 0$ et l'estimation initiale $g > 0$, cette condition sera toujours vraie. L'algorithme converge donc toujours.

La figure 3.4 montre un programme itératif pour la méthode de Newton. La fonction `{SqrtIter Guess X}` appelle `{SqrtIter {Improve Guess X} X}` jusqu'à ce que `Guess` satisfasse la condition `{GoodEnough Guess X}`. Il est clair que ce calcul est une instance du schéma général, c'est donc un calcul itératif. L'estimation améliorée est calculée selon la formule ci-dessus. Le test de « suffisamment précis » est $|x - g^2|/x < 0.00001$: la racine carrée a une précision de cinq décimales. On dit que ce test est relatif, parce que l'erreur est divisée par x. Nous pourrions aussi utiliser un test absolu, par exemple $|x - g^2| < 0.00001$, où la grandeur de l'erreur est plus petite qu'une constante. Pourquoi le test relatif est-il meilleur pour le calcul des racines carrées ?

```
fun {Sqrt X}
Guess=1.0 in
   {SqrtIter Guess X}
end
fun {SqrtIter Guess X}
   if {GoodEnough Guess X} then Guess
   else {SqrtIter {Improve Guess X} X} end
end
fun {Improve Guess X}
  (Guess + X/Guess) / 2.0
end
fun {GoodEnough Guess X}
  {Abs X-Guess*Guess}/X < 0.00001
end
fun {Abs X} if X<0.0 then ~X else X end end
```

Figure 3.4 Trouver les racines avec la méthode de Newton (première version).

3.2.3 L'utilisation des procédures locales

Dans le programme de la figure 3.4, il y a plusieurs routines auxiliaires : `SqrtIter`, `Improve`, `GoodEnough` et `Abs`. Ces routines sont des briques de base pour la fonction principale `Sqrt`. Dans cette section, nous étudions le meilleur emplacement du programme pour les définir. Le principe de base est le suivant : une routine auxiliaire qui est définie uniquement comme une aide pour la définition d'une autre routine ne devrait pas être visible ailleurs. (Nous utilisons le terme « routine » pour désigner les fonctions et les procédures.)

Dans l'exemple de Newton, `SqrtIter` est utilisée uniquement à l'intérieur de `Sqrt`, `Improve` et `GoodEnough` ne sont utilisées qu'à l'intérieur de `SqrtIter` et `Abs` pourrait être utilisée ailleurs. Il y a deux façons pour exprimer ces visibilités, avec des sémantiques légèrement différentes. La première façon est illustrée dans la figure 3.5 : les routines auxiliaires sont définies à l'extérieur de `Sqrt` dans une instruction **`local`**. La deuxième façon est illustrée dans la figure 3.6 : chaque routine auxiliaire est définie à l'intérieur de la routine qui en a besoin.[4]

Dans la figure 3.5, il y a un compromis entre la lisibilité et la visibilité : `Improve` et `GoodEnough` pourraient être définies à l'intérieur de `SqrtIter`. Cela donnerait deux niveaux d'instructions **`local`**, ce qui est plus difficile à lire. Nous avons décidé de mettre les trois routines au même niveau dans une instruction **`local`**.

4. Nous omettons la définition de `Abs` pour éviter une répétition gratuite.

```
local
   fun {Improve Guess X}
     (Guess + X/Guess) / 2.0
   end
   fun {GoodEnough Guess X}
     {Abs X-Guess*Guess}/X < 0.00001
   end
   fun {SqrtIter Guess X}
      if {GoodEnough Guess X} then Guess
      else {SqrtIter {Improve Guess X} X} end
   end
in
   fun {Sqrt X}
   Guess=1.0 in
      {SqrtIter Guess X}
   end
end
```

Figure 3.5 Trouver les racines avec la méthode de Newton (deuxième version).

```
fun {Sqrt X}
   fun {SqrtIter Guess X}
      fun {Improve Guess X}
        (Guess + X/Guess) / 2.0
      end
      fun {GoodEnough Guess X}
        {Abs X-Guess*Guess}/X < 0.00001
      end
   in
      if {GoodEnough Guess X} then Guess
      else {SqrtIter {Improve Guess X} X} end
   end
   Guess=1.0
in {SqrtIter Guess X} end
```

Figure 3.6 Trouver les racines avec la méthode de Newton (troisième version).

Dans la figure 3.6, chaque routine voit les arguments de la routine qui l'entoure comme des références externes. Ces arguments sont justement ceux avec lesquels les routines sont appelées. Nous pouvons donc simplifier la définition en les enlevant. Cela donne la figure 3.7.

```
fun {Sqrt X}
   fun {SqrtIter Guess}
      fun {Improve}
        (Guess + X/Guess) / 2.0
      end
      fun {GoodEnough}
        {Abs X-Guess*Guess}/X < 0.00001
      end
   in
      if {GoodEnough} then Guess
      else {SqrtIter {Improve}} end
   end
   Guess=1.0
in {SqrtIter Guess} end
```

Figure 3.7 Trouver les racines avec la méthode de Newton (quatrième version).

Il y a un compromis pour le placement des définitions auxiliaires par rapport à la routine qui en a besoin : les mettre à l'extérieur ou à l'intérieur.

- Le placement intérieur (les figures 3.6 et 3.7) permet aux auxiliaires de voir les arguments des routines principales comme des références externes, selon la règle de la portée lexicale (voir section 2.4.3). Elles ont donc besoin de moins d'arguments. Mais chaque fois que la routine principale est appelée, de nouvelles routines auxiliaires sont créées. Cela veut dire la création de nouvelles valeurs procédurales.
- Le placement extérieur (les figures 3.4 et 3.5) permet de créer les valeurs procédurales une fois pour toutes, pour tous les appels de la routine principale. Mais les routines auxiliaires ont besoin de plus d'arguments.

Dans la figure 3.7, des nouvelles définitions de `Improve` et `GoodEnough` sont créées pour chaque itération de `SqrtIter`, tandis que `SqrtIter` n'est créée qu'une fois. Cela suggère un bon compromis, où `SqrtIter` est locale à `Sqrt` et `Improve` et `GoodEnough` sont toutes les deux à l'extérieur de `SqrtIter`. Nous obtenons alors la définition finale de la figure 3.8, que nous considérons comme la meilleure pour l'efficacité et la visibilité.

```
fun {Sqrt X}
   fun {Improve Guess}
     (Guess + X/Guess) / 2.0
   end
   fun {GoodEnough Guess}
     {Abs X-Guess*Guess}/X < 0.00001
   end
   fun {SqrtIter Guess}
      if {GoodEnough Guess} then Guess
      else {SqrtIter {Improve Guess}} end
   end
   Guess=1.0
in {SqrtIter Guess} end
```

Figure 3.8 Trouver les racines avec la méthode de Newton (cinquième version).

3.2.4 Du schéma général vers une abstraction de contrôle

Le schéma général de la section 3.2.1 aide le programmeur à concevoir des programmes efficaces, mais il n'existe que dans l'esprit du programmeur. Nous pouvons aller plus loin en programmant le schéma général comme un composant qui peut être utilisé par d'autres composants. Nous disons que le schéma devient une abstraction de contrôle : une abstraction qui assure l'ordre désiré des opérations. Voici sa définition :

```
fun {Iterate Si}
   if {IsDone Si} then Si
   else Si+1 in
      Si+1={Transform Si}
      {Iterate Si+1}
   end
end
```

Ce schéma réalise une boucle **while** qui calcule un résultat. Pour convertir le schéma en une abstraction de contrôle, nous devons le paramétriser en extrayant les parties qui varient d'une utilisation à une autre. Il y a deux parties : les fonctions *IsDone* et *Transform*. Nous en faisons deux paramètres de `Iterate` :

```
fun {Iterate S IsDone Transform}
   if {IsDone S} then S
   else S1 in
      S1={Transform S}
      {Iterate S1 IsDone Transform}
   end
end
```

Pour utiliser cette abstraction de contrôle, il faut passer des fonctions d'un argument à `IsDone` et `Transform`. C'est un exemple de programmation d'ordre supérieur. `Iterate` se comportera exactement comme `SqrtIter` si on passe les fonctions `GoodEnough` et `Improve` :

```
fun {Sqrt X}
   {Iterate
      1.0
      fun {$ G} {Abs X-G*G}/X<0.00001 end
      fun {$ G} (G+X/G)/2.0 end}
end
```

Il y a deux valeurs fonctionnelles comme arguments à l'abstraction de contrôle. C'est une technique puissante pour la structuration des programmes parce qu'elle fait une séparation entre l'ordre des opérations et la définition des opérations. La programmation d'ordre supérieur est particulièrement utile pour cette structuration. Si l'abstraction de contrôle est utilisée souvent, l'étape suivante pourra être d'en faire une abstraction linguistique.

3.3 LE CALCUL RÉCURSIF

Le calcul itératif est un cas spécial d'un calcul plus général qui s'appelle le **calcul récursif**. Rappelez-vous qu'un calcul itératif peut être considéré comme une boucle dans laquelle une action est répétée un certain nombre de fois. La section 3.2 l'implémente dans le modèle déclaratif avec une abstraction de contrôle, la fonction `Iterate`. La fonction teste d'abord une condition. Si la condition est fausse, elle fera une action et s'appellera elle-même.

La récursion est plus générale. Une fonction récursive peut s'appeler elle-même à tout endroit dans son corps et même plusieurs fois. La récursion est importante pour les fonctions mais aussi pour les types de données. Un type de données récursif est défini par rapport à lui-même. Par exemple, le type liste est défini en utilisant le type liste : une liste est une liste vide ou un élément suivi par une liste. Les deux formes de récursion sont fortement apparentées parce que les fonctions récursives peuvent être utilisées pour calculer avec les types de données récursifs.

Nous avons vu qu'un calcul itératif a une taille de pile constante à cause de l'optimisation terminale. Ce n'est pas toujours le cas pour un calcul récursif. La taille de sa pile peut grandir quand l'entrée grandit. Parfois c'est inévitable, par exemple en faisant des calculs avec des arbres, comme nous le verrons plus loin. En d'autres cas, on peut l'éviter. Un aspect important de la programmation déclarative est d'éviter une pile grandissante quand c'est possible. Cette section donne un exemple qui montre comment on peut faire. Nous prenons un cas typique d'un calcul récursif qui n'est pas itératif, la définition naïve de la fonction factorielle. La définition mathématique est :

$$0! = 1$$
$$n! = n \cdot (n-1)! \text{ if } n > 0$$

C'est une équation de récurrence. La factorielle $n!$ est définie par rapport à une factorielle avec un argument plus petit, $(n-1)!$. Le programme naïf suit cette définition mathématique. Pour calculer `{Fact N}` il y a deux possibilités, `N=0` ou `N>0`. Dans le premier cas, il renvoie `1`. Dans le deuxième cas, il calcule `{Fact N-1}`, le multiplie par `N` et renvoie le résultat. Voici le programme :

```
fun {Fact N}
   if N==0 then 1
   elseif N>0 then N*{Fact N-1}
   else raise domainError end end
end
```

Ce programme définit la factorielle d'un grand nombre en utilisant la factorielle d'un nombre plus petit. Puisque tous les nombres sont non négatifs, tôt ou tard on s'arrêtera à zéro et l'exécution se terminera.

Remarquez que la factorielle est une fonctionne partielle sur les entiers. Elle n'est pas définie pour `N` négatif. Le programme indique cela en levant une exception pour `N` négatif. La définition du chapitre 1 a une erreur puisque pour `N` négatif elle fait une boucle infinie.

Nous avons fait deux choses pour écrire `Fact`. D'abord, nous avons suivi la définition mathématique pour obtenir une définition correcte. Ensuite, nous avons raisonné sur la terminaison. Nous avons montré que le programme se termine pour tous arguments légaux, dans le domaine de la fonction.

3.3.1 La taille grandissante de la pile

Cette définition de la factorielle définit un calcul dont la taille maximale de la pile est proportionnelle à l'argument `N`. Nous pouvons le vérifier avec la sémantique. D'abord, traduisons `Fact` en langage noyau :

```
proc {Fact N ?R}
   if N==0 then R=1
   elseif N>0 then N1 R1 in
      N1=N-1
      {Fact N1 R1}
      R=N*R1
   else raise domainError end end
end
```

Déjà on peut deviner que la taille de la pile pourrait grandir puisqu'il y a une multiplication après l'appel récursif. Pendant l'appel récursif la pile doit garder des informations pour pouvoir faire cette multiplication quand l'appel revient. En suivant la sémantique nous pouvons voir ce qui se passe pendant l'appel `{Fact 5 R}`. Pour la clarté, nous simplifions légèrement la présentation de la machine abstraite en substituant la valeur d'une variable dans l'environnement. L'environnement $\{\ldots, \texttt{N} \rightarrow n, \ldots\}$ sera donc écrit comme $\{\ldots, \texttt{N} \rightarrow 5, \ldots\}$ si la mémoire est $\{\ldots, n = 5, \ldots\}$.

- La pile sémantique initiale est $[(\{\texttt{Fact N R}\}, \{\texttt{N} \rightarrow 5, \texttt{R} \rightarrow r_0\})]$.
- Au premier appel récursif :

$$[(\{\texttt{Fact N1 R1}\}, \{\texttt{N1} \rightarrow 4, \texttt{R1} \rightarrow r_1, \ldots\}),\\ (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_0, \texttt{R1} \rightarrow r_1, \texttt{N} \rightarrow 5, \ldots\})]$$

- Au deuxième appel récursif :

$$[(\{\texttt{Fact N1 R1}\}, \{\texttt{N1} \rightarrow 3, \texttt{R1} \rightarrow r_2, \ldots\}),\\ (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_1, \texttt{R1} \rightarrow r_2, \texttt{N} \rightarrow 4, \ldots\}),\\ (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_0, \texttt{R1} \rightarrow r_1, \texttt{N} \rightarrow 5, \ldots\})]$$

- Au troisième appel récursif :

$$[(\{\texttt{Fact N1 R1}\}, \{\texttt{N1} \rightarrow 2, \texttt{R1} \rightarrow r_3, \ldots\}),\\ (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_2, \texttt{R1} \rightarrow r_3, \texttt{N} \rightarrow 3, \ldots\}),\\ (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_1, \texttt{R1} \rightarrow r_2, \texttt{N} \rightarrow 4, \ldots\}),\\ (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_0, \texttt{R1} \rightarrow r_1, \texttt{N} \rightarrow 5, \ldots\})]$$

Il est clair que la pile grandit d'une instruction à chaque appel récursif. Le dernier appel récursif est le cinquième, qui retourne tout de suite avec $r_5 = 1$. Cinq multiplications sont faites ensuite pour obtenir le résultat final $r_0 = 120$.

3.3.2 Une machine abstraite basée sur les substitutions

Cet exemple montre que la machine abstraite du chapitre 2 est encombrante pour le calcul à la main. C'est parce qu'elle garde les identificateurs en utilisant des environnements pour faire la correspondance avec les variables. Cette approche est réaliste ; c'est

comme cela que la machine abstraite est réalisée sur un ordinateur. Mais l'approche n'est pas très agréable pour le calcul à la main.

Nous pouvons faire une modification simple à la machine abstraite pour faciliter son utilisation pour le calcul à la main. L'idée est de remplacer les identificateurs dans les instructions par les entités en mémoire correspondantes. Cette opération s'appelle une **substitution**. Par exemple, l'instruction `R=N*R1` devient $r_2 = 3 * r_3$ quand on fait la substitution selon $\{\texttt{R} \rightarrow r_2, \texttt{N} \rightarrow 3, \texttt{R1} \rightarrow r_3\}$.

La machine abstraite basée sur les substitutions n'a pas d'environnements. Elle substitue directement dans chaque instruction les identificateurs par les entités en mémoire. Pour l'exemple de factorielle récursive, cela donne :

- La pile sémantique initiale est [{`Fact 5` r_0}].
- Au premier appel récursif : [{`Fact 4` r_1}, r_0=5$*r_1$].
- Au deuxième appel récursif : [{`Fact 3` r_2}, r_1=4$*r_2$, r_0=5$*r_1$].
- Au troisième appel récursif : [{`Fact 2` r_3}, r_2=3$*r_3$, r_1=4$*r_2$, r_0=5$*r_1$].

Nous voyons de nouveau que la pile grandit d'une instruction par appel. Nous résumons les différences entre les deux variantes de la machine abstraite :

- La machine abstraite basée sur les environnements est fidèle à l'implémentation sur un ordinateur qui utilise des environnements. Cependant, les environnements introduisent un niveau supplémentaire d'indirection, ce qui les rend difficiles pour le calcul à la main.
- La machine abstraite basée sur les substitutions est plus facile pour le calcul à la main parce qu'il y a beaucoup moins de symboles à manipuler. Cependant, les substitutions sont plus coûteuses à implémenter, elles ne sont donc généralement pas utilisées dans une implémentation pratique.

Les deux variantes font les mêmes liens en mémoire et les mêmes manipulations de la pile sémantique.

3.3.3 La conversion d'un calcul récursif en calcul itératif

La factorielle est suffisamment simple pour que l'on puisse la rendre itérative. Plus loin, nous donnerons une méthode systématique pour définir des calculs itératifs. Pour l'instant, nous allons montrer une technique pour transformer la factorielle en calcul itératif. Dans le calcul que l'on vient de voir :

```
R=(5*(4*(3*(2*(1*1)))))
```

il suffit de changer l'ordre des calculs :

```
R=((((1*5)*4)*3)*2)*1)
```

Dans ce nouvel ordre, le calcul peut être fait incrémentalement, en commençant avec `1*5`. Cela donne `5`. En continuant ainsi, nous obtenons `20`, `60`, `120` et finalement `120`. La définition itérative qui fait ce calcul est

```
fun {Fact N}
   fun {FactIter N A}
      if N==0 then A
      elseif N>0 then {FactIter N-1 A*N}
      else raise domainError end end
   end
in {FactIter N 1} end
```

La fonction qui fait l'itération, `FactIter`, a un deuxième argument `A`. Cet argument est essentiel ; sans un deuxième argument, une factorielle itérative est impossible. Le deuxième argument n'est pas manifeste dans la définition mathématique de la factorielle que nous avons utilisée. Il faut un raisonnement pour démontrer son utilité au programme.

3.4 LA PROGRAMMATION AVEC LA RÉCURSION

Les calculs récursifs sont au cœur de la programmation déclarative. Cette section explique comment écrire dans ce style. Nous montrons les techniques de base pour la programmation avec les listes, les arbres et d'autres types de données récursifs. Nous montrons aussi comment rendre les calculs itératifs quand c'est possible. Cette section contient les parties suivantes :

- Le premier pas est la définition des **types de données récursifs**. La section 3.4.1 présente une notation simple qui nous permet de définir les types de données récursifs les plus importants.
- Le type de données récursif le plus important est la **liste**. La section 3.4.2 montre les techniques de base pour programmer avec les listes.
- Les programmes déclaratifs efficaces doivent définir des calculs itératifs. La section 3.4.3 montre les **accumulateurs**, une technique systématique pour atteindre ce but.
- Le deuxième type de données récursif le plus important, après les structures linéaires comme les listes, est l'**arbre**. La section 3.4.4 montre les techniques de base pour programmer avec les arbres.

3.4.1 La définition des types

Le type de liste est un sous-ensemble du type d'enregistrement. Il y a d'autres sous-ensembles utiles de l'enregistrement, comme les arbres binaires. Avant d'écrire des

programmes, nous introduisons une notation simple pour définir des listes, des arbres et d'autres sous-ensembles des enregistrements. Cela nous aidera pour écrire des fonctions sur ces types.

Une liste `Xs` est définie comme étant soit `nil` soit `X|Xr` où `Xr` est une liste. Un arbre binaire peut être défini comme étant soit un nœud feuille `leaf` soit un nœud non feuille `tree(key:K value:V left:LT right:RT)` où `LT` et `RT` sont des arbres binaires. Pour écrire ces définitions de façon claire et précise, nous proposons une notation simple basée sur les grammaires hors-contexte. Les non terminaux représentent des types ou des valeurs. Nous utilisons la hiérarchie des types de la figure 2.16 comme base : tous les types de cette hiérarchie seront disponibles comme non terminaux prédéfinis. Ainsi ⟨Value⟩ et ⟨Record⟩ existent tous les deux, et comme ils sont des ensembles de valeurs, nous pouvons dire ⟨Record⟩ ⊂ ⟨Value⟩. Maintenant nous pouvons définir le type de liste :

⟨List⟩ ::= `nil`
| ⟨Value⟩ ´|´ ⟨List⟩

Une valeur sera dans l'ensemble ⟨List⟩ si elle est l'atome `nil` ou si elle est `X|Xr` où `X` est dans ⟨Value⟩ et `Xr` est dans ⟨List⟩. C'est une définition récursive de ⟨List⟩. On peut prouver qu'il y a un ensemble unique qui est le plus petit ensemble qui satisfait cette définition. La preuve est hors de notre portée, mais on peut la trouver dans tous les livres introductifs sur la sémantique, comme par exemple [99]. Nous prenons ce plus petit ensemble comme la valeur de ⟨List⟩. Intuitivement, ⟨List⟩ peut être construit en commençant avec `nil` et en répétant la règle de grammaire pour construire des ensembles de listes de plus en plus grands, jusqu'à obtenir un point fixe.

Nous pouvons aussi définir les listes dont les éléments sont d'un type donné :

⟨List T⟩ ::= `nil`
| T ´|´ ⟨List T⟩

Ici T est une variable de type et ⟨List T⟩ est une fonction de type. Appliquer la fonction de type à un type quelconque renvoie le type d'une liste de ce type. Par exemple, ⟨List ⟨Int⟩⟩ est le type d'une liste d'entiers. Remarquez que ⟨List ⟨Value⟩⟩ est équivalent à ⟨List⟩ (parce qu'ils ont la même définition).

Nous pouvons définir un arbre binaire avec des littéraux comme clés et qui contient d'éléments de type T :

⟨BTree T⟩ ::= `leaf`
| `tree(key:` ⟨Literal⟩ `value:` T
`left:` ⟨BTree T⟩ `right:` ⟨BTree T⟩`)`

Le type d'une procédure est ⟨**proc** {$ $T_1 \cdots T_n$}⟩, où $T_1, \ldots, T_n$ sont les types de ses arguments. Le type de la procédure est appelé la signature de la procédure. Le type d'une fonction est ⟨**fun** {$ $T_1 \cdots T_n$} : T⟩, qui est équivalent à

⟨**proc** {$ $T_1 \cdots T_n$ T}⟩. Par exemple, le type ⟨**fun** {$ ⟨List⟩ ⟨List⟩} : ⟨List⟩ ⟩ est une fonction qui prend deux listes et qui renvoie une liste.

Les limites de la notation

Cette notation pour définir des types est utile pour définir beaucoup d'ensembles de valeurs, mais son expressivité est certainement limitée. Voici quelques cas où la notation ne suffit pas :

- La notation ne peut pas définir les entiers strictement positifs, à savoir le sous-ensemble de ⟨Int⟩ avec seulement les éléments plus grands que zéro.
- La notation ne peut pas définir des ensembles de valeurs partielles. Par exemple, les flots ne peuvent pas être définis.

Nous pouvons étendre la notation pour couvrir le premier cas, par exemple en ajoutant des conditions booléennes.[5] Dans les exemples suivants, nous ajouterons ces conditions dans le texte quand nous en aurons besoin. Cela veut dire que la notation des types est descriptive : elle donne des assertions logiques sur l'ensemble des valeurs qu'une variable peut avoir. On ne pourrait sans doute pas vérifier ces types dans un compilateur. Même les types qui sont simples à spécifier, comme les entiers strictement positifs, ne peuvent généralement pas être vérifiés.

3.4.2 La programmation avec les listes

Les valeurs de liste sont faciles à créer et à décomposer, cependant elles sont assez puissantes pour coder toutes sortes de structures de données complexes. Le langage Lisp tire beaucoup de sa puissance de cette idée [63]. À cause de la structure simple des listes, la programmation déclarative avec elles est facile et puissante. Cette section montre les techniques de base pour programmer avec des listes :

- *Penser récursivement*. L'idée est de résoudre un problème en utilisant des versions plus petites du problème.
- *Convertir des calculs récursifs en calculs itératifs*. Un programme naïf sur les listes est souvent inefficace parce que la taille de sa pile grandit avec la taille de l'entrée. Nous montrons comment utiliser les transformations d'état pour convertir ces programmes en programmes itératifs.
- *Raisonner sur l'exactitude des calculs itératifs*. Une façon de raisonner sur les calculs itératifs est l'utilisation des invariants d'état.

5. Cela ressemble à la manière dont on a défini la syntaxe du langage dans la section 2.1.1 : une notation hors-contexte supplémentée par des conditions quand on en a besoin.

– *Construire des programmes en suivant le type*. Une fonction qui calcule avec un type a presque toujours une structure récursive qui reflète de près la structure récursive du type.

Nous finissons cette section avec un exemple plus grand, l'algorithme de tri par fusion. Les sections ultérieures montrerons comment rendre plus systématique le développement de fonctions itératives en utilisant des accumulateurs. Ceux-ci nous permettent d'écrire des fonctions qui sont itératives dès le départ. Notre expérience montre que ces techniques fonctionnent bien aussi pour les grands programmes déclaratifs.

a) Penser récursivement

Une liste est une structure de données récursive, c'est-à-dire qu'elle est définie par rapport à une version plus petite d'elle-même. Pour écrire une fonction qui calcule avec des listes il faut suivre cette structure récursive. La fonction a donc deux parties :

– Un cas de base. Pour les petites listes (par exemple de zéro, un ou deux éléments) la fonction calcule la réponse directement.
– Un cas récursif. Pour les listes plus grandes, la fonction calcule le résultat en utilisant les résultats d'une ou plusieurs listes plus petites.

Comme premier exemple, nous prenons une fonction récursive qui calcule la longueur d'une liste :

```
fun {Length Ls}
   case Ls of nil then 0
   [] _|Lr then 1+{Length Lr} end
end
{Browse {Length [a b c]}}
```

Sa signature de type est ⟨**fun** `{$` ⟨List⟩ `}` **:** ⟨Int⟩⟩ : une fonction qui prend une liste et qui renvoie un entier. Le cas de base est la liste vide `nil`, pour laquelle la fonction renvoie `0`. Le cas récursif couvre les listes non vides. Pour une liste avec longueur n, sa queue a une longueur $n - 1$. Comme elle est plus petite que la liste originale, le programme terminera.

Notre deuxième exemple est la fonction `Append` qui fait la concaténation de deux listes `Ls` et `Ms` pour construire une troisième liste. Sur quel argument faisons-nous l'induction, le premier ou le deuxième ? On peut démontrer que l'induction doit être faite sur le premier argument. Voici la fonction `Append` :

```
fun {Append Ls Ms}
   case Ls of nil then Ms
   [] X|Lr then X|{Append Lr Ms} end
end
```

Sa signature de type est ⟨**fun** {$ ⟨List⟩ ⟨List⟩} : ⟨List⟩⟩. Cette fonction suit exactement les deux propriétés suivantes de la concaténation :

$$\begin{aligned} \text{append}(\texttt{nil},\ m) &= m \\ \text{append}(x\,|\,l,\ m) &= x\ |\ \text{append}(l, m) \end{aligned}$$

Le cas récursif appelle toujours `Append` avec un premier argument plus petit, donc le programme terminera.

b) Les fonctions récursives et leurs domaines

Définissons la fonction `Nth` pour obtenir le nième élément d'une liste.

```
fun {Nth Xs N}
   if N==1 then Xs.1
   elseif N>1 then {Nth Xs.2 N-1} end
end
```

Son type est ⟨**fun** {$ ⟨List⟩ ⟨Int⟩} : ⟨Value⟩⟩. Souvenez-vous qu'une liste `Xs` est soit `nil` soit un tuple `X|Y` avec deux arguments. `Xs.1` est égal à `X` et `Xs.2` est égal à `Y`. Que se passe-t-il quand on fait ceci ? :

```
{Browse {Nth [a b c d] 5}}
```

La liste n'a que quatre éléments. Tenter d'obtenir le cinquième élément veut dire tenter de faire `Xs.1` ou `Xs.2` avec `Xs=nil`, ce qui lèvera une exception. Une exception sera levée aussi si `N` n'est pas plus grand que zéro, par exemple si `N=0`. C'est parce qu'il n'y a pas de clause **else** dans l'instruction **if**.

Cette fonction est un exemple d'une technique plus générale : utiliser des instructions qui lèvent des exceptions pour les valeurs en dehors de leurs domaines. Nous voudrions que la fonction lève une exception quand elle est appelée avec une entrée en dehors de son domaine. Nous ne pouvons pas garantir qu'une exception sera toujours levée dans ce cas, par exemple `{Nth 1|2|3 2}` renvoie 2 mais `1|2|3` n'est pas une liste. De telles garanties sont difficiles à obtenir sans faire plus de calculs. On peut parfois les obtenir dans les langages statiquement typés.

L'instruction **case** se comporte correctement à cet égard. L'utilisation d'une instruction **case** pour traverser récursivement une liste lèvera une exception quand l'argument n'est pas une liste. Par exemple, nous pouvons définir une fonction qui additionne tous les éléments d'une liste d'entiers :

```
fun {SumList Xs}
   case Xs of nil then 0
   [] X|Xr then X+{SumList Xr} end
end
```

Son type est ⟨**fun** {$ ⟨List ⟨Int⟩⟩} : ⟨Int⟩⟩. La liste d'entrée doit contenir des entiers parce que `SumList` utilise l'entier 0 dans sa définition. L'appel suivant

```
{Browse {SumList [1 2 3]}}
```

affiche 6. Comme `Xs` a deux valeurs possibles, à savoir `nil` ou `X|Xr`, il est normal d'utiliser une instruction **`case`**. Comme dans l'exemple `Nth`, ne pas utiliser une clause **`else`** dans la **`case`** lèvera une exception si l'argument est en dehors du domaine de la fonction. Par exemple :

```
{Browse {SumList 1|foo}}
```

lève une exception parce que `1|foo` n'est pas une liste, et la définition de `SumList` suppose que son entrée est une liste.

c) Les définitions naïves sont parfois lentes

Nous définissons une fonction pour inverser les éléments d'une liste. Nous commençons avec une définition récursive de l'inverse d'une liste :

- L'inverse de `nil` est `nil`.
- L'inverse de `X|Xs` est `Z`, où
  l'inverse de `Xs` est `Ys` et
  la concaténation de `Ys` et `[X]` est `Z`.

Cette définition est correcte ; on peut vérifier que le premier élément `X` devient le dernier et pour les autres on utilise un argument inductif. En suivant cette définition récursive, nous pouvons tout de suite écrire une fonction :

```
fun {Reverse Xs}
   case Xs of nil then nil
   [] X|Xr then {Append {Reverse Xr} [X]} end
end
```

Son type est ⟨**fun** {$ ⟨List⟩} : ⟨List⟩⟩. Cette fonction est-elle efficace ? Pour répondre à cette question, nous calculons son temps d'exécution avec une liste d'entrée de longueur n. Nous pouvons faire ce calcul rigoureusement avec les techniques de la section 3.6. Mais même sans ces techniques, nous pouvons voir intuitivement ce qui se passe. Il y a n appels récursifs suivis par n appels à `Append`. Chaque appel d'`Append` prend une liste de longueur $n/2$ en moyenne. Le temps total d'exécution est donc proportionnel à $n \cdot n/2$, à savoir n^2. C'est assez lent. Nous nous attendrions à ce que l'inversion d'une liste prenne un temps proportionnel à la longueur de la liste et pas à son carré.

Ce programme a un deuxième défaut : la taille de la pile grandit avec la longueur de l'entrée. Il définit un calcul récursif qui n'est pas itératif. Suivre naïvement la définition récursive de l'inverse nous a donné un programme assez mauvais ! Heureusement, il y a des techniques simples pour éliminer ces deux défauts. Nous verrons une technique importante : la transformation d'état.

d) La conversion d'un calcul récursif en calcul itératif

Nous allons convertir un calcul récursif en calcul itératif. Au lieu de `Reverse`, prenons une fonction plus simple qui calcule la longueur d'une liste :

```
fun {Length Xs}
   case Xs of nil then 0
   [] _|Xr then 1+{Length Xr} end
end
```

Cette fonction a la même structure que la fonction `SumList`. Comme `SumList`, elle a un temps linéaire mais la taille de la pile est proportionnelle à la profondeur de la récursion, qui est égal à la longueur de `Xs`. Ce problème vient du fait que l'addition `1+{Length Xr}` est faite après l'appel récursif. L'appel récursif n'est pas le dernier appel, donc l'environnement de la fonction ne peut pas être récupéré à l'appel.

Comment pouvons-nous calculer la longueur de la liste avec un calcul itératif ? Pour faire cela, nous devons formuler le problème comme une séquence de transformations d'état. Nous commençons avec un état S_0 et nous le transformons successivement, obtenant S_1, S_2, ..., jusqu'à ce que nous arrivions à l'état final S_{final}, qui contient la réponse. Pour calculer la longueur de la liste, nous allons prendre comme état la longueur i de la partie de la liste déjà vue. En fait, ce n'est qu'une partie de l'état. Le reste de l'état est la partie `Ys` de la liste non encore rencontrée. L'état complet S_i est donc la paire $(i, \texttt{Ys})$. Le cas général pour l'état intermédiaire S_i est (si la liste `Xs` est $[e_1\ e_2\ \cdots\ e_n]$) :

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

À chaque appel récursif, i sera augmenté de 1 et `Ys` diminué d'un élément. Cela nous donne la fonction suivante :

```
fun {IterLength I Ys}
   case Ys of nil then I
   [] _|Yr then {IterLength I+1 Yr} end
end
```

Son type est ⟨**fun** {$ ⟨Int⟩ ⟨List⟩} : ⟨Int⟩⟩. Remarquez la différence entre les deux définitions. Ici l'addition `I+1` est faite avant l'appel récursif à `IterLength`, qui est le dernier appel. Nous avons défini un calcul itératif.

Dans l'appel `{IterLength I Ys}`, il y a un deuxième argument avec valeur initiale de `0`. Nous pouvons cacher cet argument en définissant `IterLength` comme une procédure locale. La définition finale de `Length` est donc

```
local
   fun {IterLength I Ys}
      case Ys of nil then I
      [] _|Yr then {IterLength I+1 Yr} end
   end
in
   fun {Length Xs} {IterLength 0 Xs} end
end
```

ce qui définit un calcul itératif pour calculer la longueur d'une liste. Nous définissons `IterLength` à l'extérieur de `Length`. Cela nous permet d'éviter de créer une nouvelle valeur procédurale à chaque appel de `Length`. Il n'y a pas d'avantage à définir `IterLength` à l'intérieur de `Length`, parce qu'elle n'utilise pas l'argument `Xs` de `Length`.

Nous pouvons utiliser la même technique pour `Reverse` que celle que nous avons utilisée pour `Length`. Dans le cas de `Reverse`, l'état contient l'inverse de la partie de la liste déjà vue au lieu de sa longueur. La mise à jour de l'état est facile : il suffit d'ajouter un nouvel élément au début de la liste. L'état initial est `nil`. Ces idées nous donnent la version suivante de `Reverse` :

```
local
   fun {IterReverse Rs Ys}
      case Ys of nil then Rs
      [] Y|Yr then {IterReverse Y|Rs Yr} end
   end
in
   fun {Reverse Xs} {IterReverse nil Xs} end
end
```

Cette version de `Reverse` a un temps linéaire et une exécution itérative.

e) L'exactitude avec les invariants d'état

Prouvons que `IterLength` est correct. Nous utiliserons une technique générale qui fonctionne bien pour `IterReverse` et d'autres calculs itératifs. L'idée est de définir une propriété $P(S_i)$ de l'état et de prouver qu'elle est toujours vraie. On dit que $P(S_i)$ est un invariant d'état. Si `P` est bien choisi, l'exactitude du calcul sera une conséquence de $P(S_{\text{final}})$. Pour `IterLength` nous définissons P comme ceci :

$$P((i, \texttt{Ys})) \quad \equiv \quad (\text{length}(\texttt{Xs}) = i + \text{length}(\texttt{Ys}))$$

où length(L) donne la longueur de la liste L. Nous soupçonnons que $P((i, \texttt{Ys}))$ est un invariant d'état. Nous utilisons l'induction pour le prouver :

- D'abord nous vérifions $P(S_0)$. C'est une conséquence directe de $S_0 = (0, \texttt{Xs})$.
- En supposant $P(S_i)$ et S_i n'est pas l'état final, il faut prouver $P(S_{i+1})$. C'est une conséquence de la sémantique de l'instruction **case** et l'appel de fonction. Nous avons $S_i = (i, \texttt{Ys})$. Nous ne sommes pas dans l'état final, Ys a donc une longueur non zéro. D'après la sémantique, I+1 ajoute 1 à i et l'instruction **case** enlève un élément de Ys. En conséquence, $P(S_{i+1})$ est vrai.

Comme Ys est réduit d'un élément à chaque appel, nous arrivons tôt ou tard à l'état final $S_{\text{final}} = (i, \texttt{nil})$, et la fonction renvoie i. Comme length(nil) $= 0$ et $P(S_{\text{final}})$ est vraie, nous déduisons que $i =$ length(Xs).

L'étape difficile dans cette preuve est le choix de la propriété P. Elle doit satisfaire deux contraintes. D'abord, elle doit combiner les arguments du calcul itératif de telle façon que le résultat ne change pas pendant le calcul. Ensuite, elle doit être assez forte pour que l'exactitude soit une conséquence de $P(S_{\text{final}})$. Une bonne règle pour trouver P est d'exécuter le programme à la main pour quelques cas simples, et de formuler à partir de ces résultats le cas intermédiaire général.

f) La construction des programmes en suivant le type

Ces exemples de fonctions sur les listes ont tous une propriété curieuse. Ils ont tous un argument de liste, ⟨List T⟩, qui est défini comme :

```
⟨List T⟩  ::=  nil
            |  T ´|´ ⟨List T⟩
```

et ils ont tous une instruction **case** qui a la forme :

```
case Xs of nil then ⟨expr⟩   % Cas de base
[] X|Xr then ⟨expr⟩ end      % Appel récursif
```

Que se passe-t-il ici ? La structure récursive des fonctions sur les listes suit exactement la structure récursive de la définition du type. Nous verrons que c'est presque toujours vrai pour les fonctions sur les listes.

Nous pouvons utiliser cette propriété pour nous aider à écrire des fonctions récursives. Cela peut énormément faciliter le travail quand les définitions des types deviennent compliquées. Par exemple, définissons une fonction qui compte le nombre d'éléments dans une liste imbriquée. Une liste imbriquée (« *nested list* ») est une liste dans laquelle chaque élément peut lui-même être une liste, comme `[[1 2] 4 nil [[5] 10]]`. Nous définissons le type ⟨NestedList T⟩ comme ceci :

```
⟨NestedList T⟩  ::=  nil
                   |  ⟨NestedList T⟩ ´|´ ⟨NestedList T⟩
                   |  T ´|´ ⟨NestedList T⟩
```

Pour éviter l'ambiguïté, il faut ajouter une condition sur T, par exemple que T n'est ni `nil` ni une paire de liste. Maintenant nous pouvons écrire la fonction `{LengthL` ⟨NestedList T⟩`}` : ⟨Int⟩ qui compte le nombre d'éléments dans une liste imbriquée. En suivant la définition du type nous obtenons le squelette suivant :

```
fun {LengthL Xs}
   case Xs of nil then ⟨expr⟩
   [] X|Xr andthen {IsList X} then
      ⟨expr⟩+⟨expr⟩  % Appels récursifs pour X et Xr
   [] X|Xr then
      ⟨expr⟩  % Appel récursif pour Xr
   end
end
```

(On peut omettre `{Not {IsList X}}` dans la troisième clause parce que c'est une conséquence de la négation de la deuxième clause.) Ici `{IsList X}` est une fonction qui vérifie si `X` est `nil` ou une paire de liste :

```
fun {IsList X} X==nil orelse {IsCons X} end
fun {IsCons X}
   case X of _|_ then true else false end end
```

Compléter le squelette donne la fonction suivante :

```
fun {LengthL Xs}
   case Xs of nil then 0
   [] X|Xr andthen {IsList X} then
      {LengthL X}+{LengthL Xr}
   [] X|Xr then
      1+{LengthL Xr}
   end
end
```

Voici deux appels :

```
X=[[1 2] 4 nil [[5] 10]]
{Browse {LengthL X}}
{Browse {LengthL [X X]}}
```

Qu'est-ce qui est affiché par ces appels ?

En utilisant une autre définition du type pour les listes imbriquées nous obtenons une autre fonction de longueur. Par exemple, nous pouvons définir le type ⟨NestedList2 T⟩ comme ceci :

⟨NestedList2 T⟩ ::= `nil`
| ⟨NestedList2 T⟩ ´ | ´ ⟨NestedList2 T⟩
| T

De nouveau, il faut ajouter la condition que T n'est ni `nil` ni une paire de liste. Remarquez la différence subtile entre ⟨NestedList T⟩ et ⟨NestedList2 T⟩ ! En suivant la définition de ⟨NestedList2 T⟩ nous obtenons une autre fonction `LengthL2` plus simple :

```
fun {LengthL2 Xs}
   case Xs of nil then 0
   [] X|Xr then {LengthL2 X}+{LengthL2 Xr}
   else 1 end
end
```

Quelle est la différence entre `LengthL` et `LengthL2` ? Nous la déduisons en comparant les types ⟨NestedList T⟩ et ⟨NestedList2 T⟩. Une ⟨NestedList T⟩ est toujours une liste mais une ⟨NestedList2 T⟩ peut aussi avoir le type T. Donc l'appel `{LengthL2 foo}` est légal (il renvoie 1), mais `{LengthL foo}` est illégal (il lève une exception). Du point de vue du comportement désiré (l'entrée doit être une liste), nous concluons qu'il est raisonnable de considérer `LengthL2` comme erronée et `LengthL` comme correcte.

Il y a une leçon importante à retenir ici. La définition d'un type récursif doit être faite avant d'écrire la fonction qui l'utilise. Sinon il est facile de se laisser tromper par une fonction qui semble simple mais qui est erronée. C'est vrai aussi dans les langages fonctionnels qui font de l'inférence de types, comme Standard ML et Haskell. L'inférence de types peut vérifier qu'un type récursif est utilisé correctement, mais la conception d'un type récursif reste sous la responsabilité du programmeur.

g) Le tri par fusion

Nous définissons une fonction qui prend une liste de nombres ou atomes et qui renvoie une nouvelle liste triée en ordre croissant. Elle utilise l'opérateur de comparaison <, tous les éléments doivent donc être du même type (tous des entiers, flottants ou atomes). Nous utilisons l'algorithme de tri par fusion (« *mergesort* »), qui est efficace et qui peut être programmé facilement dans un modèle déclaratif. L'algorithme de tri par fusion est basé sur une stratégie simple qui s'appelle **diviser pour régner** :

- Découpez la liste en deux listes d'environ la même longueur.
- Utilisez le tri par fusion pour trier les deux listes.
- Fusionnez les deux listes triées pour obtenir le résultat final.

La figure 3.9 montre la structure récursive de cette stratégie. Le tri par fusion est efficace parce que les opérations de découpe et de fusion sont toutes les deux itératives

et linéaires en temps. Nous définissons d'abord les opérations de fusion (`Merge`) et de découpe (`Split`) et ensuite le tri (`Mergesort`) :

```
fun {Merge Xs Ys}
   case Xs # Ys of nil # Ys then Ys
   [] Xs # nil then Xs
   [] (X|Xr) # (Y|Yr) then
      if X<Y then X|{Merge Xr Ys}
      else Y|{Merge Xs Yr} end
   end
end
```

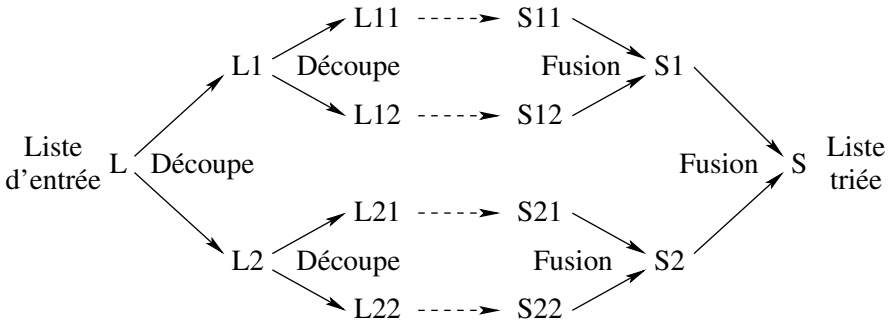


Figure 3.9 Le tri par fusion.

Le type est ⟨**fun** {$ ⟨List T⟩ ⟨List T⟩} : ⟨List T⟩⟩, où T est ⟨Int⟩, ⟨Float⟩ ou ⟨Atom⟩. Nous définissons la découpe comme une procédure parce qu'elle a deux sorties. Elle aurait pu être définie comme une fonction qui renvoie une paire comme seule sortie.

```
proc {Split Xs ?Ys ?Zs}
   case Xs of nil then Ys=nil Zs=nil
   [] [X] then Ys=[X] Zs=nil
   [] X1|X2|Xr then Yr Zr in
      Ys=X1|Yr
      Zs=X2|Zr
      {Split Xr Yr Zr}
   end
end
```

Le type est ⟨**proc** {$ ⟨List T⟩ ⟨List T⟩ ⟨List T⟩}⟩. Voici la définition du tri :

```
fun {MergeSort Xs}
   case Xs of nil then nil
   [] [X] then [X]
   else Ys Zs in
      {Split Xs Ys Zs}
      {Merge {MergeSort Ys} {MergeSort Zs}}
   end
end
```

Son type est ⟨**fun** {$ ⟨List T⟩} : ⟨List T⟩⟩ avec la même restriction sur T que `Merge`. La découpe de la liste d'entrée s'arrête avec les listes de longueur zéro et un, qui peuvent être triées immédiatement.

3.4.3 Les accumulateurs

Nous avons vu comment programmer des fonctions de liste simples et comment les rendre itératives. La programmation déclarative pratique est généralement abordée d'une autre façon, par la définition des fonctions qui sont itératives dès le début. L'idée est que le programme porte un état avec lui et le transforme juste avant chaque appel, sans jamais le transformer au moment d'un retour. Cela garde l'optimisation terminale. Un état `S` est représenté en ajoutant une paire d'arguments, `S1` et `Sn`, à chaque procédure. Cette paire s'appelle un **accumulateur**. `S1` représente l'état à l'entrée et `Sn` représente l'état à la sortie. Ainsi, chaque procédure est écrite dans le style suivant :

```
proc {P X S1 ?Sn}
   if {BaseCase X} then Sn=S1
   else
      {P1 S1 S2}
      {P2 S2 S3}
      {P3 S3 Sn}
   end
end
```

Comme le cas de base ne fait pas de calculs, l'état à la sortie est le même que l'état à l'entrée (`Sn=S1`). Le cas récursif enfile l'état dans chaque appel de procédure (`P1`, `P2` et `P3`) et finalement le retourne à `P`. La figure 3.10 montre ce qui se passe. Chaque flèche représente un argument. Nous faisons un enfilage d'état : la sortie de chaque procédure est l'entrée de la procédure suivante. Enfiler veut dire passer un fil ; ici on passe l'accumulateur comme un fil dans un tissu. Le technique d'enfiler un état dans les appels de procédure imbriqués s'appelle la **programmation par accumulateur**.

La programmation par accumulateur est utilisée dans les fonctions `IterLength` et `IterReverse` que nous avons vues auparavant. Dans ces fonctions la structure

des accumulateurs est moins claire, justement parce qu'elles sont des fonctions et pas des procédures. L'état à l'entrée est un argument de la fonction et l'état à la sortie est ce que renvoie la fonction.

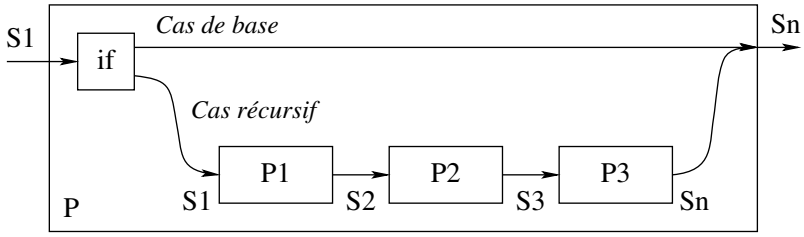


Figure 3.10 Une procédure avec état enfilé.

Un exemple avec plusieurs accumulateurs

Pour donner un exemple avec plusieurs accumulateurs, nous allons écrire un compilateur pour une simple machine à pile. Le compilateur prend une expression contenant des identificateurs, entiers et opérations d'addition (avec l'étiquette `plus`), et calcule deux résultats : le code machine pour une machine à pile et le nombre d'instructions dans ce code.

```
proc {ExprCode E C1 ?Cn S1 ?Sn}
   case E of plus(A B) then C2 C3 S2 S3 in
      C2=plus|C1
      S2=S1+1
      {ExprCode B C2 C3 S2 S3}
      {ExprCode A C3 Cn S3 Sn}
   [] I then
      Cn=push(I)|C1
      Sn=S1+1
   end
end
```

Cette procédure a deux accumulateurs : un pour construire la liste des instructions machine et un autre pour compter le nombre d'instructions. Voici un exemple de son exécution :

```
declare Code Size in
{ExprCode plus(plus(a 3) b) nil Code 0 Size}
{Browse Size#Code}
```

Il affiche

```
5#[push(a) push(3) plus push(b) plus]
```

Des programmes plus compliqués ont souvent besoin de plus d'accumulateurs. Dans certains grands programmes déclaratifs, nous avons utilisé jusquà dix accumulateurs. Le compilateur Aquarius Prolog a été écrit dans ce style [96, 94]. Certaines de ses procédures ont jusqu'à douze accumulateurs, ce qui veut dire 24 arguments supplémentaires ! Il est difficile d'utiliser autant d'arguments sans une assistance automatisée. Nous avons utilisé un préprocesseur DCG[6] étendu qui prend des déclarations d'accumulateurs et qui ajoute les arguments nécessaires [48].

Nous ne programmons plus dans ce style ; nous trouvons que la programmation avec l'état explicite est plus simple et plus efficace (voir chapitre 5). Il est raisonnable d'utiliser quelques accumulateurs dans un programme déclaratif ; en fait il est rare qu'un programme déclaratif n'en ait pas besoin. Par contre, l'utilisation d'un grand nombre d'accumulateurs est un signe qu'il faut mieux utiliser l'état explicite.

3.4.4 Les arbres

À côté des structures de données linéaires comme les listes, les arbres sont la structure de données récursive la plus importante dans la boîte à outils d'un programmeur. Un arbre est soit un nœud feuille soit un nœud non feuille qui contient un ou plusieurs arbres. Un nœud feuille s'appelle aussi un nœud externe ; un nœud non feuille s'appelle aussi un nœud interne. Nous nous intéressons aux arbres finis, qui ont un nombre fini de nœuds. Les nœuds peuvent porter des informations supplémentaires. Voici une définition d'un arbre :

⟨Tree⟩ ::= `leaf`
| `tree(`⟨Value⟩ ⟨Tree⟩$_1$ $\cdots$ ⟨Tree⟩$_n$`)`

La différence majeure entre une liste et un arbre réside dans la bifurcation : une liste a toujours une structure linéaire mais un arbre peut avoir une structure bifurquante. Une liste non vide a toujours un élément suivi par exactement une liste plus petite. Un arbre non vide a un nœud suivi par un nombre d'arbres plus petits. Ce nombre peut être tout nombre naturel : zéro pour les nœuds feuilles et tout nombre positif pour les nœuds non feuilles.

Il existe différentes sortes d'arbres, avec des branchements et des contenus différents. Par exemple, une liste est un arbre dans lequel les nœuds non feuilles ont toujours un unique sous-arbre (on peut l'appeler un arbre unaire). Dans un arbre binaire, les nœuds non feuilles ont exactement deux sous-arbres. Dans un arbre ternaire, ils ont exactement trois sous-arbres. Dans un arbre équilibré, tous les sous-arbres du même nœud ont la même taille (le même nombre de nœuds) ou approximativement la même taille.

6. DCG (« *Definite Clause Grammar* ») est une notation de grammaire pour Prolog qui est utilisée pour cacher l'enfilage explicite d'un accumulateur.

Chaque sorte d'arbre a sa propre classe d'algorithmes pour les construire, les traverser et en chercher des informations. Dans cette section nous nous limitons aux arbres binaires. Nous définissons des arbres binaires ordonnés et nous montrons comment insérer des informations, chercher des informations et enlever des informations de ces arbres.

a) Les arbres binaires ordonnés

Un arbre binaire ordonné ⟨OBTree⟩ est un arbre binaire dans lequel chaque nœud non feuille contient une paire de valeurs :

```
⟨OBTree⟩  ::=  leaf
             |  tree(⟨OValue⟩ ⟨Value⟩ ⟨OBTree⟩1 ⟨OBTree⟩2)
```

Chaque nœud non feuille contient les valeurs ⟨OValue⟩ et ⟨Value⟩. La première valeur ⟨OValue⟩ est un sous-type de ⟨Value⟩ qui est totalement ordonné, c'est-à-dire qui a des fonctions de comparaison booléennes. Par exemple, ⟨Int⟩ (le type des entiers) est une possibilité. La deuxième valeur ⟨Value⟩ est entraînée avec l'autre. Elle n'est soumise à aucune condition particulière.

La première valeur est la clé et la deuxième valeur contient les informations. Un arbre binaire est ordonné si pour chaque nœud non feuille, toutes les clés du premier sous-arbre sont plus petites que la clé du nœud, et toutes les clés du deuxième sous-arbre sont plus grandes que la clé du nœud.

b) L'enregistrement des informations dans les arbres

Un arbre binaire ordonné peut être utilisé comme un entrepôt d'informations si nous définissons trois opérations : recherche, insertion et retrait d'éléments.

Chercher des informations dans un arbre binaire ordonné veut dire vérifier si la clé est présente dans un des nœuds de l'arbre, et si c'est le cas, renvoyer les informations présentes dans ce nœud. Avec la condition d'ordre, l'algorithme de recherche peut éliminer la moitié des nœuds restants à chaque pas. Cette technique s'appelle la **recherche binaire**. Elle a besoin d'un nombre d'opérations proportionnel à la profondeur de l'arbre, la longueur du plus long chemin de la racine vers une feuille. Voici une routine qui implémente cette recherche :

```
fun {Lookup X T}
   case T of leaf then notfound
   [] tree(Y V T1 T2) then
      if X<Y then {Lookup X T1}
      elseif X>Y then {Lookup X T2}
      else found(V) end
   end
end
```

L'appel de `{Lookup X T}` renvoie `found(V)` si un nœud avec `X` est trouvé et `notfound` sinon. Une autre manière d'écrire `Lookup` est avec **`andthen`** dans l'instruction **`case`** :

```
fun {Lookup X T}
   case T of leaf then notfound
   [] tree(Y V T1 T2) andthen X==Y then found(V)
   [] tree(Y V T1 T2) andthen X<Y then {Lookup X T1}
   [] tree(Y V T1 T2) andthen X>Y then {Lookup X T2}
   end
end
```

Beaucoup de développeurs considèrent la deuxième manière plus lisible parce qu'elle est plus visuelle : elle donne des formes qui montrent à quoi ressemble l'arbre au lieu de donner des instructions pour le décomposer. En un mot, c'est plus déclaratif. Cela facilite la vérification de son exactitude. Il est plus facile de s'assurer qu'aucun cas n'a été oublié. Dans les algorithmes plus compliqués sur les arbres, la correspondance de formes avec **`andthen`** a un avantage définitif sur les instructions **`if`** explicites.

Pour insérer ou enlever des informations dans un arbre binaire ordonné, nous construisons un nouvel arbre qui est identique à l'original sauf qu'il a plus ou moins d'informations. Voici l'opération d'insertion :

```
fun {Insert X V T}
   case T of leaf then tree(X V leaf leaf)
   [] tree(Y W T1 T2) andthen X==Y then
                         tree(X V T1 T2)
   [] tree(Y W T1 T2) andthen X<Y then
                         tree(Y W {Insert X V T1} T2)
   [] tree(Y W T1 T2) andthen X>Y then
                         tree(Y W T1 {Insert X V T2})
   end
end
```

L'appel `{Insert X V T}` renvoie un nouvel arbre qui a la paire `(X V)` insérée au bon endroit. Si `T` contient déjà `X`, le nouvel arbre remplacera les vieilles informations par `V`.

c) Le retrait et la réorganisation de l'arbre

L'opération de retrait peut sembler surprenante. Voici une première tentative :

```
fun {Delete X T}
   case T of leaf then leaf
   [] tree(Y W T1 T2) andthen X==Y then leaf
   [] tree(Y W T1 T2) andthen X<Y then
                         tree(Y W {Delete X T1} T2)
   [] tree(Y W T1 T2) andthen X>Y then
                         tree(Y W T1 {Delete X T2})
   end
end
```

L'appel `{Delete X T}` doit renvoyer un nouvel arbre qui n'a pas de nœud avec la clé `X`. Si `T` ne contient pas `X`, `T` sera renvoyé sans changement. Le retrait semble simple, mais cette définition est fausse. Voyez-vous le problème ?

Il s'avère que `Delete` n'est pas aussi simple que `Lookup` ou `Insert`. L'erreur dans cette définition est quand `X==Y` tout le sous-arbre sera enlevé au lieu d'un seul nœud. Ce n'est correct que quand le sous-arbre est dégénéré, c'est-à-dire quand `T1` et `T2` sont tous les deux des nœuds feuilles. La correction n'est pas complètement évidente : quand `X==Y` nous devons réorganiser le sous-arbre pour qu'il ne contienne plus la clé `Y` mais reste un arbre binaire ordonné. Il y a deux cas, illustrés dans les figures 3.11 et 3.12.

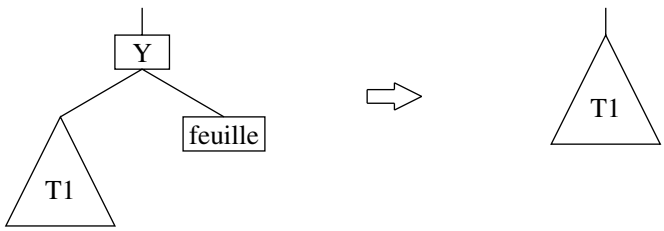


Figure 3.11 Le retrait du nœud `Y` quand un sous-arbre est une feuille (cas simple).

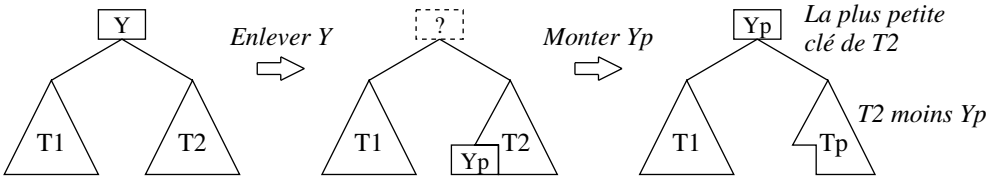


Figure 3.12 Le retrait du nœud `Y` quand aucun sous-arbre n'est une feuille (cas difficile).

La figure 3.11 est le cas simple, quand un sous-arbre est une feuille. L'arbre réorganisé est simplement l'autre sous-arbre. La figure 3.12 est le cas difficile, quand aucun sous-arbre n'est une feuille. Comment remplissons-nous le trou qui reste après le retrait de `Y` ? Une autre clé doit prendre la place de `Y`, « montant » de l'intérieur

d'un des sous-arbres. L'idée est de prendre la plus petite clé de `T2`, appelée `Yp`, et d'en faire la racine de l'arbre réorganisé. Les nœuds restants de `T2` font un sous-arbre plus petit, appelée `Tp`, qui est placé dans l'arbre réorganisé. Cela garantit que l'arbre réorganisé est toujours ordonné, puisque par construction toutes les clés de `T1` sont plus petites que `Yp`, qui est plus petite que toutes les clés de `Tp`.

Il est intéressant de voir ce qui se passe quand nous enlevons la racine à plusieurs reprises. Cela va « évider » l'arbre de l'intérieur, en enlevant de plus en plus la partie gauche de `T2`. Finalement, le sous-arbre de gauche de `T2` est enlevé complètement et le sous-arbre de droite prend sa place. En continuant, `T2` rétrécit de plus en plus, en passant par des étapes intermédiaires dans lesquelles il est toujours plus petit, tout en restant un arbre binaire ordonné. À un moment donné il disparaît complètement.

Pour implémenter la correction, nous utilisons une fonction `{RemoveSmallest T2}` qui renvoie la clé la plus petite de `T2`, sa valeur associée et un nouvel arbre qui n'a pas cette clé. Avec cette fonction nous pouvons écrire une version correcte de `Delete` :

```
fun {Delete X T}
   case T of leaf then leaf
   [] tree(Y W T1 T2) andthen X==Y then
      case {RemoveSmallest T2} of none then T1
      [] Yp#Vp#Tp then tree(Yp Vp T1 Tp) end
   [] tree(Y W T1 T2) andthen X<Y then
      tree(Y W {Delete X T1} T2)
   [] tree(Y W T1 T2) andthen X>Y then
      tree(Y W T1 {Delete X T2})
   end
end
```

La fonction `RemoveSmallest` renvoie soit un triple `Yp#Vp#Tp` soit l'atome `none`. Voici une définition récursive :

```
fun {RemoveSmallest T}
   case T of leaf then none
   [] tree(Y V T1 T2) then
      case {RemoveSmallest T1} of none then Y#V#T2
      [] Yp#Vp#Tp then Yp#Vp#tree(Y V Tp T2) end
   end
end
```

Une autre possibilité serait de prendre l'élément le plus grand de `T1` au lieu de l'élément le plus petit de `T2`. Le résultat final serait similaire.

La difficulté supplémentaire de `Delete` par rapport à `Insert` ou `Lookup` se produit souvent avec des algorithmes sur les arbres. Cela arrive parce que le fait d'être

ordonné est une condition globale sur l'arbre. Beaucoup d'arbres sont définis par des conditions globales. Les algorithmes pour ces arbres sont compliqués parce qu'ils doivent maintenir la condition globale. Le maintien d'une condition globale est un exemple d'un **calcul orienté but** (« *goal-oriented computation* ») souvent utilisé dans les techniques de l'intelligence artificielle. Les algorithmes sur les arbres sont plus compliqués que les algorithmes sur les listes parce que la récursion doit combiner les résultats de plusieurs problèmes plus petits au lieu d'un seul.

3.5 LES TYPES DE DONNÉES ABSTRAITS

Un type de données, ou simplement un type, est un ensemble de valeurs avec un ensemble d'opérations sur ces valeurs. Notre modèle déclaratif a un ensemble prédéfini de types, qui s'appellent les types de base (voir section 2.3). L'utilisateur peut aussi définir de nouveaux types. Nous disons qu'un type est abstrait s'il est complètement défini par l'ensemble de ses opérations, indépendamment de son implémentation. Le terme « type de données abstrait » est souvent abrégé en ADT. L'utilisation d'un ADT implique qu'il est possible de changer l'implémentation du type sans changer son utilisation. Nous regardons comment l'utilisateur peut définir de nouveaux ADT.

Voici un exemple d'un nouveau type de données abstrait, une pile ⟨Stack T⟩ avec des éléments de type T. Nous supposons que la pile a quatre opérations, avec les types suivants :

⟨**fun** {NewStack} : ⟨Stack T⟩⟩
⟨**fun** {Push ⟨Stack T⟩ T} : ⟨Stack T⟩⟩
⟨**fun** {Pop ⟨Stack T⟩ T} : ⟨Stack T⟩⟩
⟨**fun** {IsEmpty ⟨Stack T⟩} : ⟨Bool⟩⟩

Cet ensemble d'opérations et leurs types définit l'interface de l'ADT. Les opérations satisfont certaines lois, par exemple :

- `{IsEmpty {NewStack}}=`**`true`**. Une nouvelle pile est toujours vide.
- Pour toutes `E` et `S0`, `S1={Push S0 E}` et `S0={Pop S1 E}` sont vraies. Empilez un élément et puis dépilez un élément donne le même élément.
- Pour une `E` non liée, `{Pop {EmptyStack} E}` lève une erreur. Aucun élément ne peut être dépilé d'une pile vide.

Ces lois sont indépendantes de l'implémentation : toutes les implémentations doivent les satisfaire. Voici une implémentation de la pile qui satisfait les lois :

```
fun {NewStack} nil end
fun {Push S E} E|S end
fun {Pop S E} case S of X|S1 then E=X S1 end end
fun {IsEmpty S} S==nil end
```

Voici une autre implémentation qui satisfait les lois :

```
fun {NewStack} stackEmpty end
fun {Push S E} stack(E S) end
fun {Pop S E} case S of stack(X S1) then E=X S1 end end
fun {IsEmpty S} S==stackEmpty end
```

Tout programme qui utilise une pile fonctionnera avec les deux. C'est une conséquence du fait que la pile est abstraite.

3.5.1 Les types abstraits sécurisés

Un grand problème des deux implémentations ci-dessus est que la pile n'est pas sécurisée. La représentation interne des valeurs est visible pour les utilisateurs du type. Si les utilisateurs sont des programmeurs disciplinés, cela ne posera peut-être pas de problème. Mais ce n'est pas toujours le cas. Un utilisateur peut être tenté d'inspecter la représentation ou même de construire de nouvelles valeurs de la représentation.

Par exemple, un utilisateur du type pile peut utiliser `Length` pour voir combien d'éléments il y a sur la pile, si la pile est implémentée comme une liste. La tentation d'agir ainsi peut être très forte s'il n'y a pas d'autre manière de trouver la taille d'une pile. Une autre tentation est de bricoler le contenu d'une pile. Comme toute liste est aussi une valeur légale de pile, l'utilisateur peut construire de nouvelles valeurs de la pile, par exemple, en ajoutant ou en enlevant des éléments.

Tout utilisateur peut ajouter de nouvelles opérations sur les piles n'importe où dans le programme. L'implémentation de la pile est donc étalée sur tout le programme au lieu d'être limitée dans une petite partie. C'est une situation désastreuse, pour deux raisons :

- Le programme est bien plus difficile à maintenir. Par exemple, si nous voulions améliorer l'efficacité d'un dictionnaire en remplaçant une implémentation basée sur une liste par une implémentation basée sur un arbre, nous devrions parcourir tout le programme pour trouver les parties qui dépendent de l'implémentation basée sur une liste. Il y a aussi le problème du confinement de fautes : si le programme a des bugs dans une partie, cela peut contaminer les ADT, ce qui contamine encore d'autres parties du programme et ainsi de suite.
- Le programme est sensible aux ingérences malicieuses. C'est un problème plus subtil qui a un rapport avec la sécurité. Cela n'arrive pas avec des programmes écrits par des personnes qui se font confiance. Cela arrive plutôt avec des programmes ouverts. Un programme ouvert peut interagir avec d'autres programmes qui ne sont pas connus lors de son développement mais seulement lors de son

exécution. Que se passe-t-il si l'autre programme est malicieux et veut perturber l'exécution du programme ouvert ? À cause de l'évolution de l'Internet, le nombre de programmes ouverts est en train d'augmenter.[7]

Pour résoudre ces problèmes, nous allons protéger la représentation interne des valeurs de l'ADT contre toute interférence non autorisée. Dans les sections suivantes, nous construisons un ADT sécurisé. Nous utilisons une sorte de « clé » pour protéger les représentations internes quand elles sont en dehors de la frontière de l'ADT. Nous ajoutons cette clé comme un nouveau concept au modèle déclaratif et nous l'appelons un « nom ». Cette clé est un exemple d'un concept général de sécurité qui s'appelle une **capacité**. Plus loin, dans le chapitre 5, la section 5.4 continuera l'étude des ADT sécurisés en expliquant l'effet de l'état explicite sur la sécurité.

3.5.2 Le modèle déclaratif avec types sécurisés

Le modèle déclaratif ne nous permet pas de protéger la représentation interne des types. Pour y parvenir, nous devons étendre le modèle. Une manière de protéger les valeurs est d'ajouter une opération d'« emballage » avec une « clé ». La représentation interne est placée à l'intérieur d'une structure de données qui est inaccessible sauf pour ceux qui connaissent une valeur spéciale, la clé. Une abstraction qui connaît la clé peut créer de nouveaux emballages et regarder à l'intérieur des emballages existants faits avec la même clé. Le tableau 3.2 montre le nouveau langage noyau avec sa nouvelle opération.

$\langle s \rangle ::=$	
skip	Instruction vide
$\langle s \rangle_1\ \langle s \rangle_2$	Séquence d'instructions
local $\langle x \rangle$ **in** $\langle s \rangle$ **end**	Création de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Lien variable-variable
$\langle x \rangle = \langle v \rangle$	Création de valeur
if $\langle x \rangle$ **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**	Instruction conditionnelle
case $\langle x \rangle$ **of** $\langle$pattern$\rangle$ **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**	Correspondance de formes
$\{\langle x \rangle\ \langle y \rangle_1 \cdots \langle y \rangle_n\}$	Application de procédure
try $\langle s \rangle_1$ **catch** $\langle x \rangle$ **then** $\langle s \rangle_2$ **end**	Contexte d'exception
raise $\langle x \rangle$ **end**	Lève exception
`{NewName` $\langle x \rangle$ `}`	**Création de nom**

Tableau 3.2 Le langage noyau déclaratif avec types sécurisés.

7. Le système Mozart soutient les programmes ouverts avec une couche qui implémente la programmation répartie transparente [97].

Il y a un nouveau type de base qui s'appelle un nom ou une **valeur de nom**. Un nom est une constante semblable à un atome sauf qu'il a un jeu d'opérations plus restreint. En particulier, les noms n'ont pas de représentation textuelle : ils ne peuvent pas être imprimés ou tapés sur un clavier. À la différence des atomes, il n'est pas possible de convertir entre noms et chaînes de caractères. La seule manière de connaître un nom est que quelqu'un passe une référence au nom à l'intérieur du programme. Le type du nom a deux opérations :

Opération	Description
`{NewName}`	Renvoyez un nouveau nom
`N1==N2`	Comparez les noms `N1` et `N2`

Un nouveau nom est différent de tous les autres noms dans le système. Le lecteur attentif remarquera que `NewName` n'est pas déclaratif parce que l'appeler deux fois donne des résultats différents. La création de nouveaux noms est une opération à état. La garantie du caractère unique de chaque nouveau nom implique que `NewName` ait une mémoire interne. Quoi qu'il en soit, si nous utilisons `NewName` uniquement pour sécuriser des ADT déclaratifs, ce ne sera pas un problème. L'ADT sécurisé qui en résulte sera toujours déclaratif.

3.5.3 La création d'un type abstrait sécurisé

Pour sécuriser un type de données, il suffit de le mettre à l'intérieur d'une structure de données protégée par un nom. Par exemple, prenez la valeur `S` :

```
S=[a b c]
```

`S` est un état interne de la pile que nous avons définie auparavant. Nous pouvons le sécuriser comme ceci :

```
Key={NewName}
SS={Chunk.new w(Key:S)}
```

Nous créons d'abord un nouveau nom dans `Key`. Ensuite nous créons un « *chunk* » `SS` qui contient `S`, de telle manière que l'on peut extraire `S` seulement si on connaît `Key`. Un chunk est un enregistrement restreint qui n'a qu'une opération, l'opérateur de sélection « `.` » (voir la documentation de Mozart [23]). Nous disons que nous avons « emballé » la valeur `S` dans `SS`. Si on connaît `Key`, il est facile de retrouver `S` à partir de `SS` :

```
S=try SS.Key
  catch _ then raise error(unwrap(SS)) end end
```

Nous disons que cela « déballe » la valeur `S` de l'emballage `SS`. Si on ne connaît pas `Key`, le déballage sera impossible. Il n'y a aucune manière de connaître `Key` sauf si

on vous le donne explicitement dans le programme. Un appel de `SS` avec un argument erroné lèvera simplement une exception. Le **`try`** nous assure que la clé n'est pas divulguée à travers l'exception.

Nous pouvons définir une abstraction de données pour faire l'emballage et le déballage. L'abstraction définit deux opérations, `Wrap` (emballer) et `Unwrap` (déballer). Chaque opération est une fonction d'un argument. `Wrap` prend une valeur et renvoie une valeur protégée. `Unwrap` prend une valeur protégée et renvoie la valeur originale. Les opérations `Wrap` et `Unwrap` sont complémentaires. La seule manière de déballer une valeur emballée est d'utiliser l'opération de déballage correspondante. Nous pouvons définir une procédure `NewWrapper` qui renvoie de nouvelles paires `Wrap`/`Unwrap` :

```
proc {NewWrapper ?Wrap ?Unwrap}
Key={NewName} in
   fun {Wrap X} {Chunk.new w(Key:X)} end
   fun {Unwrap W}
      try W.Key
      catch _ then raise error(unwrap(W)) end end
   end
end
```

Pour une protection maximale, chaque ADT sécurisé que nous définissons doit utiliser sa propre paire `Wrap`/`Unwrap`. Ainsi chaque ADT est protégé contre les autres aussi bien que contre le reste du programme. Prenez la valeur `S` comme avant :

```
S=[a b c]
```

Nous la protégeons ainsi :

```
SS={Wrap S}
```

Nous retrouvons la valeur originale ainsi :

```
S={Unwrap SS}
```

3.5.4 Une pile sécurisée

Maintenant nous pouvons sécuriser la pile. L'idée est de déballer les valeurs qui entrent dans l'abstraction et d'emballer les valeurs qui en sortent. Pour faire une opération légale sur une valeur sécurisée du type, chaque opération déballe d'abord la valeur sécurisée, fait ensuite l'opération pour obtenir une nouvelle valeur, et enfin emballe la nouvelle valeur pour garantir la sécurité. Voici l'implémentation :

```
local Wrap Unwrap in
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap nil} end
   fun {Push S E} {Wrap E|{Unwrap S}} end
   fun {Pop S E}
      case {Unwrap S} of X|S1 then E=X {Wrap S1} end
   end
   fun {IsEmpty S} {Unwrap S}==nil end
end
```

La figure 3.13 montre l'opération `Pop`. La boîte avec un trou de serrure représente une valeur protégée. La clé représente le nom, qui est utilisé par `Wrap` pour fermer la boîte (emballer) et par `Unwrap` pour ouvrir la boîte (déballer). La portée lexicale garantit que l'emballage et le déballage ne sont possibles qu'à l'intérieur de l'implémentation de la pile. Les identificateurs `Wrap` et `Unwrap` ne sont visibles qu'à l'intérieur de l'instruction **`local`**. En dehors de cette portée, ils sont cachés. Parce que `Unwrap` est cachée, il n'y a aucune manière de regarder à l'intérieur d'une valeur de pile. Parce que `Wrap` est cachée, il n'y a aucune manière de bricoler les valeurs de pile.

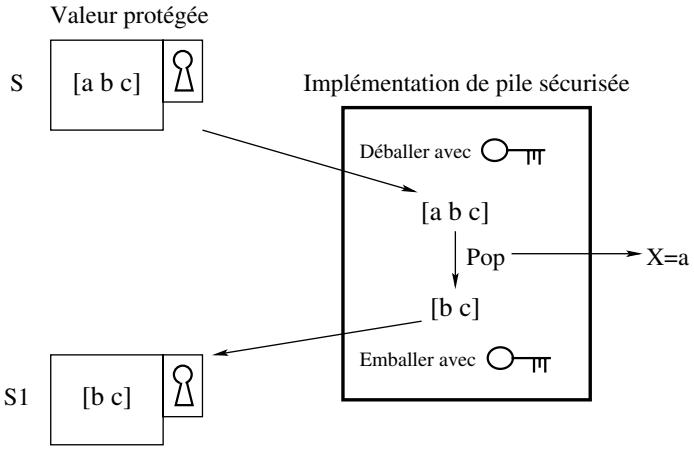


Figure 3.13 Faire `S1={Pop S X}` avec une pile sécurisée.

3.6 L'EFFICACITÉ EN TEMPS ET EN ESPACE

La programmation déclarative est toujours de la programmation ; même avec ses propriétés mathématiques fortes elle donne toujours de vrais programmes qui tournent sur de vrais ordinateurs. Il est donc important de réfléchir sur l'efficacité de ses calculs. Il y a deux côtés à l'efficacité : le temps d'exécution (par exemple, en secondes) et

l'utilisation de mémoire (par exemple, en octets). Nous montrons comment calculer les deux.

3.6.1 Le temps d'exécution

En utilisant le langage noyau et sa sémantique, nous pouvons calculer le temps d'exécution à un facteur constant près. Par exemple, pour un algorithme de tri par fusion nous pouvons dire que le temps d'exécution est proportionnel à $n \log n$, avec une liste d'entrée de longueur n. La complexité asymptotique en temps d'un algorithme est la meilleure borne supérieure de son temps d'exécution en fonction de la taille de l'entrée, à un facteur constant près. Cette borne est aussi appelée la complexité en temps dans le pire cas.

Pour trouver le facteur constant, il est nécessaire de mesurer l'exécution du programme. Calculer le facteur constant sans exécuter le programme est extrêmement difficile. C'est parce que les ordinateurs modernes ont une structure logicielle et matérielle complexe qui introduit beaucoup d'imprévisibilité dans le temps d'exécution : ils font de la gestion de mémoire (voir section 2.5), ils ont des systèmes de mémoire complexes (avec la mémoire virtuelle et plusieurs niveaux de mémoire cache), ils ont une architecture pipeline et super-scalaire (plusieurs instructions s'exécutent simultanément ; le temps d'exécution d'une instruction dépend souvent des autres instructions présentes) et le système d'exploitation fait des changements de contexte à des moments imprévisibles. Cette imprévisibilité améliore la performance moyenne au prix d'une augmentation de sa variabilité. Pour plus d'informations sur les mesures de performance et ses dangers, nous recommandons [43].

La notation grand O

Nous donnons le temps d'exécution du programme avec la notation « grand O ». Cette notation nous permet de parler du temps d'exécution sans avoir à préciser le facteur constant. Soit $T(n)$ une fonction qui donne le temps d'exécution d'un programme, mesuré en fonction de la taille de l'entrée n. Soit $f(n)$ une autre fonction définie sur les entiers non négatifs. Nous disons alors que $T(n)$ est de l'ordre de $f(n)$, noté $O(f(n))$, si $T(n) \leqslant c \cdot f(n)$ pour une constante positive c, pour tout n sauf certaines petites valeurs $n \leqslant n_0$. En d'autres termes, quand n grandit il y a un point n_0 au-delà duquel $T(n)$ ne devient jamais plus grand que $c \cdot f(n)$.

Parfois on écrit $T(n) = O(f(n))$, avec une égalité « = ». Attention ! Cette utilisation de l'égalité est un abus de notation, parce qu'il n'y a pas d'égalité. Si $g(n) = O(f(n))$ et $h(n) = O(f(n))$, il n'est pas vrai que $g(n) = h(n)$. Une meilleure manière de comprendre la notation grand O est avec les ensembles et leurs membres : $O(f(n))$ est un ensemble de fonctions et $T(n) = O(f(n))$ veut simplement dire que $T(n)$ est un membre de l'ensemble.

Calculer le temps d'exécution

Nous utilisons le langage noyau comme un guide. Chaque instruction noyau a un temps d'exécution bien défini, qui peut être une fonction de la taille de ses arguments. Supposons un programme qui contient les p fonctions `F1`, ..., `Fp`. Nous voudrions calculer les p fonctions $T_{\texttt{F1}}, \ldots, T_{\texttt{Fp}}$. On peut le faire en trois étapes :

1. Traduisez le programme en langage noyau.
2. Utilisez les temps d'exécution des instructions noyau pour faire un ensemble d'équations qui contient $T_{\texttt{F1}}, \ldots, T_{\texttt{Fp}}$. Nous appelons ces équations des **équations de récurrence** parce qu'elles définissent le résultat pour n en utilisant les résultats pour des valeurs plus petites que n.
3. Résolvez les équations de récurrence pour $T_{\texttt{F1}}, \ldots, T_{\texttt{Fp}}$.

Le tableau 3.3 donne le temps d'exécution $T(s)$ pour chaque instruction noyau $\langle \mathsf{s} \rangle$. Dans ce tableau, s est un entier et les arguments $y_i = E(\langle \mathsf{y} \rangle_i)$ avec $1 \leqslant i \leqslant n$, pour l'environnement approprié E. Chaque instance de k est une autre constante réelle et positive. La fonction $I_x(\{y_1, \ldots, y_n\})$ renvoie le sous-ensemble des arguments d'une procédure qui est utilisé comme entrées.[8] La fonction $\text{size}_x(\{y_1, \ldots, y_k\})$ est la « taille » des entrées pour la procédure x. Nous avons la liberté de définir la taille comme nous le voulons ; si elle est mal définie, les équations de récurrence n'auront pas de solution. Pour les instructions $\langle \mathsf{x} \rangle = \langle \mathsf{y} \rangle$ et $\langle \mathsf{x} \rangle = \langle \mathsf{v} \rangle$ il y a un cas rare pour lequel elles prennent plus de temps qu'un temps constant, à savoir quand les deux arguments sont liés à des valeurs partielles de grande taille. Dans ce cas, le temps est proportionnel à la taille de la partie commune des deux valeurs partielles.

Un exemple : la fonction `Append`

Voici un exemple simple pour illustrer la méthode. Prenons la fonction `Append` :

```
fun {Append Xs Ys}
   case Xs of nil then Ys
   [] X|Xr then X|{Append Xr Ys} end
end
```

Elle a la traduction suivante en langage noyau (légèrement simplifiée) :

```
proc {Append Xs Ys ?Zs}
   case Xs of nil then Zs=Ys
   [] X|Xr then Zr in Zs=X|Zr {Append Xr Ys Zr} end
end
```

8. Cela peut changer d'appel en appel, par exemple quand la même procédure est utilisée pour accomplir des tâches différentes lors des appels différents.

Avec le tableau 3.3, nous obtenons l'équation de récurrence suivante pour l'appel récursif :

$$T_{\texttt{Append}}(\text{size}(I(\{\texttt{Xs}, \texttt{Ys}, \texttt{Zs}\}))) = \\ k_1 + \max(k_2, k_3 + T_{\texttt{Append}}(\text{size}(I(\{\texttt{Xr}, \texttt{Ys}, \texttt{Zr}\}))))$$

(Les indices pour size et I ne sont pas nécessaires ici.) Pour simplifier, utilisons $I(\{\texttt{Xs}, \texttt{Ys}, \texttt{Zs}\}) = \{\texttt{Xs}\}$ et supposons que $\text{size}(\{\texttt{Xs}\}) = n$, où n est la longueur de `Xs`. Cela donne

$$T_{\texttt{Append}}(n) \quad = \quad k_1 + \max(k_2, k_3 + T_{\texttt{Append}}(n-1))$$

Avec un peu de simplification :

$$T_{\texttt{Append}}(n) \quad = \quad k_4 + T_{\texttt{Append}}(n-1)$$

Nous traitons le cas de base en prenant une valeur particulière de `Xs` pour laquelle nous pouvons calculer le résultat directement. Prenons `Xs=nil`. Cela donne

$$T_{\texttt{Append}}(0) \quad = \quad k_5$$

La solution des deux équations est

$$T_{\texttt{Append}}(n) \quad = \quad k_4 \cdot n + k_5$$

$T_{\texttt{Append}}(n)$ est donc $O(n)$.

⟨s⟩ : :=	
skip	k
⟨x⟩$_1$=⟨x⟩$_2$	k
⟨x⟩=⟨v⟩	k
⟨s⟩$_1$ ⟨s⟩$_2$	$T(s_1) + T(s_2)$
local ⟨x⟩ **in** ⟨s⟩ **end**	$k + T(s)$
proc {⟨x⟩ ⟨y⟩$_1$ ⋯ ⟨y⟩$_n$} ⟨s⟩ **end**	k
if ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	$k + \max(T(s_1), T(s_2))$
case ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	$k + \max(T(s_1), T(s_2))$
{⟨x⟩ ⟨y⟩$_1$ ⋯ ⟨y⟩$_n$}	$T_x(\text{size}_x(I_x(\{y_1, \ldots, y_n\})))$

Tableau 3.3 Les temps d'exécution des instructions noyau.

Les équations de récurrence

Avant de regarder d'autres exemples, examinons de plus près les équations de récurrence. Une équation de récurrence a l'une des deux formes suivantes :

- Une équation qui définit une fonction $T(n)$ en termes de $T(m_1), \ldots, T(m_k)$, où $m_1, \ldots, m_k < n$.
- Une équation qui donne la valeur de $T(n)$ pour certaines valeurs de n, comme $T(0)$ or $T(1)$.

Dans les calculs des temps d'exécution, il y a certaines formes d'équations de récurrence qui apparaissent souvent. Le tableau 3.4 montre certaines de ces équations et leurs solutions. Le tableau suppose que les k_i sont des constantes non nulles. Il y a beaucoup de techniques pour calculer ces solutions. Nous en verrons quelques-unes dans les exemples à venir. La boîte explique deux des plus utiles.

Équation	Solution
$T(n) = k + T(n-1)$	$O(n)$
$T(n) = k_1 + k_2.n + T(n-1)$	$O(n^2)$
$T(n) = k + T(n/2)$	$O(\log n)$
$T(n) = k_1 + k_2.n + T(n/2)$	$O(n)$
$T(n) = k + 2.T(n/2)$	$O(n)$
$T(n) = k_1 + k_2.n + 2.T(n/2)$	$O(n \log n)$
$T(n) = k_1.n^{k_2} + k_3.T(n-1)$	$O({k_3}^n)$ (si $k_3 > 1$)

Tableau 3.4 Quelques équations de récurrence courantes et leurs solutions.

La résolution d'équations de récurrence

Les techniques suivantes sont souvent utiles :

- Une technique simple qui fonctionne presque toujours en pratique. D'abord, obtenez des nombres exacts pour quelques petites entrées (par exemple : $T(0) = k$, $T(1) = k + 3$, $T(2) = k + 6$). Ensuite, devinez la forme du résultat (par exemple : $T(n) = an + b$, pour des valeurs inconnues de a et b). Enfin, substituez cette forme dans les équations. Dans notre exemple, cela donne $b = k$ et $(an + b) = 3 + (a \cdot (n-1) + b)$. Donc $a = 3$, ce qui donne le résultat final $T(n) = 3n + k$. Cette technique fonctionnera si la forme devinée est correcte.
- Une technique bien plus puissante, les fonctions génératrices, permet souvent d'obtenir une formule exacte ou asymptotique sans la nécessité de deviner la forme. Elle nécessite certaines connaissances des séries infinies et de l'analyse, mais pas plus que dans un premier cours universitaire. Voir Knuth [50] et Wilf [98] pour de bonnes introductions aux fonctions génératrices.

Un exemple : la fonction `FastPascal`

Dans le chapitre 1, nous avons défini la fonction `FastPascal` et nous avons prétendu que `{FastPascal N}` est $O(n^2)$. Nous faisons maintenant une dérivation plus rigoureuse. Voici la définition :

```
fun {FastPascal N}
   if N==1 then [1] else L in
      L={FastPascal N-1}
      {AddList {ShiftLeft L} {ShiftRight L}} end
end
```

Nous pouvons obtenir les équations directement à partir de cette définition, sans traduire les fonctions en procédures. En inspectant la définition, il est facile de voir que `ShiftRight` est $O(1)$, un temps constant. Avec un raisonnement semblable que pour `Append`, nous calculons que `AddList` et `ShiftLeft` sont $O(n)$ où n est la longueur de la liste `L`. Cela nous donne l'équation de récurrence suivante pour l'appel récursif :

$$T_{\texttt{FastPascal}}(n) = k_1 + \max(k_2, k_3 + T_{\texttt{FastPascal}}(n-1) + k_4 \cdot n)$$

où n est la valeur de l'argument `N`. Avec un peu de simplification, nous obtenons

$$T_{\texttt{FastPascal}}(n) = k_5 + k_4 \cdot n + T_{\texttt{FastPascal}}(n-1)$$

Dans le cas de base, nous choisissons `N`=1. Cela donne

$$T_{\texttt{FastPascal}}(1) = k_6$$

Pour résoudre ces deux équations, nous « devinons » que la solution est de la forme :

$$T_{\texttt{FastPascal}}(n) = a \cdot n^2 + b \cdot n + c$$

Cette estimation vient d'un argument intuitif comme celui du chapitre 1. Nous insérons cette forme dans les deux équations. Si nous trouvons des solutions pour a, b et c, nous pourrons conclure que notre supposition était correcte. Nous obtenons les trois équations en a, b et c :

$$\begin{aligned} k_4 - 2a &= 0 \\ k_5 + a - b &= 0 \\ a + b + c - k_6 &= 0 \end{aligned}$$

Il n'est pas nécessaire de résoudre ce système complètement ; il suffit de vérifier que $a \neq 0$.[9] Donc $T_{\texttt{FastPascal}}(n)$ est $O(n^2)$.

9. Si nous devinons $a \cdot n^2 + b \cdot n + c$ et la vraie solution a la forme $b \cdot n + c$, nous aurons $a = 0$.

Un exemple : la fonction `MergeSort`

Nous avons vu un algorithme de tri par fusion (« *mergesort* »). Calculons le temps d'exécution de cet algorithme. Voici la fonction principale :

```
fun {MergeSort Xs}
   case Xs of nil then nil
   [] [X] then [X]
   else Ys Zs in
      {Split Xs Ys Zs}
      {Merge {MergeSort Ys} {MergeSort Zs}}
   end
end
```

Soit $T(n)$ le temps d'exécution de `{MergeSort Xs}`, où n est la longueur de `Xs`. Supposons que `Split` et `Merge` sont $O(n)$ dans la longueur de leurs entrées. Nous savons que `Split` renvoie deux listes avec longueurs $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$, Avec la définition de `MergeSort`, cela nous permet de définir les équations de récurrence suivantes :

$$\begin{aligned} T(0) &= k_1 \\ T(1) &= k_2 \\ T(n) &= k_3 + k_4 n + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) \text{ if } n \geqslant 2 \end{aligned}$$

Ces équations utilisent les fonctions plafond (« *ceiling* ») et plancher (« *floor* »), dont la manipulation est un peu délicate. Pour nous en débarrasser, supposons que n est une puissance de 2, donc $n = 2^k$ pour un entier positif k. Les équations deviennent alors :

$$\begin{aligned} T(0) &= k_1 \\ T(1) &= k_2 \\ T(n) &= k_3 + k_4 n + 2T(n/2) \text{ if } n \geqslant 2 \end{aligned}$$

L'expansion de la dernière équation donne (avec $L(n) = k_3 + k_4 n$) :

$$T(n) = \overbrace{L(n) + 2L(n/2) + 4L(n/4) + \cdots + (n/2)L(2)}^{k} + 2T(1)$$

Le remplacement de $L(n)$ et $T(1)$ par leurs valeurs donne

$$T(n) = \overbrace{(k_4 n + k_3) + (k_4 n + 2k_3) + (k_4 n + 4k_3) + \cdots + (k_4 n + (n/2)k_3)}^{k} + k_2$$

Simplifier la somme donne

$$T(n) = k_4 kn + (n-1)k_3 + k_2$$

Nous concluons que $T(n) = O(n \log n)$. Pour des valeurs de n qui ne sont pas des puissances de 2, nous utilisons le fait, qui est facile à prouver, que $n \leqslant m \Rightarrow T(n) \leqslant T(m)$ pour montrer que la borne grand O est toujours valable. Cette borne est indépendante du contenu de la liste d'entrée. La borne $O(n \log n)$ est donc aussi une borne pour le pire cas.

3.6.2 L'utilisation de mémoire

L'utilisation de mémoire n'est pas un seul nombre comme le temps d'exécution. Il y a deux concepts très différents :

- La taille instantanée de mémoire active $m_a(t)$, en mots. Ce nombre indique la quantité de mémoire nécessaire par le programme pour continuer son exécution. Un nombre apparenté est la taille maximale de la mémoire active, $M_a(t) = \max_{0 \leqslant u \leqslant t} m_a(u)$. Ce nombre est utile pour calculer la quantité de mémoire physique nécessaire dans l'ordinateur pour exécuter le programme.
- La consommation instantanée de mémoire $m_c(t)$, en mots par seconde. Ce nombre indique le taux d'allocation de mémoire du programme pendant son exécution. Une grande valeur veut dire qu'il y a plus de travail en gestion de mémoire : le ramasse-miettes sera exécuté plus souvent, ce qui augmentera le temps d'exécution. Un nombre apparenté est la consommation totale de mémoire, $M_c(t) = \int_0^t m_c(u)du$, qui est une mesure de la quantité totale de travail qui doit être faite en gestion de mémoire pour exécuter le programme.

Il ne faut pas confondre ces deux nombres. Le premier est bien plus important. Un programme peut allouer de la mémoire très lentement (1 Ko/s) et avoir tout de même une grande mémoire active (100 Mo) ; par exemple une grande base de données qui est gardée en mémoire vive et qui traite des requêtes simples. Le contraire est possible aussi. Un programme peut consommer de la mémoire à un taux élevé (100 Mo/s) mais avoir une petite mémoire active (10 Ko) ; par exemple un algorithme de simulation qui s'exécute dans le modèle déclaratif.[10]

La taille instantanée de mémoire active

La taille de mémoire active peut être calculée à tout moment pendant l'exécution en suivant toutes les références de la pile sémantique en mémoire et en faisant la somme des tailles de toutes les valeurs partielles accessibles. Elle est approximativement égale à la taille de toutes les structures de données dont le programme a besoin pendant son exécution.

10. À cause de ce comportement, le modèle déclaratif n'est pas conseillé pour l'exécution des simulations sauf si son ramasse-miettes est excellent !

La consommation totale de mémoire

La consommation totale de mémoire peut être calculée avec une technique similaire à celle utilisée pour le temps d'exécution. Chaque opération du langage noyau a une consommation de mémoire bien définie. Le tableau 3.5 donne la consommation de mémoire $M(s)$ pour chaque instruction noyau $\langle \mathsf{s} \rangle$. Avec ce tableau, on peut établir des équations de récurrence pour le programme, à partir desquelles la consommation totale de mémoire du programme peut être calculée en fonction de la taille de l'entrée. À ce nombre il faut ajouter la consommation de mémoire de la pile sémantique. Pour l'instruction $\langle \mathsf{x} \rangle = \langle \mathsf{v} \rangle$ il y a un cas rare pour lequel la consommation de mémoire est plus petite que memsize(v), à savoir quand $\langle \mathsf{x} \rangle$ est partiellement instanciée. Dans ce cas, il n'y a que la mémoire des nouvelles entités qui doit être comptée. La fonction memsize(v) est définie selon le type et la valeur de v :

- Pour un entier : 0 pour des petits entiers, sinon proportionnel au nombre de chiffres de l'entier. Calculez le nombre de bits nécessaire pour représenter l'entier en complément à 2. Si ce nombre est plus petit que 28, alors 0. Sinon divisez par 32 et faites l'arrondi à l'entier supérieur.
- Pour un flottant : 2.
- Pour une paire de liste : 2.
- Pour un tuple ou un enregistrement : $1 + n$, où $n = \text{length}(\text{arity}(v))$.
- Pour une valeur procédurale : $k + n$, où n est le nombre de références externes du corps de la procédure et k est une constante qui dépend de l'implémentation.

$\langle \mathsf{s} \rangle$: :=	
skip	0
$\langle \mathsf{x} \rangle_1 = \langle \mathsf{x} \rangle_2$	0
$\langle \mathsf{x} \rangle = \langle \mathsf{v} \rangle$	memsize(v)
$\langle \mathsf{s} \rangle_1\ \langle \mathsf{s} \rangle_2$	$M(s_1) + M(s_2)$
local $\langle \mathsf{x} \rangle$ **in** $\langle \mathsf{s} \rangle$ **end**	$1 + M(s)$
if $\langle \mathsf{x} \rangle$ **then** $\langle \mathsf{s} \rangle_1$ **else** $\langle \mathsf{s} \rangle_2$ **end**	$\max(M(s_1), M(s_2))$
case $\langle \mathsf{x} \rangle$ **of** $\langle \text{pattern} \rangle$ **then** $\langle \mathsf{s} \rangle_1$ **else** $\langle \mathsf{s} \rangle_2$ **end**	$\max(M(s_1), M(s_2))$
$\{ \langle \mathsf{x} \rangle\ \langle \mathsf{y} \rangle_1 \cdots \langle \mathsf{y} \rangle_n \}$	$M_x(\text{size}_x(I_x(\{y_1, \ldots, y_n\})))$

Tableau 3.5 La consommation de mémoire des instructions noyau.

Tous les nombres sont calculés en multiples d'un mot de 32 bits et sont corrects pour Mozart 1.3.0. Pour les valeurs imbriquées, prenez la somme de toutes les valeurs. Pour les enregistrements et valeurs procédurales, il y a un coût supplémentaire qui est payé une fois. Pour chaque arité distincte le coût supplémentaire est approximativement

proportionnel à n. C'est parce que l'arité est stockée une fois dans une table de symboles. Pour chaque procédure distincte dans le code source, le coût additionnel dépend de la taille du code machine, qui est approximativement proportionnelle au nombre total d'instructions et d'identificateurs dans le corps de la procédure. Dans la plupart des cas, ces coûts supplémentaires ajoutent une constante à la consommation totale de mémoire ; pour le calcul on peut généralement les ignorer.

3.6.3 La complexité amortie

Il arrive parfois qu'une opération ait une complexité trop élevée mais qu'une série d'opérations ait une complexité acceptable. Par exemple, on peut implémenter les files ainsi. Une opération individuelle d'insertion ou de retrait a une complexité $O(n)$, ce qui est trop cher, mais une série de n opérations a aussi une complexité de $O(n)$, ce qui est acceptable. En fait la plupart des opérations sont $O(1)$ mais de temps en temps il y a une opération $O(n)$. Comme les opérations chères sont peu fréquentes, elles n'augmentent pas la complexité de la série. En général, si une série de n opérations a une temps d'exécution $O(f(n))$, nous disons qu'une opération individuelle a une complexité amortie $O(f(n)/n)$.

La complexité amortie versus la complexité au pire

Dans la plupart des domaines d'application, il suffit d'avoir une complexité amortie acceptable. Mais il y a trois domaines qui ont besoin de garanties sur le temps d'exécution des opérations individuelles. Ce sont les systèmes temps réel dur, les systèmes parallèles et les systèmes interactifs à haute performance.

Un système temps réel dur doit satisfaire des échéances strictes sur la terminaison des calculs. Manquer une telle échéance peut avoir des conséquences graves, y compris des pertes humaines. De tels systèmes existent, par exemple les stimulateurs cardiaques et les dispositifs de sécurité ferroviaires (pour éviter les collisions des trains).

Un système parallèle exécute plusieurs calculs simultanément pour augmenter la vitesse du calcul global. Souvent le calcul global ne peut avancer qu'après la terminaison de tous les calculs simultanés. Si un de ces calculs prend plus de temps, il fera ralentir le calcul global.

Un système interactif, comme un jeu sur ordinateur, doit avoir un temps de réaction qui ne varie pas trop. Par exemple, si un jeu multijoueur a une réaction retardée pour un des joueurs, la satisfaction de ce joueur sera de beaucoup réduite.

La méthode du banquier et la méthode du physicien

Le calcul de la complexité amortie est un peu plus difficile que le calcul de la complexité au pire. Il y a essentiellement deux méthodes, appelées la méthode du banquier et la méthode du physicien.

La méthode du banquier compte des crédits, où un « crédit » représente une unité de temps d'exécution ou d'espace de mémoire. Chaque opération met de côté des crédits. Une opération chère est autorisée quand il y a assez de crédits disponibles pour payer son exécution.

La méthode du physicien est basée sur la découverte d'une fonction de potentiel. On peut la considérer comme une sorte d'altitude par rapport au niveau de la mer. Chaque opération change le potentiel, c'est-à-dire qu'elle monte ou descend un peu. Le coût de chaque opération est son changement en potentiel : de combien elle monte ou descend. La complexité totale est une fonction de la différence entre le potentiel initial et final. Si elle reste petite, de grandes variations seront autorisées entre les deux.

Pour plus d'informations sur ces méthodes et beaucoup d'exemples de leur utilisation avec des algorithmes déclaratifs, nous recommandons le livre de Okasaki [72].

3.6.4 Des considérations sur la performance

Depuis le début de l'ère de l'ordinateur dans les années 1940, les coûts du temps et de l'espace ont diminué à un taux exponentiel : un facteur multiplicatif d'amélioration chaque année. Ils sont actuellement très bon marché, en termes absolus et en perception : un ordinateur personnel actuel a au moins 512 Mo de mémoire vive, 120 Go de mémoire persistante sur disque dur et une fréquence d'horloge de 3 GHz. Il a une performance maximale soutenue d'environ un milliard d'instructions par seconde, où chaque instruction peut faire une opération complète de 64 bits, y compris en virgule flottante. Il est nettement plus rapide qu'un super-ordinateur Cray-1, l'ordinateur le plus rapide au monde en 1975. Un super-ordinateur est défini comme un des ordinateurs les plus rapides à un moment donné. Le premier Cray-1 avait une fréquence d'horloge de 80 MHz et pouvait exécuter par cycle plusieurs opérations à virgule flottante de 64 bits [83]. À prix constant, la performance d'un ordinateur personnel s'améliore à un taux exponentiel (doublant environ tous les deux ans), et on peut prévoir que cela continuera au moins pendant dix ans. C'est une conséquence de la loi de Moore sur la densité des transistors.

La loi de Moore

La loi de Moore dit que la densité des circuits intégrés double environ tous les 18 mois. Ce phénomène a été observé pour la première fois par Gordon Moore vers 1965 et se vérifie jusqu'à ce jour. Les experts pensent que la loi peut ralentir mais qu'elle se maintiendra substantiellement pendant encore au moins 10 ans. Une interprétation

fausse mais courante de la loi originale est que la performance doublera environ tous les deux ans. Cette interprétation semble se vérifier aussi.[11]

La loi de Moore ne regarde qu'une petite période de temps par rapport au temps pendant lequel des calculs ont été faits par machine. Depuis le 19ème siècle, il y a eu au moins cinq technologies pour le calcul par machine, y compris la mécanique, l'électromécanique, les tubes à vide, les transistors individuels et les circuits intégrés. Pendant cette période plus longue, il est apparent que la croissance de la puissance de calcul à toujours été exponentielle. Selon Raymond Kurzweil, qui a étudié cette croissance, la prochaine technologie sera le calcul moléculaire en trois dimensions [57].[12]

À cause de cette situation, la performance n'est généralement pas un problème critique. Si votre problème est soluble en pratique, c'est-à-dire que l'on connaît un algorithme efficace pour le résoudre, alors si vous utilisez de bonnes techniques de conception algorithmique, le temps et l'espace effectifs utilisés par l'algorithme seront presque toujours acceptables. En d'autres termes, si la complexité asymptotique du programme est raisonnable, le facteur constant ne sera presque jamais critique. C'est vrai même pour la plupart des applications multimédia (qui utilisent la vidéo et l'audio) à cause des excellentes bibliothèques qui existent.

Les problèmes insolubles en pratique

Il existe des problèmes qui ne sont pas solubles en pratique. Il y a beaucoup de problèmes qui sont chers en ressources calculatoires, comme dans les domaines de l'optimisation combinatoire, la recherche opérationnelle, la simulation et le calcul scientifique, l'apprentissage par ordinateur, l'infographie et la reconnaissance de la parole et de la vision. Certains problèmes sont chers simplement parce qu'ils ont beaucoup de travail à faire. Par exemple, les jeux avec un graphisme réaliste et les effets spéciaux dans les films sont par définition toujours à la frontière ce qui est possible. D'autres problèmes sont chers pour des raisons plus fondamentales. Par exemple, les **problèmes NP-complets**. Ces problèmes sont dans la classe NP, c'est-à-dire qu'il est simple de vérifier une solution si on a un candidat.[13] Mais trouver une solution peut être bien plus difficile. Un exemple simple est le problème de satisfaisabilité des circuits digitaux. Soit un circuit digital combinatoire fait avec des portes `Et`, `Ou` et `Non` : existe-t-il un ensemble d'entrées qui rend la sortie vraie ? Ce problème est NP-complet [17]. Un problème NP-complet est un problème NP avec la particularité que si on peut le résoudre en temps polynomial, alors on pourra résoudre

11. Par contre, l'augmentation de la fréquence horloge semble avoir nettement ralenti depuis quelques années.

12. Kurzweil prétend que le taux de croissance est en train d'augmenter et mènera à une « singularité » quelque part au milieu du XXI^{e} siècle.

13. NP veut dire « en temps non-déterministe polynomial ».

tous les problèmes NP en temps polynomial. Beaucoup de chercheurs en informatique ont essayé pendant des décennies de trouver une solution en temps polynomial aux problèmes NP-complets, et aucun n'a réussi. La plupart des chercheurs soupçonne donc que les problèmes NP-complets ne peuvent pas être résolus en temps polynomial. Nous ne parlerons plus des problèmes qui ont besoin d'une grande puissance de calcul. Comme notre but est d'expliquer la programmation, nous nous limiterons aux problèmes solubles en pratique.

L'optimisation

Dans certains cas, la performance d'un problème peut être insuffisante même si le problème est théoriquement soluble en pratique. Il faut alors réécrire le programme pour améliorer sa performance. Réécrire un programme pour améliorer une de ses caractéristiques s'appelle l'**optimisation**, mais le programme n'est jamais optimal dans un sens mathématique. Généralement, on peut améliorer le programme jusqu'à un certain point, au-delà duquel il devient rapidement de plus en plus compliqué pour des améliorations de plus en plus petites. L'optimisation ne doit donc pas être faite sans nécessité. L'optimisation prématurée est la source de tous les maux.[14] Ce principe ne libère pas le programmeur de la responsabilité de faire une bonne conception du système ; il s'agit plutôt de ne pas perdre son temps à optimiser à petite échelle.

L'optimisation a un bon et un mauvais côté. Le bon côté est que le temps d'exécution de la plupart des applications est largement déterminé par une toute petite partie du texte du programme. L'optimisation de la performance, si nécessaire, peut donc presque toujours être faite en réécrivant cette petite partie (parfois quelques lignes suffisent). Le mauvais côté est qu'il n'est pas évident, même pour des programmeurs expérimentés, de savoir a priori où se trouve cette partie. La partie peut être identifiée en exécutant l'application, mais seulement s'il y a un problème de performance. S'il n'y a pas de problème, aucune optimisation de la performance ne doit être faite. La meilleure technique pour identifier la partie est de profiler l'application : instrumenter l'application pour mesurer ses caractéristiques à l'exécution.

3.7 LES BESOINS NON DÉCLARATIFS

La programmation déclarative, à cause de sa vue purement fonctionnelle de la programmation, est un peu détachée du monde réel, dans lequel les entités ont de la mémoire (l'état) et peuvent évoluer de façon indépendante et proactive (la concurrence). Pour connecter un programme déclaratif au monde réel, il faut quelques opérations non-déclaratives. Cette section présente deux classes de ces opérations : l'entrée/sortie

14. Une phrase célèbre de C.A.R. Hoare : « *premature optimization is the root of all evil* ».

des fichiers et les interfaces graphiques. Une troisième classe, la compilation des applications, est expliquée dans la section 3.8.

Les opérations de cette section sont regroupées dans des modules. Un module est un enregistrement qui contient des opérations apparentées. Par exemple, le module `List` contient beaucoup d'opérations sur les listes, comme `List.append` et `List.member` (qui peuvent être référencées comme `Append` et `Member`). Cette section introduit les deux modules `File` (pour les fichiers texte) et `QTk` (pour les interfaces graphiques). Certains modules de Mozart (modules de base appelés « *Base modules* » [23] et modules système appelés « *System modules* » [22]) sont immédiatement utilisables quand le système démarre, mais d'autres (comme `File` et `QTk`) doivent être chargés. Nous montrons comment charger `File` et `QTk`. Plus d'informations sur les modules sont données dans la section 3.8.

3.7.1 L'entrée/sortie de texte dans un fichier

Une manière simple d'interfacer un programme déclaratif avec le monde réel est d'utiliser des fichiers. Un fichier est une séquence de valeurs qui est enregistrée à l'extérieur du programme sur un substrat permanent comme un disque dur. Un fichier de texte est un fichier qui contient une séquence de caractères. Dans cette section, nous expliquons comment lire et écrire des fichiers de texte. Cela suffit pour l'utilisation pratique des programmes déclaratifs. La manière d'utilisation est simple :

Fichier d'entrée $\xrightarrow{lire}$ évaluation de fonction $\xrightarrow{écrire}$ fichier de sortie

Nous utilisons le module `File`, qui peut être chargé sur le site Web du livre.

Charger le module `File`

La première chose à faire est de charger le module `File` dans le système, comme l'explique l'annexe A.1.2. Nous supposons que vous avez une version compilée du module `File`, dans le fichier `File.ozf`. Alors exécutez l'instruction suivante :

```
declare [File]={Module.link [´File.ozf´]}
```

`Module.link` est appelée avec une liste de noms ou chemins de modules compilés. Dans cet exemple il n'y a qu'un module compilé. Le module est chargé avec le nom `File`, ses liens sont édités dans le système et il est initialisé.[15] Maintenant nous sommes prêts pour faire des opérations sur des fichiers.

15. Pour être précis, le module est chargé et ses liens sont édités de façon paresseuse : les liens seront effectivement édités au moment de sa première utilisation.

Lire un fichier

L'opération `File.readList` lit tout le contenu du fichier et renvoie une chaîne de caractères :

```
L={File.readList "foo.txt"}
```

Cet exemple lit le fichier `foo.txt` et crée la chaîne `L`. Nous pouvons aussi écrire :

```
L={File.readList ´foo.txt´}
```

Rappelez-vous que `"foo.txt"` est une chaîne (une liste de codes de caractère) et `´foo.txt´` est un atome (une constante avec une représentation imprimée). Le nom du fichier peut être représenté des deux manières. Il y a une troisième manière pour représenter un nom de fichier : comme une chaîne virtuelle (« *virtual string* »). Une chaîne virtuelle est un tuple avec étiquette `´#´` qui représente une chaîne de caractères. Par exemple, nous pouvons écrire :

```
L={File.readList foo#´.´#txt}
```

Le tuple `foo#´.´#txt`, que nous pouvons écrire comme `´#´(foo ´.´ txt)`, représente la chaîne `"foo.txt"`. L'utilisation des chaînes virtuelles nous évite de faire des concaténations explicites des chaînes. Toutes les opérations de base de Mozart qui prennent des chaînes sont valables aussi avec des chaînes virtuelles. Les trois manières de charger `foo.txt` ont le même effet. Elles lient `L` à une liste des codes de caractère dans le fichier `foo.txt`.

On peut aussi référencer un fichier par URL. Un URL (« *Uniform Resource Locator* ») est une adresse globale commode pour les fichiers parce qu'il est largement soutenu par l'infrastructure du World Wide Web. Il est aussi facile de lire un fichier par son URL que par son nom de fichier :

```
L={File.readList
   ´http ://www.mozart-oz.org/features.html´}
```

On peut utiliser les URL uniquement pour lire les fichiers, mais pas pour les écrire. C'est parce que les URL sont traités par les serveurs Web, qui en général ne permettent que la lecture.

Mozart a des opérations pour la lecture incrémentale ou paresseuse d'un fichier, au lieu de tout d'un coup. C'est important pour des fichiers qui sont trop grands pour l'espace mémoire du processus Mozart. Pour la simplicité, nous recommandons pour l'instant de lire le fichier tout d'un coup. Dans la documentation de Mozart il est expliqué comment faire la lecture incrémentale [35, 97].

Écrire un fichier

Un fichier est généralement écrit incrémentalement, en y ajoutant une chaîne de caractères à la fois. Le module `File` fournit trois opérations : `File.writeOpen` pour ouvrir le fichier au début ; `File.write` pour ajouter une chaîne (ou un atome) au fichier ; et `File.writeClose` pour fermer le fichier à la fin. Voici un exemple :

```
{File.writeOpen ´foo.txt´}
{File.write ´Cette phrase sera dans le fichier.\n´}
{File.write "Les chaînes sont acceptées aussi.\n"}
{File.writeClose}
```

Après ces opérations, le fichier `'foo.txt'` contient les deux lignes de texte suivantes :

```
Cette phrase sera dans le fichier.
Les chaînes sont acceptées aussi.
```

3.7.2 L'entrée/sortie de texte avec une interface graphique

La façon la plus directe d'interfacer un programme avec un être humain est avec une interface graphique. Cette section montre une manière simple mais puissante pour définir des interfaces graphiques, en utilisant des spécifications concises qui sont déclaratives pour la plupart. C'est un excellent exemple d'un langage déclaratif descriptif, comme nous l'avons défini dans la section 3.1. Le langage descriptif est compris par le module `QTk` du système Mozart. L'interface graphique est spécifiée comme un enregistrement imbriqué, dans lequel on ajoute quelques objets et procédures. (Les objets seront introduits dans le chapitre 6. Pour l'instant, vous pouvez les considérer comme des procédures avec un état interne, comme les exemples du chapitre 1.)

Cette section explique comment construire des interfaces graphiques pour entrer et sortir des données textuelles dans une fenêtre. Cela suffit pour beaucoup de programmes déclaratifs. Nous donnons un bref résumé du module `QTk`, juste assez pour construire ces interfaces. Pour plus d'informations voir la documentation `QTk` [31].

La spécification déclarative des gadgets logiciels

Une fenêtre sur l'écran contient un ensemble de gadgets logiciels. Un gadget logiciel (« *widget* ») est une surface rectangulaire dans une fenêtre qui a un comportement interactif particulier. Par exemple, certains gadgets peuvent afficher du texte ou des informations graphiques, et d'autres gadgets peuvent accepter des entrées comme des clics du clavier ou de la souris. Nous spécifions chaque gadget de façon déclarative par un enregistrement dont l'étiquette et les noms des champs définissent le type du gadget et son état initial. La fenêtre est spécifiée comme un enregistrement imbriqué (un arbre) qui définit la structure logique des gadgets dans la fenêtre. Voici les cinq gadgets que nous utilisons :

– Le gadget `label` spécifie un texte à afficher par l'enregistrement

```
label(text:VS)
```

où `VS` est une chaîne virtuelle.

– Le gadget `text` spécifie un texte à afficher et permet d'entrer plusieurs lignes de texte. Il peut utiliser des barres de défilement (« *scroll bars* ») pour afficher plus de texte que l'on peut voir en une fois. Avec une barre verticale (« td » veut dire « top-down »), le gadget est spécifié par l'enregistrement

```
text(handle:H tdscrollbar:true)
```

Quand la fenêtre est créée, la variable `H` sera liée à un objet qui sert à contrôler le gadget. Nous appelons un tel objet un manipulateur (« *handler* »). Vous pouvez considérer l'objet comme une procédure avec un argument : `{H set(VS)}` affiche le texte `VS` et `{H get(VS)}` lit le texte et le renvoie dans `VS`.

– Le gadget `button` spécifie un bouton et une action à exécuter quand le bouton est appuyé. Le gadget est spécifié par l'enregistrement

```
button(text:VS action:P)
```

où `VS` est un chaîne virtuelle et `P` est une procédure avec zéro arguments. `{P}` est invoquée chaque fois que le bouton est appuyé.[16] Pour chaque fenêtre, toutes les actions sont exécutées séquentiellement.

– Les gadgets `td` (« *top-down* ») et `lr` (« *left-to-right* ») spécifient un arrangement d'autres gadgets dans un ordre de haut en bas ou de gauche à droite :

```
lr(W1 W2 ... Wn)
td(W1 W2 ... Wn)
```

où `W1`, `W2`, ..., `Wn` sont d'autres gadgets.

La spécification déclarative des changements de taille

Quand la taille d'une fenêtre est changée, les gadgets à l'intérieur doivent se comporter correctement, c'est-à-dire changer de taille ou garder la même taille, selon ce que doit faire l'interface. Le comportement d'un gadget lors d'un changement de taille est spécifié déclarativement avec un champ facultatif `glue` dans l'enregistrement du gadget. Le champ `glue` indique pour chacun des quatre côtés du gadget s'il doit être « collé » ou pas au gadget qui l'entoure. La valeur du champ `glue` est un atome qui peut contenir toute combinaison des quatre caractères `n` (nord), `s` (sud), `w` (ouest, « *west* ») ou `e` (est), ce qui indique pour chaque direction si le bord doit être collé ou pas (le nord vers le haut). Voici quelques exemples :

16. Pour être précis, quand le bouton gauche de la souris est appuyé et relâché pendant que la souris reste au-dessus du gadget. Cela permet à l'utilisateur de corriger un clic erroné.

- Pas de colle. Le gadget garde sa taille naturelle et il est centré dans l'espace qui lui est donné, horizontalement et verticalement.
- `glue:nswe` colle aux quatre bords et s'étend pour prendre tout l'espace qui lui est donné, horizontalement et verticalement.
- `glue:we` colle horizontalement gauche et droite et s'étend pour prendre tout l'espace horizontal. Verticalement, le gadget ne s'étend pas mais reste centré dans l'espace qui lui est donné.
- `glue:w` colle à gauche et ne s'étend pas.
- `glue:wns` colle verticalement en haut et en bas, s'étend verticalement pour prendre tout l'espace vertical, et colle à gauche sans s'étendre horizontalement.

Le chargement du module `QTk`

La première étape dans la construction d'une interface graphique est le chargement et l'édition des liens du module `QTk` dans le système. Comme `QTk` fait partie de la Mozart Standard Library (la bibliothèque standard), il suffit de connaître son chemin dans l'arborescence de la bibliothèque. Nous le chargeons avec la commande suivante :

```
declare
[QTk]={Module.link [´x-oz ://system/wp/QTk.ozf´]}
```

(Dans le *Labo interactif*, il est chargé automatiquement si on en a besoin pour l'exécution d'un exemple.) Maintenant que les liens de `QTk` sont édités, nous pouvons l'utiliser pour construire des interfaces selon les spécifications de la section précédente.

La construction de l'interface

Le module `QTk` a une fonction `QTk.build` qui prend une spécification d'interface, qui est un enregistrement imbriqué de gadgets logiciels, et construit une fenêtre avec ses gadgets. Voici une interface simple avec un bouton qui affiche `ouille` dans le Browser chaque fois que le bouton est cliqué :

```
D=td(button(text:"Cliquez-moi"
            action:proc {$} {Browse ouille} end))
W={QTk.build D} {W show}
```

L'enregistrement `D` doit toujours commencer avec `td` ou `lr`, même si la fenêtre n'a qu'un gadget. `QTk.build` renvoie un objet `W` qui représente la fenêtre. Au début elle est cachée. Elle peut être affichée ou cachée de nouveau avec les appels `{W show}` et `{W hide}`. La figure 3.14 montre une spécification plus élaborée pour une interface complète d'entrée/sortie de texte. La figure 3.15 montre l'interface. À première vue, la spécification peut sembler compliquée, mais les apparences sont trompeuses : il y

```
declare In Out
A1=proc {$} X in {In get(X)} {Out set(X)} end
A2=proc {$} {W close} end
D=td(title:"Simple text I/O interface"
     lr(label(text:"Input :")
        text(handle:In  tdscrollbar:true glue:nswe)
        glue:nswe)
     lr(label(text:"Output :")
        text(handle:Out tdscrollbar:true glue:nswe)
        glue:nswe)
     lr(button(text:"Do It" action:A1 glue:nswe)
        button(text:"Quit"  action:A2 glue:nswe)
        glue:we))
W={QTk.build D} {W show}
```

Figure 3.14 Une interface graphique simple pour l'entrée/sortie du texte.

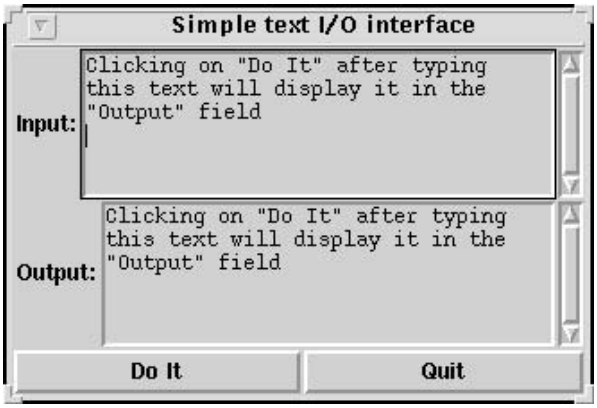


Figure 3.15 Une capture d'écran de l'interface.

a six gadgets (deux `label`, deux `text`, deux `button`) arrangés avec `td` et `lr`. La fonction `QTk.build` prend la description `D`, construit la fenêtre et crée les objets manipulateurs `In` et `Out`. Comparez l'enregistrement `D` de la figure 3.14 avec la capture d'écran dans la figure 3.15.

Il y a deux procédures d'action, `A1` et `A2`, une pour chaque bouton. L'action `A1` est attachée au bouton « `Do It` ». Cliquer sur le bouton appelle `A1`, ce qui transfère du texte du premier gadget texte vers le deuxième gadget texte. L'appel `{In get(X)}` obtient le texte du premier gadget texte et le renvoie dans `X`. Ensuite `{Out set(X)}` met le texte `X` dans le deuxième gadget texte. L'action `A2` est attachée au bouton « `Quit` ». Elle appelle `{W close}`, ce qui ferme la fenêtre de façon permanente.

Avec la colle `nswe` presque partout, la fenêtre se comporte bien quand on change sa taille. Le gadget `lr` qui contient les deux boutons est construit avec la colle `we`, pour que les boutons ne s'étendent pas verticalement. Les gadgets `label` n'ont pas de colle, ils ont donc une taille fixe. Le gadget `td` à la racine n'a pas besoin de colle parce qu'il reste toujours collé à sa fenêtre.

3.8 LA PROGRAMMATION À PETITE ÉCHELLE

Maintenant que nous avons vu des techniques de programmation, nous allons les utiliser pour résoudre des problèmes. Cette étape s'appelle la **conception de programmes**. Elle commence avec un problème à résoudre, dans la plupart des cas expliqué en mots et pas toujours avec beaucoup de précision. Nous concevons la structure du programme à haut niveau avec les techniques de programmation appropriées. Finalement nous avons un programme complet qui résout le problème.

Dans la conception de programmes, il y a une distinction importante entre « la programmation à petite échelle » et « la programmation à grande échelle ». Nous appelons les programmes respectivement des « petits programmes » et des « grands programmes ». Cette distinction n'a rien à voir avec la taille du programme en nombre de lignes de code source, mais plutôt avec le nombre de personnes impliquées dans son développement. Les petits programmes sont écrits par une personne sur une courte période de temps. Les grands programmes sont écrits par plusieurs personnes ou sur une longue période de temps. Cette section donne une introduction à la programmation à petite échelle ; la programmation à grande échelle est étudiée dans la section 5.6.

3.8.1 La méthodologie de conception

Supposons que nous avons un problème qui peut être résolu par un petit programme. Pour concevoir ce programme, nous recommandons la méthodologie de conception suivante, qui est un mélange de créativité et de réflection rigoureuse :

- *La spécification informelle*. Nous commençons par noter le plus précisément possible ce que le programme doit faire : les entrées et sorties et les relations entre elles. Cette description s'appelle une spécification informelle. Elle est « informelle » parce qu'elle est écrite en langage naturel. Les spécifications « formelles » sont écrites en notation mathématique.
- *Les exemples*. Pour que la spécification soit limpide, il est toujours bon d'imaginer des exemples de ce que le programme fait dans des cas particuliers. Les exemples doivent « stresser » le programme : l'utiliser dans des conditions de limite et de la façon la plus imprévue imaginable.
- *L'exploration*. Pour découvrir la structure nécessaire du programme, une bonne manière est d'utiliser l'interface interactive pour expérimenter avec des fragments

de programme. L'idée est d'écrire de petites opérations qui sont nécessaires. Nous utilisons les opérations fournies par le système comme point de départ. Cette étape nous donne une vue plus claire de ce que doit être la structure du programme.

– *La structure et le codage*. À ce point nous pouvons tracer la structure du programme. Nous faisons un plan préliminaire de la structure du programme avec des opérations de haut niveau. Ensuite nous complétons le code effectif du programme avec les autres opérations. Toutes les opérations doivent être simples. Pour améliorer la structure nous pouvons grouper les opérations apparentées dans des modules.
– *Les tests et le raisonnement*. Maintenant que nous avons un programme, nous devons vérifier qu'il fait ce qu'il faut. Nous l'essayons sur une série de tests, y compris les exemples que nous avons imaginés auparavant. Nous corrigeons des erreurs jusqu'à ce que le programme fonctionne bien. Nous pouvons aussi raisonner sur le programme et sa complexité en utilisant la sémantique formelle. Le raisonnement est important surtout pour les parties qui ne sont pas claires. Les tests et le raisonnement sont complémentaires : il est important de faire les deux pour obtenir un programme de qualité.
– *L'évaluation de la qualité*. Le point final est de prendre du recul pour juger la qualité du programme. Il y a beaucoup de facteurs impliqués dans la qualité : le fait qu'il résout le bon problème, l'exactitude, l'efficacité, la facilité d'entretien, l'extensibilité et la simplicité. La simplicité est particulièrement importante, parce qu'elle facilite beaucoup d'autres facteurs. Un programme compliqué est un programme inachevé. À côté de la simplicité, il y a aussi la complétude : la conception a-t-elle (potentiellement) un ensemble complet de fonctionnalités, pour qu'elle puisse être utilisée comme une base pour l'avenir ?

Cette méthodologie n'est pas figée. Vous devez adapter ces étapes à vos circonstances. Par exemple, en imaginant des exemples, le programmeur peut réaliser que la spécification doit être changée. Mais n'oubliez pas l'étape la plus importante, les tests. Ils sont importants parce qu'ils ferment la boucle : ils donnent une rétroaction de l'étape du codage vers l'étape de la spécification.

3.8.2 Un exemple de conception de programme

Pour illustrer ces étapes, traçons le développement d'une application qui compte la fréquence des mots dans un fichier texte. Voici une première tentative d'une spécification informelle :

> Si on donne le nom d'un fichier, l'application ouvrira une fenêtre et affichera une liste de paires, où chaque paire contient un mot et un entier qui compte le nombre de fois que le mot apparaît dans le fichier.

Cette spécification est-elle suffisamment précise ? Que faire d'un fichier qui contient un mot qui n'est pas dans le dictionnaire ou d'un fichier qui contient des caractères non ASCII ? Notre spécification n'est pas assez précise : elle ne définit pas le terme « mot ». Pour la rendre plus précise nous devons connaître le but de l'application. Disons que nous voulons juste avoir une idée générale des fréquences des mots qui est indépendante de tout langage particulier. Nous pouvons alors définir un mot tout simplement comme :

> Un « mot » est une séquence contiguë maximale de lettres et chiffres.

Les mots sont donc séparés par au moins un caractère qui n'est pas une lettre ou un chiffre. Cette définition accepte un mot qui n'est pas dans le dictionnaire mais n'accepte pas des mots qui contiennent des caractères non ASCII. Est-ce suffisant ? Que faire des mots avec un tiret (comme « sous-classe ») ou des expressions qui se comporte comme des unités (comme « premier entré premier sorti ») ? Pour garder la simplicité, nous ne faisons rien de spécial pour ces cas. Mais il faudra peut-être changer la spécification plus tard pour les traiter. Cela dépend de la façon dont nous utilisons l'application.

Nous avons maintenant une spécification finale. Remarquez le rôle essentiel joué par les exemples. Ils sont des balises importantes vers une spécification précise. Nous avons conçu nos exemples spécialement pour tester les limites de la spécification.

La prochaine étape est de concevoir la structure du programme. La structure appropriée semble être un pipeline : d'abord lire le fichier dans une liste de caractères et ensuite convertir la liste en une liste de mots, où un mot est représenté comme une chaîne de caractères. Pour compter les mots, nous utilisons une structure de données qui s'appelle un **dictionnaire** (voir figure 3.18 plus loin). Le dictionnaire sera utilisé par le cœur de l'application, la fonction `WordFreq` qui prend une liste de caractères et qui renvoie un dictionnaire qui contient les comptes (voir figure 3.19). La sortie de `WordFreq` sera affichée dans une fenêtre. Nous utilisons les opérations de fichier et les opérations d'interface graphique expliquées dans la section 3.7. Enfin, nous emballons l'application proprement dans un composant logiciel qui peut être exécuté de façon autonome. Nous expliquerons toute la démarche dans les sections suivantes.

3.8.3 Les composants logiciels

Comment un programme doit-il être organisé ? On pourrait l'écrire comme un bloc monolithique, mais ce serait difficile à comprendre. Une meilleure approche est de le partitionner en unités logiques, dont chacune implémente un ensemble d'opérations apparentées. Chaque unité logique a deux parties, une interface et une implémentation. Seule l'interface est visible de l'extérieur de l'unité logique. Une unité logique peut en utiliser d'autres dans son implémentation.

Ainsi un programme est simplement un graphe orienté d'unités logiques, où une arête entre deux unités veut dire que la première a besoin de la deuxième pour son implémentation. Dans l'usage populaire, ces unités logiques sont appelées « modules » ou « composants ». Cette section introduit les concepts de base avec une définition précise et montre comment ils peuvent être utilisés pour aider dans la conception de petits programmes déclaratifs. La section 5.6 explique l'utilité de ces idées dans la conception de grands programmes.

Les modules et les foncteurs

Un module regroupe des opérations apparentées dans une entité avec une interface et une implémentation. Nos modules ont une représentation simple :

- L'interface du module est un enregistrement qui regroupe des entités du langage (typiquement des procédures, mais tout est permis y compris des classes, des objets, etc.).
- L'implémentation du module est un ensemble d'entités qui sont accessibles uniquement par les opérations de l'interface. L'implémentation est cachée avec la portée lexicale.

Nous considérons les spécifications de modules séparément des modules eux-mêmes. Une spécification de module est une sorte de gabarit qui crée un module chaque fois qu'elle est instanciée. La spécification est parfois appelée un **composant logiciel**. Malheureusement, le terme « composant logiciel » est utilisé avec beaucoup de significations différentes [93]. Pour minimiser la confusion, nous appellerons les spécifications de modules des **foncteurs**. Un foncteur est une fonction dont les arguments sont des modules et dont le résultat est un nouveau module. (Pour être précis, le foncteur prend les interfaces des modules comme arguments, crée un nouveau module, et renvoie l'interface de ce module !) À cause de l'importance du foncteur dans la structuration des programmes, il est fourni comme une abstraction linguistique. Un foncteur a trois parties : une clause **`import`**, qui spécifie les autres modules dont il a besoin, une clause **`export`**, qui spécifie l'interface du module, et une clause **`define`**, qui donne l'implémentation du module y compris le code d'initialisation. La syntaxe des foncteurs permet leur utilisation comme instructions ou expressions, tout comme les procédures. Le tableau 3.6 donne la syntaxe des déclarations de foncteurs comme instructions. Une déclaration de foncteur peut être une expression si le premier identificateur ⟨variable⟩ est remplacé par une balise d'insertion « `$` ».

Dans la terminologie du génie logiciel, un composant logiciel est une unité de déploiement indépendant, une unité de développement par tiers et n'a pas d'état persistant (d'après la définition donnée par [93]). Les foncteurs satisfont cette définition. Un foncteur est donc une sorte de composant logiciel. Dans cette terminologie, un module est une instance de composant ; il est le résultat de l'édition des liens d'un

```
⟨statement⟩ : :=
   functor ⟨variable⟩
   [ import{ ⟨variable⟩ [ at ⟨atom⟩ ]
             | ⟨variable⟩ ´ ( ´ { (⟨atom⟩ | ⟨int⟩) [ ':' ⟨variable⟩ ] }+ ´ ) ´
             }+ ]
   [ export { [ (⟨atom⟩ | ⟨int⟩) ':' ] ⟨variable⟩ }+ ]
   define { ⟨declPart⟩ }+ [ in ⟨statement⟩ ] end
 | ...
```

Tableau 3.6 La syntaxe des foncteurs.

foncteur dans un environnement de modules. L'environnement de modules contient un ensemble de modules, dont chacun peut avoir un état d'exécution.

Une application est autonome si elle peut être exécutée en dehors de l'interface interactive. Elle contient un foncteur principal qui est évalué quand le programme démarre. Le foncteur principal importe les modules dont il a besoin, ce qui cause l'édition des liens d'autres foncteurs. Le foncteur principal est utile pour démarrer l'application et non pour le module renvoyé, qui est ignoré. L'évaluation ou l'édition des liens d'un foncteur crée un module en trois étapes.

D'abord, une variable est créée avec une synchronisation par besoin.[17] Ensuite, quand on a besoin de la valeur de la variable (ce qui arrive quand le programme essaie de faire une opération sur le module), le foncteur sera chargé en mémoire et appelé dans un nouveau fil avec les modules qu'il importe comme arguments. Cela s'appelle l'**édition dynamique des liens**, par opposition à l'**édition statique des liens** dans laquelle les liens du foncteur sont édités avant l'exécution de l'application. Enfin, l'appel du foncteur renvoie le nouveau module et le lie à la variable. À tout moment, l'ensemble de modules dont les liens sont édités s'appelle l'**environnement de module**.

L'implémentation des modules et des foncteurs

Nous allons construire un composant logiciel. Nous prenons un exemple de module et nous voyons comment obtenir un composant logiciel à partir de l'exemple. Ensuite nous le convertissons en abstraction linguistique.

➤ Un exemple de module

L'interface d'un module est un enregistrement. On peut y accéder par les champs de l'enregistrement. Nous construisons un module `MyList` qui fournit une interface

17. La synchronisation par besoin est l'opération fondamentale de l'exécution paresseuse (voir section 4.1).

pour trois opérations sur les listes, la concaténation, le tri et le test d'appartenance. On peut l'écrire ainsi :

```
declare MyList in
local
   proc {Append ... }  ... end
   proc {MergeSort ...} ... end
   proc {Sort ... } ... {MergeSort ...} ... end
   proc {Member ...} ... end
in
   MyList=´export´(append:Append
                   sort:Sort
                   member:Member)
end
```

La procédure `MergeSort` est inaccessible à l'extérieur de l'instruction **`local`**. On ne peut pas accéder aux autres procédures directement, mais seulement par les champs du module `MyList` qui est un enregistrement. Par exemple, `Append` est accessible comme `MyList.append`. La plupart des modules de bibliothèque de Mozart, comme les modules Base et System, respectent cette structure.

➤ Un composant logiciel

Avec l'abstraction procédurale, nous pouvons convertir ce module en composant logiciel. Le composant logiciel est une fonction qui renvoie un module :

```
fun {MyListFunctor}
   proc {Append ... }  ... end
   proc {MergeSort ...} ... end
   proc {Sort ... } ... {MergeSort ...} ... end
   proc {Member ...} ... end
in
   ´export´(append:Append
            sort:Sort
            member:Member)
end
```

Chaque fois que la fonction `MyListFunctor` est appelée, elle crée un autre module `MyList`. Dans le cas général, `MyListFunctor` aurait comme arguments les autres modules utilisés pour la création de `MyList`.

Cette définition montre clairement que les foncteurs sont des valeurs dans le langage. Ils partagent les propriétés suivantes avec les valeurs procédurales :

– Une définition de foncteur peut être évaluée à l'exécution, ce qui donne une valeur de foncteur.

– Un foncteur peut avoir des références externes aux autres entités du langage. Par exemple, il est facile de faire un foncteur qui contient des données calculées à l'exécution. C'est utile, par exemple, pour inclure de grands tableaux ou des images sous forme numérique.
– Un foncteur peut être enregistré dans un fichier en utilisant le module `Pickle` (voir la documentation de Mozart [23]). Ce fichier peut être lu par tout processus Mozart, ce qui facilite la création des bibliothèques des foncteurs faits par tiers, comme la bibliothèque MOGUL de Mozart.
– Un foncteur est léger ; il peut être utilisé pour encapsuler une entité comme un objet ou une classe, afin de rendre explicite les modules dont l'entité a besoin.

Parce que les foncteurs sont des valeurs, il est possible de les manipuler dans le langage. Par exemple, un composant logiciel peut faire lui-même de la manipulation des composants : le composant détermine les composants dont il a besoin et édite ses liens quand il faut. Davantage de souplesse est possible avec le typage dynamique. Un composant peut éditer les liens d'un autre composant à l'exécution, en éditant directement les liens des foncteurs en les évaluant avec les bons arguments.

➤ L'abstraction linguistique

L'abstraction de composant logiciel est utile pour l'organisation de grands programmes. Pour la rendre plus facile à utiliser, pour garantir qu'elle n'est pas utilisée de façon inexacte et pour clarifier l'intention du programmeur (en évitant la confusion avec d'autres techniques de programmation), nous en faisons une abstraction linguistique. La fonction `MyListFunctor` correspond à la syntaxe suivante de l'abstraction :

```
functor
export
   append:Append
   sort:Sort
   member:Member
define
   proc {Append ... }  ... end
   proc {MergeSort ...} ... end
   proc {Sort ... } ... {MergeSort ...} ... end
   proc {Member ...} ... end
end
```

Remarquez que l'instruction entre **`define`** et **`end`** fait une déclaration implicite des variables, exactement comme l'instruction entre **`local`** et **`in`**.

Supposons que ce foncteur a été compilé et stocké dans le fichier `MyList.ozf`. (Nous verrons plus loin comment compiler un foncteur.) Un module peut être créé dans l'interface interactive :

```
declare [MyList]={Module.link [´MyList.ozf´]}
```

La fonction `Module.link` est définie dans le module système `Module`. Cette fonction prend une liste de foncteurs, les charge, édite leurs liens (les évalue ensemble, pour que chaque module voie ses modules importés) et renvoie une liste correspondante de modules. Le module `Module` permet de faire beaucoup d'autres opérations sur les foncteurs et les modules.

➤ L'importation des modules

Les composants logiciels peuvent dépendre d'autres composants logiciels. Pour être précis, l'instanciation d'un composant logiciel crée un module. Cette instanciation peut avoir besoin d'autres modules. Dans la syntaxe de **`functor`**, nous déclarons cette dépendance avec la clause **`import`**. Pour importer un module système il suffit de donner le nom de son foncteur. Par contre, pour importer un module défini par l'utilisateur il faut donner le nom du fichier ou l'URL où le foncteur est stocké. C'est raisonnable parce que Mozart sait où les modules de bibliothèque sont stockés mais ne sait pas où vous avez stocké vos propres foncteurs. Considérez le foncteur suivant :

```
functor
import
   Browser
   FO at ´file :///home/mydir/FileOps.ozf´
define
   {Browser.browse {FO.countLines ´/etc/passwd´}}
end
```

La clause **`import`** importe le module système `Browser` et le module `FO` défini par l'utilisateur et spécifié par le foncteur dans le fichier `/home/mydir/FileOps.ozf`. Quand les liens de ce foncteur sont édités, l'instruction entre **`define ... end`** est exécutée. La fonction `FO.countLines` est exécutée et ensuite la procédure `Browser.browse` est appelée pour afficher le résultat. Ce foncteur est comme un foncteur principal : il est défini pour son effet et non pour le module qu'il crée. Il n'exporte donc aucune interface.

3.8.4 Un exemple d'implémentation de programme

Nous implémentons le compteur de fréquence des mots comme une application autonome. La figure 3.16 montre une capture d'écran de son exécution. Le programme contient deux composants, `Dict` et `WordApp`, des foncteurs dont le code source est dans les fichiers `Dict.oz` et `WordApp.oz`. Ces composants implémentent un dictionnaire déclaratif et le foncteur principal de l'application. Le composant `WordApp` importe aussi les modules `File` et `QTk`. Il utilise ces modules pour lire le texte et pour afficher les résultats.

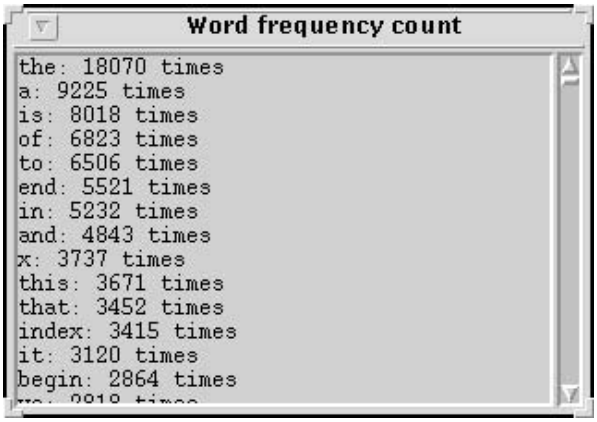


Figure 3.16 Une capture d'écran de l'application compteur de fréquence des mots.

La spécification du dictionnaire déclaratif

Le compteur utilise un ADT qui s'appelle un **dictionnaire déclaratif**. Un dictionnaire est un tableau dynamique où les indices sont des constantes. Ces indices sont appelés les **clés**. Dans notre dictionnaire nous utiliserons des atomes ou des entiers comme clés. Voici l'ensemble d'opérations de base sur le type ⟨Dict⟩ :

- ⟨**fun** `{NewDictionary}` : ⟨Dict⟩⟩ renvoie un nouveau dictionnaire vide.
- ⟨**fun** `{Put` ⟨Dict⟩ ⟨Feature⟩ ⟨Value⟩`}` : ⟨Dict⟩⟩ prend un dictionnaire et renvoie un nouveau dictionnaire qui ajoute la paire ⟨Feature⟩→⟨Value⟩. Si ⟨Feature⟩ existe déjà, le nouveau dictionnaire contiendra ⟨Value⟩ au lieu de l'ancien contenu.
- ⟨**fun** `{Get` ⟨Dict⟩ ⟨Feature⟩`}` : ⟨Value⟩⟩ renvoie la valeur qui correspond à ⟨Feature⟩. S'il n'y en a pas, une exception est levée.
- ⟨**fun** `{Entries` ⟨Dict⟩`}` : ⟨List ⟨Feature⟩⟩⟩ renvoie une liste des paires de clés et valeurs dans ⟨Dict⟩. Chaque paire est un tuple avec étiquette ´#´.

Pour cet exemple nous définissons le type ⟨Feature⟩ comme ⟨Atom⟩ | ⟨Int⟩. La figure 3.18 montre une implémentation dans laquelle le dictionnaire est représenté comme une liste de paires `Key#Value` qui sont triées sur la clé. Au lieu de `Get`, nous définissons une opération `CondGet` un peu plus générale :

- ⟨**fun** `{CondGet` ⟨Dict⟩ ⟨Feature⟩ ⟨Value⟩$_1$`}` : ⟨Value⟩$_2$⟩ renvoie la valeur qui correspond à ⟨Feature⟩. Si ⟨Feature⟩ est absent, le dernier argument ⟨Value⟩$_1$ sera renvoyé.

Pour l'implémentation du dictionnaire, nous utilisons une opération pour convertir les chaînes de caractères en atomes : `StringToAtom` (voir documentation de Mozart [69]). Cela permet de faire une implémentation plus rapide du dictionnaire.

Le code source et les composants

Le code source complet des composants `Dict` et `WordApp` est affiché dans les figures 3.18 et 3.19. Ce code utilise la boucle **`for`** qui est expliquée dans la documentation de Mozart [21]. La principale différence entre le code autonome et le code que l'on écrirait dans l'interface interactive est que les composants sont entourés de **`functor ... end`** avec les clauses **`import`** et **`export`**. La figure 3.17 montre les dépendances. Les modules `Open` et `Finalize` sont des modules système de Mozart [22]. Le composant `File` est disponible sur le site Web du livre. Le composant `QTk` fait partie de la Mozart Standard Library. Il peut être utilisé dans un foncteur en ajoutant la clause suivante :

```
import QTk at 'x-oz://system/wp/QTk.ozf'
```

Le composant `Dict` satisfait la spécification donnée dans la section précédente.

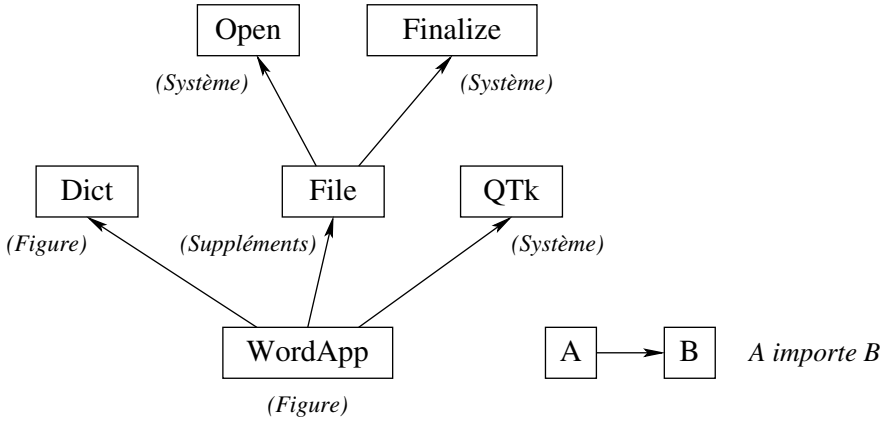


Figure 3.17 Les dépendances de l'application compteur de fréquence des mots.

La compilation et l'exécution autonomes

Nous compilons l'application comme un programme autonome. Un foncteur peut être utilisé de deux manières : comme foncteur compilé, qui est importable par d'autres foncteurs, ou comme programme autonome, qui peut être exécuté directement sur la ligne de commande. Tout foncteur peut être compilé en programme autonome. Dans ce cas, aucune clause **`export`** n'est nécessaire et la partie initialisation (entre **`define`**

```
functor
export
   new:NewDict put:Put condGet:CondGet entries:Entries
define
   fun {NewDict} leaf end
   fun {Put Ds Key Value}
      case Ds of leaf then tree(Key Value leaf leaf)
      [] tree(K _ L R) andthen K==Key then
                          tree(K Value L R)
      [] tree(K V L R) andthen K>Key then
                          tree(K V {Put L Key Value} R)
      [] tree(K V L R) andthen K<Key then
                          tree(K V L {Put R Key Value})
      end
   end
   fun {CondGet Ds Key Default}
      case Ds of leaf then Default
      [] tree(K V _ _) andthen K==Key then V
      [] tree(K _ L _) andthen K>Key then
                          {CondGet L Key Default}
      [] tree(K _ _ R) andthen K<Key then
                          {CondGet R Key Default}
      end
   end
   fun {Entries Ds}
      proc {EntriesD Ds S1 ?Sn}
         case Ds of leaf then
            S1=Sn
         [] tree(K V L R) then S2 S3 in
            {EntriesD L S1 S2}
            S2=K#V|S3
            {EntriesD R S3 Sn}
         end
      end
   in {EntriesD Ds $ nil} end
end
```

Figure 3.18 Un dictionnaire en forme bibliothèque (fichier `Dict.oz`).

```
functor
import
   QTk at 'x-oz ://system/wp/QTk.ozf' Dict File
define
   fun {WordChar C}
   (&a=<C andthen C=<&z) orelse (&A=<C andthen C=<&Z)
   orelse (&0=<C andthen C=<&9) end
   fun {WordToAtom PW} {StringToAtom {Reverse PW}} end
   fun {IncW D W}
      {Dict.put D W {Dict.condGet D W 0}+1} end
   fun {CharsToWords PW Cs}
      case Cs of nil andthen PW==nil then nil
      [] nil then [{WordToAtom PW}]
      [] C|Cr andthen {WordChar C} then
         {CharsToWords {Char.toLower C}|PW Cr}
      [] _|Cr andthen PW==nil then
         {CharsToWords nil Cr}
      [] _|Cr then {WordToAtom PW}|{CharsToWords nil Cr}
      end
   end
   fun {CountWords D Ws}
      case Ws of W|Wr then {CountWords {IncW D W} Wr}
      [] nil then D end
   end
   fun {WordFreq Cs}
      {CountWords {Dict.new} {CharsToWords nil Cs}} end

   L={File.readList stdin}
   E={Dict.entries {WordFreq L}}
   S={Sort E fun {$ A B} A.2>B.2 end}
   H Des=td(title:'Word frequency count'
            text(handle:H tdscrollbar:true glue:nswe))
   W={QTk.build Des} {W show}
   for X#Y in S do
      {H insert('end' X#' : '#Y#' times\n')} end
end
```

Figure 3.19 L'application autonome compteur de fréquence des mots (fichier `WordApp.oz`).

et **end**) définit l'effet du programme. Nous compilons d'abord le fichier `Dict.oz` pour en faire un foncteur compilé, avec la commande `ozc` sur la ligne de commande :

```
ozc -c Dict.oz
```

Nous compilons ensuite le fichier `WordApp.oz` pour en faire le programme autonome `WordApp`, avec la commande suivante :

```
ozc -x WordApp.oz
```

On peut l'exécuter ainsi :

```
WordApp < book.raw
```

où `book.raw` est un fichier qui contient un texte. Le texte est passé à l'entrée standard du programme, qui est vue par le programme comme un fichier avec le nom `stdin`. Le programme fera l'édition dynamique des liens de `Dict.ozf` quand le dictionnaire est utilisé pour la première fois. Il est aussi possible d'éditer les liens de `Dict.ozf` statiquement dans le code compilé de l'application `WordApp`, ce qui évite la nécessité de faire une édition dynamique des liens (voir documentation du système Mozart [69]). Le code compilé s'appelle aussi code machine, parce qu'il s'exécute sur la machine.

Les modules de bibliothèque

L'application utilise le module `QTk` qui fait partie du système Mozart. Tout langage de programmation, afin d'être pratique, doit être accompagné d'un ensemble d'abstractions utiles. Celles-ci sont organisées en bibliothèques. Une bibliothèque (« *library* ») est une collection cohérente d'une ou plusieurs abstractions apparentées qui sont utiles dans un domaine particulier de problèmes. Selon le langage et la bibliothèque, la bibliothèque peut être considérée comme une partie du langage ou à l'extérieur du langage. La ligne de démarcation est floue : souvent la plupart des opérations de base du langage sont implémentées dans des bibliothèques. Par exemple, les fonctions sur les nombres réels (sinus, cosinus, logarithme, etc.) sont dans la plupart des cas implémentées dans des bibliothèques. Comme le nombre de bibliothèques peut être très grand, c'est une bonne idée de les organiser comme des modules.

Les modules de bibliothèque disponibles dans le système Mozart sont regroupés en modules de base (« *Base modules* », toujours disponibles au démarrage [23]) et modules système (« *System modules* », disponibles dans l'interface interactive et à importer dans les foncteurs [22]). Les modules dans le Mozart Standard Library, comme `QTk`, doivent toujours être importés, même dans l'interface interactive.

3.9 EXERCICES

➤ **Exercice 1** — *La valeur absolue des nombres réels*
Nous voulons définir une fonction `Abs` qui calcule la valeur absolue d'un nombre réel. La définition suivante est fausse :

```
fun {Abs X} if X<0 then ~X else X end end
```

Pourquoi ? Comment la corrigeriez-vous ?
Indice : Le problème est trivial.

➤ **Exercice 2** — *Les racines cubiques*
Ce chapitre utilise la méthode de Newton pour calculer les racines carrées. La méthode peut être étendue pour calculer des racines de tout degré. Par exemple, la méthode suivante calcule les racines cubiques. Soit une estimation g de la racine cubique de x, une estimation améliorée est donnée par $(x/g^2 + 2g)/3$. Écrivez un programme déclaratif pour calculer les racines cubiques avec la méthode de Newton.

➤ **Exercice 3** — *La méthode du demi-intervalle*[18]
La méthode du demi-intervalle est une technique simple mais puissante pour trouver des racines de l'équation $f(x) = 0$, où f est une fonction réelle continue. L'idée est la suivante : si nous avons des nombres réels a et b tels que $f(a) < 0 < f(b)$, alors f doit avoir au moins une racine entre a et b. Pour la localiser, prenez $x = (a+b)/2$ et calculez $f(x)$. Si $f(x) > 0$, f aura une racine entre a et x. Si $f(x) < 0$, f aura une racine entre x et b. La répétition de cette procédure définira des intervalles de plus en plus petits qui convergeront vers une racine. Écrivez un programme déclaratif pour résoudre ce problème avec les techniques du calcul itératif.

➤ **Exercice 4** — *Une* `SumList` *itérative*
Réécrivez la fonction `SumList` de la section 3.4.2 pour qu'elle soit itérative avec les techniques développées pour `Length`.

➤ **Exercice 5** — *Les invariants d'état*
Écrivez un invariant d'état pour la fonction `IterReverse`.

➤ **Exercice 6** — *Une autre* `Append`
La section 3.4.2 définit la fonction `Append` en faisant de la récursion sur le premier argument. Que se passe-t-il si nous essayons de faire de la récursion avec le deuxième argument ? Voici une solution possible :

18. Cet exercice vient de [1].

```
fun {Append Ls Ms}
   case Ms of nil then Ls
   [] X|Mr then {Append {Append Ls [X]} Mr} end
end
```

Ce programme est-il correct ? Termine-t-il ? Pourquoi oui ou pourquoi non ?

➤ **Exercice 7** — *Une* `Append` *itérative*

Cet exercice explore la puissance d'expression des variables dataflow. Dans le modèle déclaratif, la définition suivante d'`Append` est itérative :

```
fun {Append Xs Ys}
   case Xs of nil then Ys
   [] X|Xr then X|{Append Xr Ys} end
end
```

Nous pouvons vérifier cela en regardant l'expansion :

```
proc {Append Xs Ys ?Zs}
   case Xs of nil then Zs=Ys
   [] X|Xr then Zr in Zs=X|Zr {Append Xr Ys Zr} end
end
```

Cette définition fait une récursion terminale parce que la variable non liée `Zr` peut être placée dans la paire `X|Zr` et liée après. Restreignons maintenant le modèle de calcul pour utiliser uniquement des valeurs. Comment pouvons-nous écrire une `Append` itérative ? Une approche est de définir deux fonctions : (1) une fonction itérative qui inverse une liste et (2) une fonction itérative qui fait la concaténation de l'inverse d'une liste avec une autre liste. Définissez une `Append` itérative avec cette approche.

➤ **Exercice 8** — *Les calculs itératifs et les variables dataflow*

L'exercice précédent montre que l'utilisation des variables dataflow peut simplifier la définition des fonctions itératives sur les listes. Pour toute opération itérative définie avec des variables dataflow, est-il possible de définir une autre définition itérative de la même opération qui n'utilise pas les variables dataflow ?

➤ **Exercice 9** — *Le test de paire de liste*

La section 3.4.3 définit une fonction `LengthL` qui calcule le nombre d'éléments dans une liste imbriquée. Pour voir si `X` est une paire de liste ou pas, `LengthL` utilise la fonction `IsCons` :

```
fun {IsCons X}
   case X of _|_ then false else true end end
```

Que se passe-t-il si nous utilisons la définition suivante :

```
fun {IsCons X} X\=(_|_) end
```

Pourquoi cette version de `IsCons` ne fonctionne-t-elle pas ?

➤ **Exercice 10** — *Les opérations sur les matrices*
Supposons que nous représentons une matrice comme une liste de listes, où chaque liste interne contient une rangée de la matrice. Définissez des fonctions pour faire les opérations de base sur les matrices comme l'addition, la transposition et la multiplication.

➤ **Exercice 11** — *Le tri rapide*
Voici un algorithme pour trier les listes. Son inventeur C.A.R. Hoare l'a appelé tri rapide (« *quicksort* »), parce qu'à l'époque de son invention il était l'algorithme de tri général le plus rapide connu. Il utilise une stratégie de diviser pour régner qui donne une complexité en temps moyenne $O(n \log n)$. Voici une description informelle de l'algorithme pour le modèle déclaratif. Avec une liste d'entrée `L`, faites les opérations suivantes :

a) Choisissez le premier élément `X` de `L` comme l'élément pivot.

b) Partitionnez `L` en deux listes, `L1` et `L2`, tel que chaque élément de `L1` est plus petit que `X` et chaque élément de `L2` est plus grand ou égal à `X`.

c) Utilisez le tri rapide pour trier `L1` dans `S1` et pour trier `L2` dans `S2`.

d) Concaténez les listes `S1` et `S2` pour obtenir la réponse.

Chapitre **4**

La programmation concurrente dataflow

> *Il y a vingt ans, le ski parallèle était considéré comme une compétence acquise seulement après beaucoup d'années de formation et d'entraînement. Aujourd'hui, il est appris en une seule saison. . . . Tous les buts des parents sont atteints par les enfants : . . . Mais les mouvements qu'ils font pour produire ces résultats sont très différents.*
>
> – *Mindstorms : Children, Computers, and Powerful Ideas*, Seymour Papert (1980)

Le modèle déclaratif du chapitre 2 nous permet d'écrire beaucoup de programmes et de leur appliquer des techniques de raisonnement puissantes. Mais, comme la section 4.4 l'explique, il existe des programmes utiles qui ne peuvent pas être écrits dans ce modèle. Par exemple, des programmes qui contiennent un ensemble d'activités qui s'exécutent de façon indépendante. Ils sont dits **concurrents**. La concurrence est essentielle pour les programmes qui interagissent avec leur environnement, comme les agents, les programmes avec des interfaces graphiques, les systèmes d'exploitation, etc. Elle permet aussi d'organiser un programme en parties indépendantes qui interagissent uniquement quand il le faut comme des programmes client/serveur et producteur/consommateur. L'indépendance est une propriété importante en génie logiciel.

La concurrence peut être simple

Ce chapitre étend le modèle déclaratif du chapitre 2 avec la concurrence tout en restant déclaratif. Toutes les techniques de programmation et de raisonnement pour la

programmation déclarative restent applicables. C'est une propriété remarquable qui mérite d'être mieux connue. L'intuition sous-jacente est assez simple. Elle est basée sur le fait qu'une variable dataflow ne peut être liée qu'à une valeur. La programmation concurrente avec des variables dataflow gardera donc les bonnes propriétés de la programmation déclarative.

– Ce qui ne change pas : Le résultat d'un programme est le même, qu'il soit concurrent ou pas. Mettre une partie quelconque d'un programme dans un fil ne change pas le résultat.
– Ce qui est nouveau : Le résultat d'un programme peut être calculé de façon incrémentale. Si l'entrée d'un programme concurrent est donnée incrémentalement, le programme calculera sa sortie incrémentalement aussi.

Donnons un exemple pour concrétiser cette intuition. Considérons un programme séquentiel qui calcule une liste de carrés à partir d'une liste d'entiers qui est transformée par `Map` :

```
fun {Gen L H}
   {Delay 100}
   if L>H then nil else L|{Gen L+1 H} end
end
Xs={Gen 1 10}
Ys={Map Xs fun {$ X} X*X end}
{Browse Ys}
```

(L'appel `{Delay 100}` attend au moins 100 ms avant de continuer.) Nous pouvons en faire un programme concurrent en mettant la génération et la transformation chacune dans son propre fil :

```
thread Xs={Gen 1 10} end
thread Ys={Map Xs fun {$ X} X*X end} end
{Browse Ys}
```

L'instruction **thread** ⟨s⟩ **end** exécute ⟨s⟩ de façon concurrente. Quelle est la différence entre les versions concurrente et séquentielle ? Le résultat du calcul est le même dans les deux cas, à savoir `[1 4 9 16 ... 81 100]`. Dans la version séquentielle, `Gen` calcule toute la liste avant le début de `Map`. Le résultat final est affiché d'un coup quand le calcul est terminé, après une seconde. Dans la version concurrente, `Gen` et `Map` s'exécutent en même temps. Chaque fois que `Gen` ajoute un élément à sa liste, `Map` calcule immédiatement son carré. Le résultat est affiché au fur et à mesure que les entiers sont générés, un entier chaque dixième de seconde. Les listes `Xs` et `Ys`, calculées incrémentalement, s'appellent des **flots**.

La raison pour laquelle cette forme de concurrence est aussi simple est que les programmes n'ont pas de non-déterminisme observable. Un programme dans le modèle

déclaratif concurrent a toujours cette propriété, du moins si le programme n'essaie pas de lier la même variable à des valeurs incompatibles (voir la section 4.1). On peut dire qu'il n'y a pas de courses (situations de compétition ou « *race conditions* ») dans un programme déclaratif concurrent. Une **course** est simplement un comportement non-déterministe observable.

La structure du chapitre

Ce chapitre explique la première forme de concurrence déclarative, la concurrence dataflow (« par flux de données »), que l'on peut appeler la concurrence dirigée par l'offre (« *supply-driven concurrency* »). Il y a trois sections qui expliquent cette forme de programmation concurrente, suivies par une section qui explique les limitations de tous les modèles déclaratifs :

- La section 4.1 définit le modèle déclaratif concurrent qui étend le modèle déclaratif avec un concept, le fil. La section définit la déclarativité dans un contexte concurrent.
- La section 4.2 donne les bases de la programmation avec plusieurs fils qui communiquent par dataflow.
- La section 4.3 explique la technique la plus populaire, la communication par flots.
- La section 4.4 montre les deux limitations principales de la programmation déclarative : le manque de modularité et l'incapacité à exprimer le non-déterminisme. Ces limitations sont les motivations principales pour introduire l'état explicite dans le chapitre 5.

Faute de place, nous ne pouvons explorer les autres formes de programmation concurrente, à savoir la concurrence paresseuse (qui est déclarative aussi), la concurrence par envoi de messages et la concurrence par état partagé. Pour ces modèles, voir [97].

4.1 LE MODÈLE CONCURRENT DATAFLOW

Le modèle déclaratif du chapitre 2 est séquentiel : il y a une pile d'instructions qui s'exécute sur la mémoire à affectation unique. Nous pouvons étendre le modèle en deux étapes, en ajoutant un concept dans chaque étape :

- La première étape est la plus importante. Nous ajoutons les fils et une instruction **`thread`** ⟨s⟩ **`end`**. Un fil est simplement une instruction en exécution, c'est-à-dire, une pile sémantique. C'est tout ce dont nous avons besoin pour programmer avec la concurrence déclarative. Nous verrons que l'addition des fils garde toutes

les bonnes propriétés du modèle déclaratif. Nous appelons le modèle résultant **modèle concurrent dataflow** ou **modèle déclaratif concurrent**.

– La deuxième étape étend le modèle avec l'exécution paresseuse. Nous ajoutons la synchronisation par besoin et l'instruction `WaitNeeded`. L'appel `{WaitNeeded X}` attend jusqu'à ce qu'une opération ait besoin de `X`. Par exemple, si on fait une addition `X+Y` ou une correspondance de formes **`case X of ... end`**, le `{WaitNeeded X}` continuera son exécution. Avec `WaitNeeded` on peut définir des fonctions paresseuses, qui seront évaluées seulement si on a besoin de leur résultat. Oz a une abstraction linguistique, **`fun lazy`**, pour les fonctions paresseuses. Nous appelons le modèle résultant **modèle concurrent paresseux**. Ce modèle garde aussi les bonnes propriétés du modèle déclaratif. C'est le modèle déclaratif le plus expressif que nous connaissons.

Nous présentons uniquement le modèle concurrent dataflow. Le modèle concurrent paresseux dépasse la portée de cet ouvrage.

Dans le modèle concurrent dataflow, s'il y a une erreur pendant l'exécution il est possible que l'exécution sorte du modèle parce qu'elle n'est plus déclarative. Cela peut être traité en ajoutant des exceptions au modèle comme nous avons ajouté les exceptions au modèle déclaratif. Ainsi, on peut détecter et éventuellement réparer l'erreur pour que l'exécution reste déclarative si on le souhaite.

4.1.1 Les concepts de base

Nous étendons le modèle déclaratif avec la possibilité d'avoir plusieurs piles sémantiques qui s'exécutent « en même temps » avec une mémoire partagée. Cela donne le modèle illustré dans la figure 4.1 avec le langage noyau du tableau 4.1. Le langage noyau étend la figure 2.1 avec la nouvelle instruction **`thread`**.

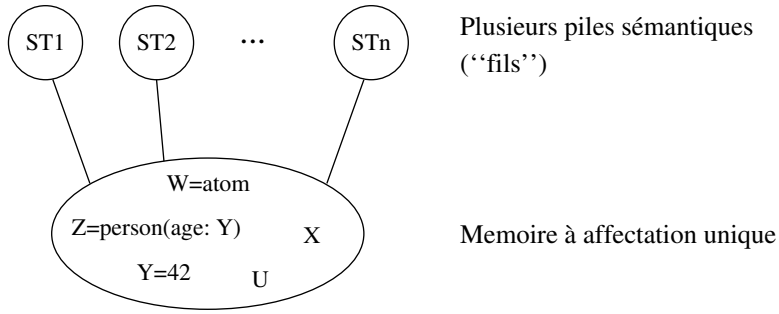


Figure 4.1 Le modèle concurrent dataflow.

$\langle s \rangle$: :=	
skip	Instruction vide
$\langle s \rangle_1$ $\langle s \rangle_2$	Séquence d'instructions
local $\langle x \rangle$ **in** $\langle s \rangle$ **end**	Création de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Lien variable-variable
$\langle x \rangle = \langle v \rangle$	Création de valeur
if $\langle x \rangle$ **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**	Instruction conditionnelle
case $\langle x \rangle$ **of** $\langle$pattern$\rangle$ **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**	Correspondance de formes
$\{\langle x \rangle\ \langle y \rangle_1 \cdots \langle y \rangle_n\}$	Application de procédure
thread $\langle s \rangle$ **end**	**Création de fil**

Tableau 4.1 Le langage noyau concurrent dataflow.

L'entrelacement

Réfléchissons un instant pour comprendre exactement ce que veut dire « en même temps ». Il y a deux façons de voir ce qui se passe ; le point de vue langage et le point de vue implémentation :

- Le point de vue langage est la sémantique du langage telle qu'elle est vue par le programmeur. De ce point de vue, l'hypothèse la plus simple est de laisser les fils faire une exécution entrelacée ; il y a une séquence globale de pas d'exécution et les fils s'exécutent à tour de rôle : chaque fil attend son tour pour exécuter quelques pas et passe ensuite le contrôle au suivant. Les pas d'exécution ne se chevauchent pas ; nous pouvons dire que chaque pas d'exécution est atomique. Le raisonnement sur les programmes est donc simplifié.
- Le point de vue implémentation est l'implémentation des fils sur une machine réelle. Si le système est implémenté sur un seul processeur, l'implémentation fera de l'entrelacement aussi. Mais le système pourrait être implémenté sur plusieurs processeurs, ce qui permettrait à des fils de faire des pas simultanément. Le parallélisme ainsi obtenu peut améliorer la performance.

Nous utilisons la sémantique d'entrelacement partout. On peut démontrer qu'avec toute exécution parallèle, il existe au moins un entrelacement qui est équivalent pour l'observation. Si nous observons la mémoire pendant l'exécution, nous pourrons toujours trouver une exécution entrelacée qui donne la même évolution de la mémoire.

L'ordre causal

La différence entre les exécutions séquentielle et concurrente peut être comprise en termes d'un ordre défini entre les états d'exécution d'un programme :

L'ordre causal des pas d'exécution

Pour un programme donné, tous les pas d'exécution forment un ordre partiel qui s'appelle l'**ordre causal**. Un pas d'exécution *arrive avant* un autre pas si dans toutes les exécutions possibles du programme il est avant l'autre. Il en va de même pour un pas d'exécution qui arrive après un autre. Parfois un pas n'est ni avant ni après un autre pas. Dans ce cas, nous disons que les deux pas sont concurrents.

Dans un programme séquentiel, tous les pas d'exécution sont dans un ordre total. Il n'y a pas de pas concurrents. Dans un programme concurrent, tous les pas d'exécution de chaque fil sont dans un ordre total. Les pas d'exécution de tout le programme sont dans un ordre partiel. Deux pas dans cet ordre partiel auront un lien causal si (1) ils sont dans le même fil, ou (2) le premier lie une variable dataflow et le deuxième a besoin de la valeur de cette variable ou (3) il y a un pas intermédiaire qui a un lien de causalité avec les deux (transitivité).

La figure 4.2 montre la différence entre les exécutions séquentielles et concurrentes. La figure 4.3 donne un exemple qui montre quelques exécutions qui correspondent à un même ordre causal. L'ordre causal a deux fils, T1 et T2, où T1 a deux opérations (I_1 et I_2) et T2 a trois opérations (I_a, I_b et I_c). Quatre exécutions possibles sont montrées. Chaque exécution respecte l'ordre causal, c'est-à-dire que toutes les instructions en ordre causal ont le même ordre dans l'exécution. Combien d'exécutions sont possibles en tout ? *Indice* : Il n'y en a pas beaucoup dans cet exemple.

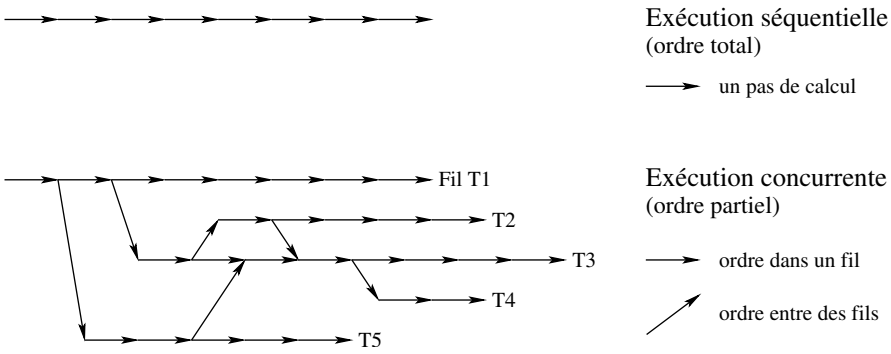


Figure 4.2 Les ordres causaux des exécutions séquentielle et concurrente.

Le non-déterminisme

Une exécution est dite non-déterministe s'il existe un état d'exécution dans lequel plusieurs fils peuvent s'exécuter. Il faut donc choisir quel fil fera le pas suivant. Ce

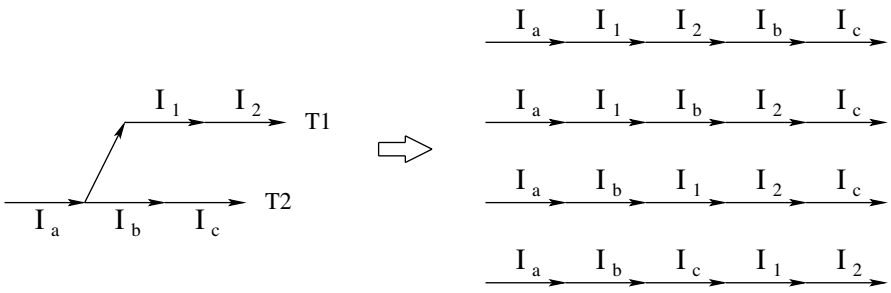


Figure 4.3 La relation entre l'ordre causal et les exécutions entrelacées.

choix s'appelle le **non-déterminisme**. Le non-déterminisme apparaît naturellement quand il y a des états concurrents. Par exemple, dans la figure 4.3, après le premier pas, qui fait toujours I_a, il y a le choix entre I_1 ou I_b pour le pas suivant.

Dans un modèle déclaratif concurrent, le non-déterminisme n'est pas visible pour le programmeur.[1] C'est à cause de deux choses. Premièrement, les variables dataflow ne peuvent être liées qu'à une valeur. Le non-déterminisme influence le moment précis où chaque lien a lieu ; il n'influence pas le fait qu'il y a un lien. Deuxièmement, toute opération qui a besoin de la valeur d'une variable n'a d'autre choix que d'attendre jusqu'à ce que la variable soit liée. Si nous permettons des opérations qui pourraient choisir d'attendre ou pas, le non-déterminisme deviendra visible.

En conséquence, un modèle déclaratif concurrent garde les bonnes propriétés du modèle déclaratif. En plus, nous allons voir que le modèle concurrent enlève certaines limitations du modèle déclaratif, mais pas toutes.

L'ordonnancement

Le choix du fil à exécuter à tout moment est fait par une partie du système qui s'appelle l'**ordonnanceur**. À chaque pas d'exécution, l'ordonnanceur choisit un fil parmi tous ceux qui sont prêts à s'exécuter. Nous disons qu'un fil est prêt, ou exécutable, si sa première instruction a toutes les informations nécessaires pour exécuter au moins un pas. Une fois qu'un fil est prêt, il reste prêt indéfiniment. Nous disons que la réduction des fils est monotone. Un fil prêt peut être exécuté à tout moment.

1. S'il n'y a pas de tentatives de lier la même variable à des valeurs partielles incompatibles. Cela s'appelle un **échec d'unification** parce que l'algorithme qui lie les variables s'appelle l'unification. En général, l'échec d'unification est une conséquence d'une erreur de programmation.

Un fil qui n'est pas prêt est dit suspendu. Sa première instruction ne peut pas continuer parce qu'elle n'a pas toutes les informations dont elle a besoin. Nous disons que la première instruction est bloquée. Le blocage est un concept important que nous verrons à d'autres occasions.

Nous disons que le système est équitable s'il ne laisse pas un fil prêt « s'affamer » ; c'est-à-dire, tous les fils prêts s'exécuteront tôt ou tard. C'est une propriété importante pour que le comportement du programme soit prévisible et pour simplifier le raisonnement sur les programmes. Elle est apparentée à la modularité : l'équité implique que l'exécution d'un fil ne dépende pas de celle d'aucun autre fil. Nous supposons que les fils sont ordonnancés équitablement.

4.1.2 La sémantique des fils

Nous étendons la machine abstraite de la section 2.4 en lui permettant de s'exécuter avec plusieurs piles sémantiques au lieu d'une seule. Chaque pile sémantique correspond au concept intuitif de « fil ». Toutes les piles sémantiques accèdent à la même mémoire. Les fils communiquent entre eux par cette mémoire partagée.

Les concepts

Nous gardons les concepts de mémoire à affectation unique σ, environnement E, instruction sémantique $(\langle \mathsf{s} \rangle, E)$ et pile sémantique ST. Nous étendons les concepts d'état d'exécution et calcul pour prendre en compte plusieurs piles sémantiques :

- Un état d'exécution est une paire (MST, σ) où MST est un multi-ensemble de piles sémantiques et σ est une mémoire à affectation unique. Un multi-ensemble est un ensemble dans lequel le même élément peut apparaître plusieurs fois. MST doit être un multi-ensemble parce que nous pouvons avoir deux piles sémantiques différentes avec les mêmes contenus, par exemple deux fils qui exécutent les mêmes instructions.
- Un calcul est une séquence d'états d'exécution qui commence avec un état initial : $(MST_0, \sigma_0) \rightarrow (MST_1, \sigma_1) \rightarrow (MST_2, \sigma_2) \rightarrow \cdots$.

L'exécution d'un programme

Comme avant, un programme est une instruction $\langle \mathsf{s} \rangle$. Voici comment l'exécuter :

- L'état d'exécution initial est

$$(\{\underbrace{[\overbrace{(\langle \mathsf{s} \rangle, \phi)}^{\text{instruction}}]}_{\text{pile}}\}, \phi)$$
$$\underbrace{\qquad\qquad}_{\text{multi-ensemble}}$$

La mémoire initiale est vide (aucune variable, $\sigma = \phi$) et l'état d'exécution initial a une pile sémantique qui contient une instruction sémantique $(\langle s \rangle, \phi)$. La seule différence avec le chapitre 2 est que la pile sémantique est dans un multi-ensemble.

– À chaque pas, une pile sémantique exécutable *ST* est sélectionnée de *MST*, ce qui laisse *MST'*. Nous pouvons dire $MST = \{ST\} \uplus MST'$. (L'opérateur $\uplus$ désigne l'union de multi-ensembles.) Un pas d'exécution est fait dans *ST* selon la sémantique du chapitre 2, ce qui donne

$$(ST, \sigma) \rightarrow (ST', \sigma')$$

Le pas d'exécution pour le calcul complet est alors

$$(\{ST\} \uplus MST', \sigma) \rightarrow (\{ST'\} \uplus MST', \sigma')$$

La sémantique est entrelacée parce qu'il y a une séquence globale de pas d'exécution. Chaque fil attend son tour pour avancer.

– Le choix de la pile *ST* à sélectionner est fait par l'ordonnanceur selon un ensemble de règles précises qui s'appelle l'**algorithme d'ordonnancement**. Cet algorithme s'assure que les propriétés désirées, comme l'équité, sont satisfaites pour tout calcul. Un ordonnanceur pratique doit prendre en compte bien plus que l'équité. La section 4.2.4 rentre dans les détails de ces propriétés pour l'ordonnanceur de Mozart.

– S'il n'y a aucune pile sémantique exécutable dans *MST*, alors le calcul ne peut pas continuer :

  - Si toutes les *ST* dans *MST* sont terminées, nous dirons que le calcul est terminé.
  - S'il existe au moins une *ST* suspendue dans *MST* qui ne peut pas être récupérée (voir plus loin), alors nous dirons que le calcul est suspendu.

L'instruction `thread`

La sémantique de l'instruction **`thread`** est définie en termes de modifications du multi-ensemble *MST*. Une instruction **`thread`** ne bloque jamais. Si la *ST* sélectionnée a la forme $[(\textbf{thread}\ \langle s \rangle\ \textbf{end}, E)] + ST'$, le nouveau multi-ensemble sera $\{[(\langle s \rangle, E)]\} \uplus \{ST'\} \uplus MST'$. En d'autres termes, nous ajouterons une nouvelle pile sémantique $[(\langle s \rangle, E)]$ qui correspond au nouveau fil. La figure 4.4 en donne une illustration. Nous résumons cette opération dans le pas d'exécution suivant :

$$(\{[(\textbf{thread}\ \langle s \rangle\ \textbf{end}, E)] + ST'\} \uplus MST', \sigma) \rightarrow (\{[(\langle s \rangle, E)]\} \uplus \{ST'\} \uplus MST', \sigma)$$

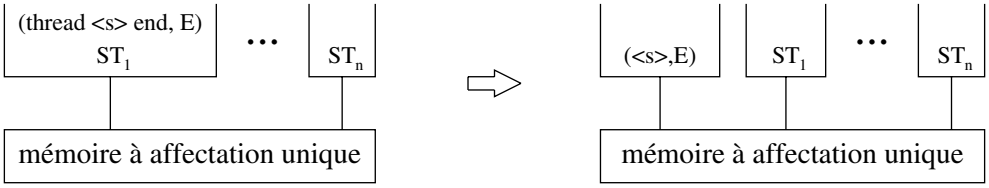


Figure 4.4 L'exécution de l'instruction **thread**.

La gestion de mémoire

La gestion de mémoire est étendue au multi-ensemble :

– Une pile sémantique terminée peut être désallouée.
– Une pile sémantique suspendue peut être récupérée si sa condition d'activation dépend d'une variable non accessible. Dans ce cas, la pile sémantique ne redeviendrait jamais exécutable, on peut donc l'enlever en toute sécurité.

L'intuition simple du chapitre 2 que « les structures de contrôle sont désallouées et les structures de données sont récupérées », n'est plus complètement vraie.

4.1.3 Un exemple d'exécution

Ce premier exemple montre comment les fils sont créés et comment ils communiquent par synchronisation dataflow. Prenons l'instruction suivante :

```
local B in
   thread B=true end
   if B then {Browse yes} end
end
```

Nous utiliserons la machine abstraite basée sur les substitutions de la section 3.3.

– Nous omettons les pas d'exécution initiaux et nous allons directement au moment où les instructions **thread** et **if** sont toutes les deux sur la pile sémantique. Cela donne

$$(\{[\textbf{thread}\ b\text{=}\textbf{true}\ \textbf{end},\ \textbf{if}\ b\ \textbf{then}\ \texttt{\{Browse yes\}}\ \textbf{end}]\},\ \{b\} \cup \sigma)$$

où b est une variable en mémoire. Il y a une pile sémantique qui contient deux instructions.

– Après l'exécution de l'instruction **thread**, nous obtenons

$$(\{[b\text{=}\textbf{true}],\ [\textbf{if}\ b\ \textbf{then}\ \texttt{\{Browse yes\}}\ \textbf{end}]\},\ \{b\} \cup \sigma)$$

Il y a maintenant deux piles sémantiques (des « fils »). La première, qui contient b=**true**, est prête. La deuxième, qui contient l'instruction **if**, est suspendue parce que la condition d'activation (b déterminée) est fausse.

– L'ordonnanceur choisit le fil prêt. Après l'exécution d'un pas, nous obtenons

$$(\{[], [\textbf{if}\ b\ \textbf{then}\ \texttt{\{Browse yes\}}\ \textbf{end}]\},\ \{b = \textbf{true}\} \cup \sigma)$$

Le premier fil est terminé (pile sémantique vide). Le deuxième fil est maintenant prêt, parce que b est déterminée.

– Nous récupérons la pile sémantique vide et nous exécutons l'instruction **if**. Cela donne

$$(\{[\texttt{\{Browse yes\}}]\},\ \{b = \textbf{true}\} \cup \sigma)$$

Il reste un fil prêt. Continuer le calcul affichera `yes`.

4.2 LA PROGRAMMATION DE BASE AVEC LES FILS

Il y a beaucoup de nouvelles techniques de programmation qui deviennent possibles dans le modèle concurrent par rapport au modèle séquentiel. Cette section examine quelques-unes des plus simples, qui sont basées sur une utilisation simple des fils. Nous étudions aussi l'ordonnanceur pour voir les opérations qui sont possibles sur les fils eux-mêmes.

4.2.1 La création des fils

L'instruction **thread** crée un nouveau fil :

```
thread
   proc {Count N} if N>0 then {Count N-1} end end
in {Count 1000000} end
```

Le nouveau fil s'exécute de façon concurrente avec le fil principal. La notation **thread** ... **end** peut aussi être utilisée comme une expression :

```
{Browse thread 10*10 end + 100*100}
```

C'est du sucre syntaxique pour :

```
local X Y in
   thread Y=10*10 end
   X=Y+100*100
   {Browse X}
end
```

Une nouvelle variable dataflow, `Y`, est créée pour la communication entre le nouveau fil et le fil principal. L'addition bloque jusqu'à ce que le calcul `10*10` soit terminé.

Quand un fil n'a plus d'instructions à exécuter, il termine. Chaque fil non terminé qui n'est pas suspendu sera exécuté tôt ou tard. Nous disons que les fils sont ordonnancés équitablement. L'exécution des fils est implémenté avec un ordonnancement préemptif. Si plusieurs fils sont prêts à s'exécuter, chaque fil aura des tranches de temps en intervalles. Il n'est pas possible pour un fil de monopoliser tout le temps du processeur.

4.2.2 Les fils et le Browser

Le Browser est un bon exemple d'un programme qui fonctionne bien dans un environnement concurrent. Par exemple :

```
thread {Browse 111} end
{Browse 222}
```

Dans quel ordre les valeurs `111` et `222` seront-elles affichées ? La réponse est : les deux ordres sont possibles ! Est-il possible que l'on affiche quelque chose comme `112122`, ou pire, que le Browser ait un comportement erroné ? À première vue, cela semble possible, parce que le Browser doit exécuter beaucoup d'instructions pour afficher les valeurs `111` et `222`. Sans précautions particulières, ces instructions peuvent effectivement être exécutées dans beaucoup d'ordres différents. Mais le Browser a été conçu pour une utilisation concurrente. Il n'affichera jamais des entrelacements étranges. Chaque appel du Browser affichera dans sa propre partie de la fenêtre. Si l'argument contient une variable non liée qui est liée plus tard, l'affichage sera mis à jour quand la variable sera liée. Ainsi, le Browser affichera correctement plusieurs flots qui s'étendront en concurrence, par exemple :

```
declare X1 X2 Y1 Y2 in
thread {Browse X1} end
thread {Browse Y1} end
thread X1=all|roads|X2 end
thread Y1=all|roams|Y2 end
thread X2=lead|to|rome|_ end
thread Y2=lead|to|rhodes|_ end
```

Les deux flots

```
all|roads|lead|to|rome|_
all|roams|lead|to|rhodes|_
```

sont affichés correctement dans deux parties différentes de la fenêtre du Browser. Nous verrons comment écrire des programmes concurrents qui se comportent correctement, comme le Browser.

4.2.3 Le calcul dataflow avec des fils

Regardons ce que nous pouvons faire en ajoutant des fils aux programmes simples. Il est important de se souvenir que chaque fil est un fil dataflow, qui suspend jusqu'à ce que les données soient disponibles.

Le comportement dataflow simple

Nous commençons par l'observation du comportement dataflow dans un calcul simple. Exécutez le programme suivant :

```
declare X0 X1 X2 X3 in
thread
Y0 Y1 Y2 Y3 in
   {Browse [Y0 Y1 Y2 Y3]}
   Y0=X0+1  Y1=X1+Y0  Y2=X2+Y1  Y3=X3+Y2
   {Browse completed}
end
{Browse [X0 X1 X2 X3]}
```

Le Browser affiche toutes les variables comme non liées. Observez ce qui se passe quand vous entrez les instructions suivantes une par une :

```
X0=0  X1=1  X2=2  X3=3
```

Après chaque instruction, le fil se réveille, exécute une addition et suspend de nouveau. Quand `X0` est liée, le fil peut exécuter `Y0=X0+1`. Il suspend de nouveau parce qu'il a besoin de la valeur de `X1` pour l'exécution de `Y1=X1+Y0`, et ainsi de suite.

Un programme déclaratif dans un contexte concurrent

Prenons un programme du chapitre 3 pour regarder comment il se comporte dans un contexte d'exécution concurrente. La boucle `ForAll` est définie ainsi :

```
proc {ForAll L P}
   case L of nil then skip
   [] X|L2 then {P X} {ForAll L2 P} end
end
```

(Nous remarquons que Oz a une abstraction linguistique, la boucle **`for`**, qui utilise `ForAll` et qui a la syntaxe suivante :

```
for X in L do {P X} end
```

La boucle **`for`** a d'autres fonctionnalités qui sont expliquées dans la documentation de Mozart [21].) Que se passe-t-il quand nous exécutons `ForAll` dans un fil ? :

```
declare L in
thread {ForAll L Browse} end
```

Si L est non liée, l'exécution suspendra tout de suite. Nous pouvons lier L dans d'autres fils :

```
declare L1 L2 in
thread L=1|L1 end
thread L1=2|3|L2 end
thread L2=4|nil end
```

Quel est le résultat final ? Est-ce différent du résultat de l'appel séquentiel `{ForAll [1 2 3 4] Browse}` ? Quel est le comportement de `ForAll` dans un contexte concurrent ?

Une fonction concurrente sur les listes

Voici une version concurrente de la fonction `Map` qui permet d'appliquer une fonction `F` à chaque membre d'une liste `Xs` :

```
fun {Map Xs F}
   case Xs of nil then nil
   [] X|Xr then thread {F X} end|{Map Xr F} end
end
```

L'instruction **thread** est utilisée comme une expression. Explorons le comportement de ce programme. Si nous exécutons l'instruction suivante :

```
declare F Xs Ys Zs
{Browse thread {Map Xs F} end}
```

un nouveau fil sera créé qui contient `{Map Xs F}`. Il suspend immédiatement dans l'instruction **case** parce que `Xs` est non liée. Si nous exécutons les instructions suivantes (sans **declare** !) :

```
Xs=1|2|Ys
fun {F X} X*X end
```

le fil principal traversera la liste, ce qui créera deux fils pour les deux premiers arguments de la liste, **thread** `{F 1}` **end** et **thread** `{F 2}` **end**, et ensuite il suspendra de nouveau sur la queue `Ys`. Enfin, l'exécution de

```
Ys=3|Zs
Zs=nil
```

crée un troisième fil avec **thread** `{F 3}` **end** et termine le calcul du fil principal. Les trois fils termineront, donnant la liste finale `[1 4 9]`. Remarquez que le résultat est le même que celui de la fonction `Map` séquentielle, sauf qu'il pourra être obtenu de façon incrémentale si l'entrée est incrémentale. La fonction `Map` séquentielle s'exécute par lots : le calcul ne donne aucun résultat avant que l'entrée complète soit donnée, c'est seulement ensuite qu'il donne le résultat complet.

Une fonction Fibonacci concurrente

Voici un programme concurrent qui utilise la stratégie de diviser pour régner pour calculer la fonction Fibonacci :

```
fun {Fib X}
   if X=<2 then 1
   else thread {Fib X-1} end + {Fib X-2} end
end
```

Ce programme est basé sur la fonction récursive séquentielle de Fibonacci ; la seule différence est que le premier appel récursif est fait dans son propre fil. Ce programme crée un nombre exponentiel de fils ! La figure 4.5 montre toutes les créations de fil et les synchronisations pour l'appel `{Fib 6}`. Ce calcul utilise huit fils en tout. Vous pouvez utiliser ce programme pour tester combien de fils votre implantation de Mozart peut créer. Par exemple, exécutez

```
{Browse {Fib 26}}
```

en observant le `Oz Panel` pour voir combien de fils sont actifs. Si `{Fib 26}` se termine trop vite, essayez un argument plus grand. Le `Oz Panel`, illustré dans la figure 4.6, est un outil dans Mozart qui donne des informations sur l'utilisation des ressources du système (temps d'exécution, utilisation de mémoire, fils, etc.). Pour démarrer le `Oz Panel`, sélectionnez l'entrée `Oz Panel` dans le menu `Oz` de l'interface interactive.

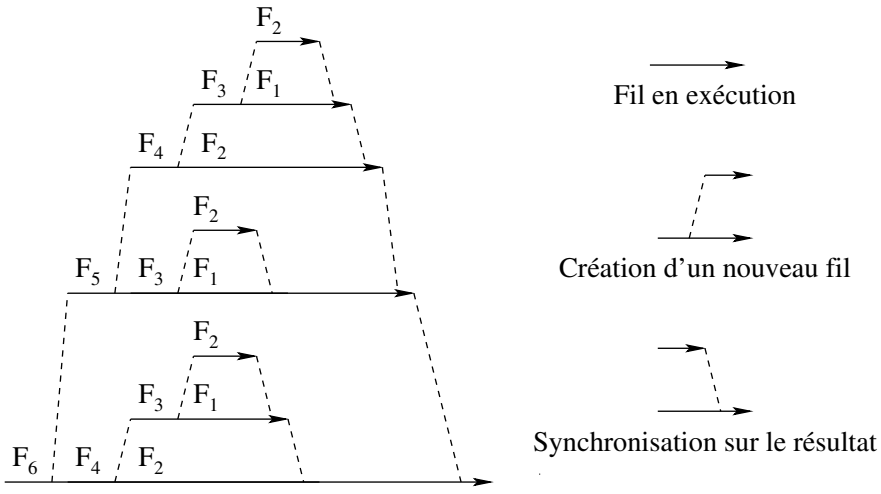


Figure 4.5 Les créations de fil pour l'appel `{Fib 6}`.

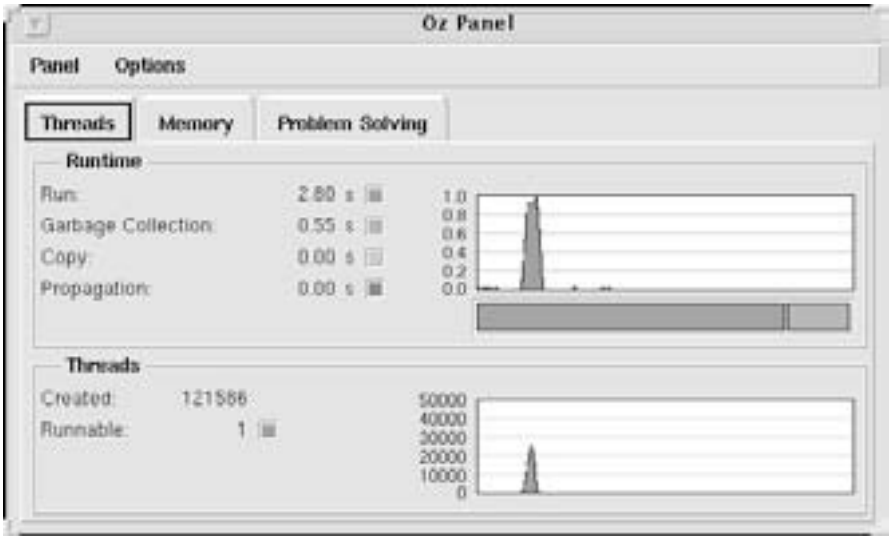


Figure 4.6 Le Panel Oz montrant la création de fils (« *threads* ») dans `X={Fib 26}`.

Le dataflow et les élastiques

Il devient clair que tout programme déclaratif du chapitre 3 peut être rendu concurrent en ajoutant **`thread ... end`** autour de quelques-unes de ses instructions et expressions. Parce que chaque variable dataflow sera liée à la même valeur qu'avant, le résultat final de la version concurrente sera exactement le même que la version séquentielle.

Une manière intuitive de comprendre pourquoi est l'analogie des élastiques. Chaque variable dataflow a son propre élastique. Un bout de l'élastique est attaché là où la variable est liée et l'autre bout là où la variable est utilisée. La figure 4.7 montre ce qui se passe dans les modèles séquentiel et concurrent. Dans le modèle séquentiel, le lien et l'utilisation sont généralement proches, l'élastique ne s'étend donc pas beaucoup. Dans le modèle concurrent, le lien et l'utilisation peuvent se faire dans des fils différents, l'élastique s'étend donc plus. Mais il ne se déchire jamais : l'utilisateur voit toujours la bonne valeur.

La concurrence bon marché et la structure du programme

En utilisant des fils, il est souvent possible d'améliorer la structure d'un programme. Par exemple, pour le rendre plus modulaire. La plupart de grands programmes contiennent beaucoup d'endroits où les fils peuvent être utilisés ainsi. Idéalement, le système sous-jacent devrait soutenir cela avec des fils qui demandent peu de ressources de calcul. À cet égard, le système Mozart est excellent. Les fils sont tellement bon marché que l'on peut les créer en grande quantité. Par exemple, un

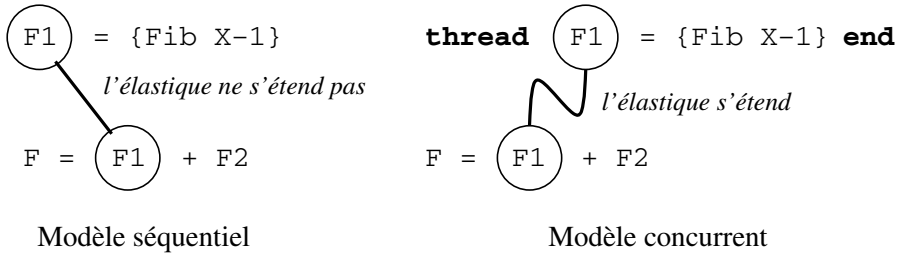


Figure 4.7 Le dataflow et les élastiques.

ordinateur personnel bon marché de l'année 2007 a typiquement au moins 512 Mo de mémoire vive, avec laquelle on peut créer plus que 100000 fils actifs simultanément.

Si l'utilisation de la concurrence permet à votre programme d'avoir une structure plus simple, il faudra l'utiliser sans hésitation. Mais il ne faut pas oublier que, même si les fils sont efficaces, les programmes séquentiels le sont encore plus. Les programmes séquentiels sont toujours plus rapides que les programmes concurrents avec la même structure. Le programme `Fib` dans la section 4.2.3 sera plus rapide si l'instruction **`thread`** est enlevée. Il ne faudra pas créer des fils si le programme n'en a pas besoin. Mais il ne faut pas hésiter à créer un fil s'il améliore la structure du programme.

4.2.4 L'ordonnancement des fils

Nous avons vu que l'ordonnanceur doit être équitable, c'est-à-dire que tout fil prêt s'exécutera tôt ou tard. Mais un vrai ordonnanceur doit faire bien plus que simplement garantir l'équité. Regardons les autres problèmes qui peuvent arriver et comment un ordonnanceur peut les résoudre.

Les tranches de temps

L'ordonnanceur met tous les fils prêts dans une file. À chaque pas, il prend le premier fil de la file, l'exécute un certain nombre de pas, et le remet dans la file. Cette technique s'appelle l'**ordonnancement cyclique** (« *round-robin scheduling* »). Elle garantit que le temps du processeur est étalé équitablement parmi les fils prêts.

Il serait inefficace de permettre à chaque fil d'exécuter un seul pas d'exécution avant de le remettre dans la file. Le surcoût pour la gestion de la file (enlever et remettre les fils) serait élevé par rapport au calcul effectif. L'ordonnanceur permet donc à chaque fil d'exécuter beaucoup de pas d'exécution avant de le remettre dans la file. Chaque fil a un temps maximum qu'il est permis d'exécuter avant qu'il soit arrêté par l'ordonnanceur. Cette durée de temps s'appelle sa **tranche de temps** ou **quantum**. Après l'épuisement de la tranche de temps d'un fil, l'ordonnanceur arrête

son exécution et le remet dans la file. L'arrêt forcé d'un fil qui s'exécute s'appelle la **préemption**.

Pour s'assurer que chaque fil obtient environ le même temps de calcul, un ordonnanceur a deux approches. La première est de compter le nombre de pas d'exécution et de donner le même nombre à chaque fil. La deuxième est d'utiliser un rythmeur qui donne le même temps à chaque fil. Les deux approches sont pratiquées. Comparons-les :

– L'approche du compteur a l'avantage que l'exécution de l'ordonnanceur est déterministe : l'exécution répétée du même programme préemptera les fils exactement aux mêmes moments. Un ordonnanceur déterministe est souvent utilisé pour les applications à temps réel dur, où il faut des garanties pour les durées des calculs.
– L'approche du rythmeur est plus efficace parce que le rythmeur est implémenté dans le matériel. Par contre, l'ordonnanceur n'est plus déterministe. Tout événement dans le système d'exploitation, comme une opération disque ou réseau, changera l'instant exact des préemptions.

Le système Mozart utilise un rythmeur.

Les niveaux de priorité

Pour beaucoup d'applications, il faut plus de contrôle sur le partage de temps entre les fils. Par exemple, pendant un calcul, un événement peut arriver qui nécessite un traitement d'urgence qui doit contourner le calcul « normal ». D'un autre côté, il ne devrait pas être possible pour des calculs urgents d'affamer des calculs normaux, causant des ralentis démesurés.

Un compromis qui marche bien en pratique est d'avoir des niveaux de priorité pour les fils. On donne à chaque niveau de priorité un pourcentage minimum du temps du processeur. À l'intérieur de chaque niveau de priorité, les fils partagent équitablement le temps comme avant. Le système Mozart utilise cette technique. Il a trois niveaux de priorité, haut (`high`), moyen (`medium`) et bas (`low`). Il y a trois files, une par niveau de priorité. Par défaut, le temps est partagé entre les priorités dans les proportions 100 :10 :1 pour les priorités haut-moyen-bas. L'implémentation est très simple : pour toutes les dix tranches de temps données à un fil à haute priorité, on donne une tranche à un fil à moyenne priorité. Pareillement, pour toutes les dix tranches de temps données à un fil à moyenne priorité, on donne une tranche à un fil à basse priorité. Les fils à haute priorité, s'il y en a, partagent au moins 100/111 (environ 90 %) du temps entre eux. Pareillement, les fils à moyenne priorité, s'il y en a, partagent au moins 10/111 (environ 9 %) du temps entre eux. Finalement, les fils à basse priorité, s'il y en a, partagent au moins 1/111 (environ 1 %) du temps entre eux. Ces pourcentages sont des bornes inférieures garanties. S'il y a moins de fils, ils peuvent être plus élevés. Par exemple, s'il n'y a pas de fils à haute priorité, alors un fil à moyenne priorité peut

recevoir jusqu'à 10/11 du temps. En Mozart, les taux haut-moyen et moyen-bas sont tous les deux à 10 par défaut. Ils sont définis avec le module `Property`.

L'héritage de priorité

Quand un fil crée un fil enfant, l'enfant a la même priorité que son parent. C'est particulièrement important pour les fils à haute priorité. Dans une application, ces fils sont utilisés pour la gestion des urgences, afin de faire le travail qui doit être fait avant le travail normal. La partie de l'application qui fait la gestion des urgences peut être concurrente. Si l'enfant d'un fil à haute priorité avait, par exemple, une priorité moyenne, il y aurait une courte « fenêtre » de temps pendant laquelle le fil enfant aurait une priorité moyenne, jusqu'à ce que le parent ou l'enfant change la priorité du fil. L'existence de cette fenêtre est suffisante pour empêcher l'ordonnancement du fil enfant pour beaucoup de tranches de temps, parce que l'enfant est dans la file de la priorité moyenne. Cela donnerait peut-être des bugs difficiles à trouver. Un fil enfant ne doit donc jamais recevoir une priorité plus basse que son parent.

La durée d'une tranche de temps

Quel est l'effet de la durée d'une tranche de temps ? Une tranche courte donne une concurrence « fine » : les fils réagissent rapidement aux événements externes. Mais si la tranche est trop courte, le coût pour changer de fil deviendra significatif. Une autre question est comment implémenter la préemption : le fil surveille-t-il lui-même son temps de calcul ou est-il fait à l'extérieur ? Les deux possibilités sont viables, mais la deuxième est bien plus facile à réaliser. Les systèmes d'exploitation multitâches modernes, comme Unix, Windows XP/Vista ou Mac OS X, ont des dispositifs d'interruption régulière (des rythmeurs) que l'on peut utiliser pour déclencher la préemption. Ces interruptions arrivent à une basse fréquence, 60 ou 100 fois par seconde. Le système Mozart utilise cette technique.

Une tranche de temps de 10 ms peut apparaître assez courte, mais pour certaines applications c'est trop long. Par exemple, supposez que l'application ait 100 000 fils actifs. Chaque fil aura une tranche de temps toutes les 1 000 secondes. Cette attente peut être trop longue. En pratique, nous constatons que ce n'est pas un problème. Dans les applications avec beaucoup de fils, comme de grands programmes à contraintes, les fils sont fortement dépendants l'un de l'autre et pas du monde externe. Chaque fil n'utilisera donc qu'une petite partie de sa tranche avant de la céder à un autre fil.

Par contre, il est possible d'imaginer une application avec beaucoup de fils, dont chacun interagit avec le monde externe indépendamment des autres. Pour une telle application, il est clair que Mozart, tout comme les systèmes d'exploitation récents, Unix, Windows ou Mac OS X, est insatisfaisant. La racine du problème est plus profonde : c'est le matériel (y compris le processeur et le système de mémoire) d'un ordinateur personnel qui est insatisfaisant. Ce qu'il faut est un ordinateur à temps réel

dur qui utilise un matériel spécialisé avec un système d'exploitation spécialisé. Le temps réel dur dépasse la portée de cet ouvrage.

4.2.5 La concurrence coopérative et compétitive

Les fils sont faits pour la concurrence coopérative et non pour la concurrence compétitive. La concurrence coopérative concerne des entités qui travaillent ensemble pour atteindre un but commun. Les fils soutiennent cela. Par exemple tout fil peut changer les taux entre les trois priorités, comme nous le verrons. Les fils sont faits pour des applications qui s'exécutent dans un environnement où toutes les parties se font confiance.

Par contre, la concurrence compétitive concerne des entités dont chacune a un but local : elles travaillent pour elles-mêmes. Elles ne sont intéressées que par leur propre performance, pas par la performance globale. La concurrence compétitive est normalement gérée par le système d'exploitation, par un concept qui s'appelle un **processus**.

Cela veut dire que les calculs ont souvent une structure à deux niveaux, comme illustré dans la figure 4.8. Au plus haut niveau, il y a un ensemble de processus du système d'exploitation qui interagissent en faisant de la concurrence compétitive. Les processus appartiennent généralement à des applications différentes, avec des buts différents et peut-être conflictuels. Dans chaque processus, il y a un ensemble de fils qui interagissent en faisant de la concurrence coopérative. Les fils d'un même processus appartiennent généralement à la même application.

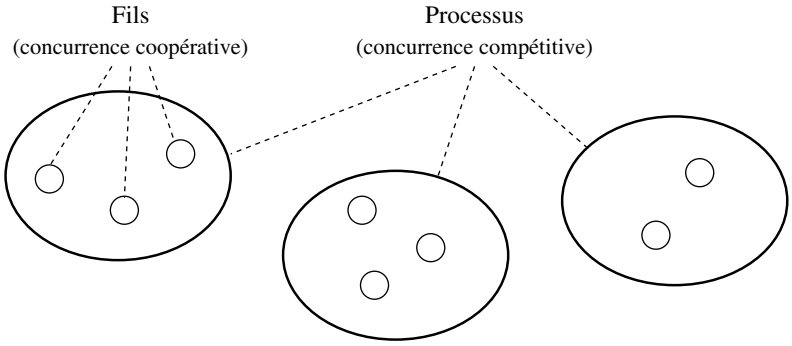


Figure 4.8 La concurrence coopérative et compétitive.

La concurrence compétitive est soutenue en Mozart par son modèle de calcul réparti et par le module `Remote`. Le module `Remote` crée un nouveau processus du système d'exploitation avec ses propres ressources calculatoires. Un calcul compétitif peut alors être créé (avec l'édition de ses liens) dans ce processus. C'est assez facile à programmer parce que le modèle réparti est transparent par rapport au réseau : le

même programme peut s'exécuter dans différentes structures réparties sur différents ensembles de processus, et il donnera toujours le même résultat.[2]

4.3 LES FLOTS

La technique la plus courante dans le modèle déclaratif concurrent est l'utilisation des flots. Un flot (« *stream* » en anglais) est une liste potentiellement illimitée de messages, plus précisément c'est une liste dont la queue est une variable dataflow non liée. L'envoi d'un message est fait par l'extension du flot d'un élément : la queue du flot est liée à une paire de liste qui contient le message et une nouvelle queue non liée. La réception d'un message revient à lire un élément du flot. Un fil qui communique avec des flots est une sorte d'« objet actif » que nous appellerons un **objet à flots**. Aucun verrou ou exclusion mutuelle n'est nécessaire parce que chaque variable est liée par un seul fil.

La programmation par flots est une approche générale que l'on peut utiliser dans beaucoup de domaines. C'est le concept qui est à la base des tuyaux Unix (« *pipes* »). Morrison l'utilise à bon escient dans des applications industrielles, avec une approche qu'il appelle « *flow-based programming* » [68]. Nous étudions un cas spécial de la programmation par flots, la programmation par flots déterministe, dans lequel chaque objet à flots sait toujours d'où viendra le message suivant. Ce cas est intéressant parce qu'il est déclaratif. Il est déjà très utile. Nous n'étudierons pas ici la programmation par flots non-déterministe.[3]

4.3.1 Le producteur/consommateur

Nous expliquons d'abord comment fonctionnent les flots et ensuite comment programmer un producteur/consommateur asynchrone avec des flots. Un flot est une liste dont la queue est une variable non liée :

```
declare Xs Xs2 in
Xs=0|1|2|3|4|Xs2
```

Un flot est construit incrémentalement en liant la queue à un autre flot :

```
declare Xs3 in
Xs2=5|6|7|Xs3
```

Un fil, que l'on appelle le **producteur**, crée le flot de cette manière, et d'autres fils, appelés les **consommateurs**, lisent le flot. Parce que la queue du flot est une

2. C'est exactement vrai si aucun processus n'échoue. Pour des exemples et plus d'informations voir [97].
3. La programmation par flots non-déterministe est expliquée dans [97].

variable dataflow, les consommateurs liront le flot au fur et à mesure de sa création. Le programme suivant crée de façon asynchrone un flot d'entiers et les additionne :

```
fun {Generate N Limit}
   if N<Limit then N|{Generate N+1 Limit}
   else nil end
end

fun {Sum Xs A}
   case Xs of X|Xr then {Sum Xr A+X}
   [] nil then A end
end

local Xs S in
  thread Xs={Generate 0 150000} end % Fil producteur
  thread S={Sum Xs 0} end           % Fil consommateur
  {Browse S}
end
```

La figure 4.9 montre une façon particulièrement jolie de présenter cette technique avec une notation graphique précise. Chaque rectangle représente une fonction récursive dans un fil, la flèche représente un flot et la direction de la flèche va du producteur au consommateur. À la fin du calcul, la somme `11249925000` est affichée. Le producteur, `Generate`, et le consommateur, `Sum`, s'exécutent dans leurs propres fils. Ils communiquent par la variable partagée `Xs`, qui est liée à un flot d'entiers. L'instruction **`case`** dans `Sum` bloque quand `Xs` est non liée (plus d'éléments) et reprend quand `Xs` est liée (de nouveaux éléments arrivent).

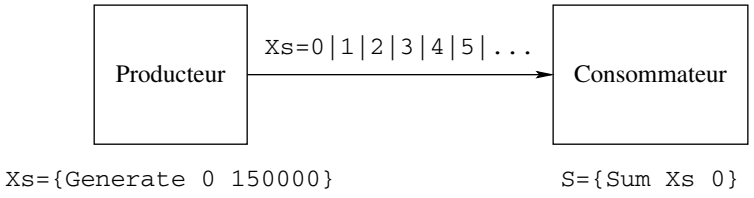


Figure 4.9 La communication par flot producteur/consommateur.

Dans le consommateur, le comportement dataflow de l'instruction **`case`** bloque l'exécution jusqu'à l'arrivée de l'élément suivant du flot. Cela synchronise le consommateur avec le producteur. Attendre le lien d'une variable dataflow est le mécanisme de base pour la synchronisation et la communication dans le modèle concurrent dataflow.

L'utilisation d'un itérateur d'ordre supérieur

L'appel récursif de `Sum` a un argument `A` qui est la somme de tous les éléments vus à un moment donné. Cet argument et le résultat de la fonction pris ensemble font un accumulateur, comme nous avons vu dans la section 3.4.3. Nous pouvons éliminer l'accumulateur en utilisant une abstraction de boucle :

```
local Xs S in
   thread Xs={Generate 0 150000} end
   thread S={FoldL Xs fun {$ X Y} X+Y end 0} end
   {Browse S}
end
```

La fonction `FoldL` fait une itération et accumule le résultat. À cause des variables dataflow, `FoldL` fonctionne bien dans un contexte concurrent sans nécessiter aucun changement. L'élimination d'un accumulateur en utilisant un itérateur d'ordre supérieur est une technique générale. L'accumulateur n'est pas vraiment éliminé, mais caché dans l'itérateur. Mais le programme est plus simple parce que le programmeur ne doit plus raisonner avec un état. Le module `List` a encore d'autres abstractions de boucle et opérations d'ordre supérieur qui peuvent souvent remplacer les fonctions récursives [23].

Plusieurs lecteurs

Nous pouvons utiliser plusieurs consommateurs sans changer le programme. Par exemple, voici trois consommateurs qui lisent le même flot :

```
local Xs S1 S2 S3 in
   thread Xs={Generate 0 150000} end
   thread S1={Sum Xs 0} end
   thread S2={Sum Xs 0} end
   thread S3={Sum Xs 0} end
end
```

Le fil de chaque consommateur recevra les éléments du flot indépendamment des autres. Les consommateurs ne se gênent pas mutuellement parce qu'ils ne « consomment » pas vraiment le flot ; ils le lisent sans le perturber.

4.3.2 Les transformateurs et les pipelines

Nous pouvons intercaler un troisième objet à flots entre le producteur et le consommateur. Il lit le flot du producteur et crée un autre flot qui est lu par le consommateur. Nous l'appelons un **transformateur**. En général, une séquence d'objets à flots dont chacun alimente le suivant s'appelle un **pipeline**. Regardons quelques pipelines avec différentes sortes de transformateurs.

Le filtrage d'un flot

Un transformateur simple est le filtre qui transmet les éléments du flot d'entrée qui satisfont une condition donnée. Une manière simple de construire un filtre est d'appeler la fonction `Filter` dans son propre fil. Cette fonction prend une liste d'entrée et une fonction booléenne et calcule une liste de sortie qui ne contient les éléments de la liste d'entrée qui satisfont la fonction. Voici un transformateur qui ne transmet que les éléments qui sont des entiers impairs :

```
local Xs Ys S in
   thread Xs={Generate 0 150000} end
   thread Ys={Filter Xs IsOdd} end
   thread S={Sum Ys 0} end
   {Browse S}
end
```

où `IsOdd` est une fonction booléenne d'un argument qui est vraie uniquement pour les entiers impairs :

```
fun {IsOdd X} X mod 2 \= 0 end
```

La figure 4.10 montre cette technique. Cette figure étend la notation graphique avec une flèche pointillée, qui représente une valeur qui n'est pas un flot.

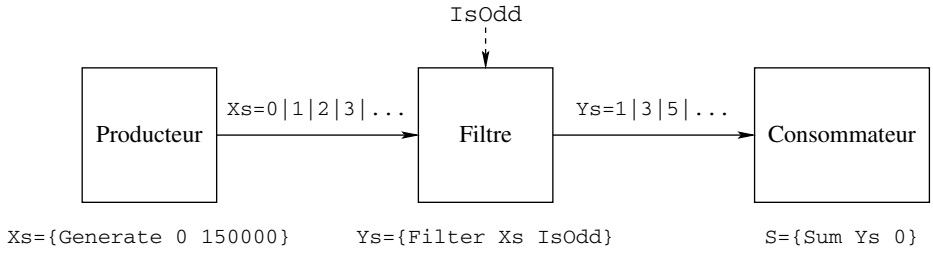


Figure 4.10 Le filtrage d'un flot.

Le crible d'Ératosthène

Nous définissons un pipeline qui implémente le crible d'Ératosthène, un algorithme pour générer des nombres premiers. La sortie du crible est un flot qui ne contient que des nombres premiers. Le programme s'appelle un « crible » parce qu'il fait des filtrages successifs qui enlèvent les nombres non premiers des flots, pour ne laisser que des nombres premiers. Les filtres sont créés dynamiquement quand on en a besoin. Le producteur génère un flot d'entiers consécutifs qui commence par 2. Le crible prend le premier élément du flot et crée un filtre pour enlever les multiples de cet élément.

Il s'appelle alors récursivement sur le flot d'éléments restants. La figure 4.11 donne une illustration. Cette figure étend la notation graphique avec un autre élément, le triangle, qui représente la décomposition ou la construction d'un flot à partir d'un premier élément et un autre flot. Voici la définition du crible (« *sieve* » en anglais) :

```
fun {Sieve Xs}
   case Xs of nil then nil
   [] X|Xr then Ys in
      thread Ys={Filter Xr
                 fun {$ Y} Y mod X \= 0 end} end
      X|{Sieve Ys}
   end
end
```

Cette définition est assez simple, étant donné qu'elle installe un pipeline d'activités concurrentes. Voici un appel du crible :

```
local Xs Ys in
   thread Xs={Generate 2 100000} end
   thread Ys={Sieve Xs} end
   {Browse Ys}
end
```

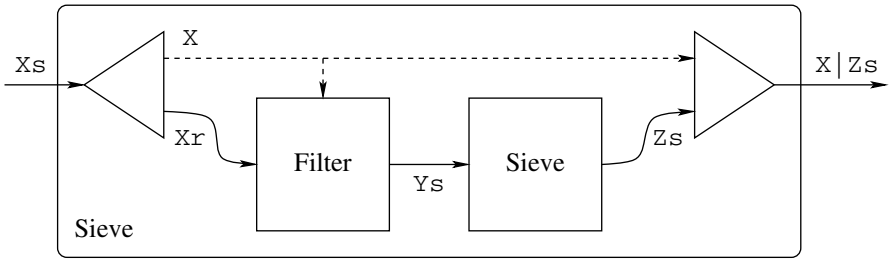


Figure 4.11 Un crible de nombres premiers implémenté avec des flots.

Ce programme affiche les nombres premiers jusqu'à 100 000. Il est un peu simpliste parce qu'il crée trop de fils, à savoir un par nombre premier. Un nombre de fils tellement grand n'est pas nécessaire. Il est facile de voir que la génération de nombres premiers jusqu'à n ne nécessite le filtrage de multiples que jusqu'à $\sqrt{n}$.[4] Nous modifions le programme pour arrêter la création des filtres après cette limite :

4. Si le facteur f est plus grand que $\sqrt{n}$, il y aura un autre facteur n/f plus petit que $\sqrt{n}$.

```
fun {Sieve Xs M}
   case Xs of nil then nil
   [] X|Xr then Ys in
      if X=<M then
         thread Ys={Filter Xr
                    fun {$ Y} Y mod X \= 0 end} end
      else Ys=Xr end
      X|{Sieve Ys M}
   end
end
```

Avec une liste de 100 000 éléments, nous l'appelons comme `{Sieve Xs 316}` (parce que $316 = \lfloor\sqrt{100\,000}\rfloor$). Cela crée le pipeline de filtres montré dans la figure 4.12. Parce que les petits facteurs sont plus fréquents que les grands, l'essentiel du travail de filtrage sera fait au début du pipeline.

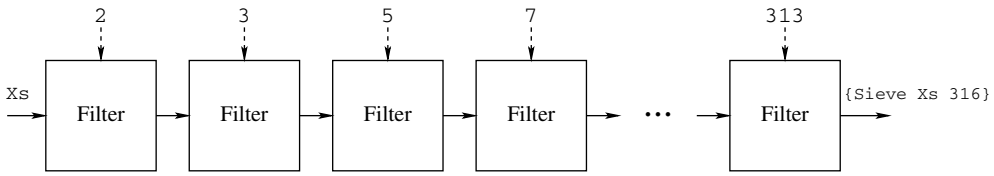


Figure 4.12 Le pipeline de filtres généré par `{Sieve Xs 316}`.

4.4 LES PRINCIPALES LIMITATIONS DE LA PROGRAMMATION DÉCLARATIVE

Cette section explique les deux principales limitations de la programmation déclarative, relatives à la modularité et au non-déterminisme.

4.4.1 La modularité

Nous disons qu'un programme est modulaire par rapport à un changement d'une partie si le changement peut être fait sans changer le reste du programme. La modularité ne peut pas être réalisée dans un modèle déclaratif, mais elle pourra être réalisée si le modèle est étendu avec l'état explicite. Voici deux exemples où les programmes déclaratifs ne sont pas modulaires :

1. Le premier exemple est une mémoire cache de mémoisation. C'est une mémoire à l'intérieur d'une fonction qui garde un ensemble de paires d'arguments et leurs résultats. À l'appel, si l'argument est dans la mémoire cache le résultat pourra

être obtenu sans aucun calcul. L'ajout d'une telle mémoire cache à une fonction n'est pas modulaire parce qu'un accumulateur doit être enfilé à l'extérieur de la fonction. L'accumulateur contient le contenu actuel de la mémoire cache.

2. Le deuxième exemple est l'instrumentation d'un programme. Nous voulons savoir combien de fois certains sous-composants sont appelés. Nous voulons ajouter des compteurs à ces sous-composants, de préférence sans changer leurs interfaces ou le reste du programme. Si le programme est déclaratif, c'est impossible car la seule solution est d'enfiler des accumulateurs dans toutes les procédures qui appellent les sous-composants.

Regardons de plus près le deuxième exemple pour comprendre exactement pourquoi le modèle déclaratif est insuffisant. Supposons que nous utilisons le modèle déclaratif pour implémenter un grand composant. Voici la définition du composant :

```
fun {SC ...}
   proc {P1 ...} ...  end
   proc {P2 ...} ...  {P1 ...} {P2 ...} end
   proc {P3 ...} ...  {P2 ...} {P3 ...} end
in ´export´(p1:P1 p2:P2 p3:P3) end
```

Un appel de `SC` instancie le composant : il renvoie un module avec trois opérations, `P1`, `P2` et `P3`. Nous voulons instrumenter le composant pour compter le nombre d'appels de la procédure `P1`. Les valeurs successives du compte forment un état. Nous pouvons coder cet état comme un accumulateur, en ajoutant deux arguments à chaque procédure. Avec cette instrumentation, la définition du composant ressemblerait à ceci :

```
fun {SC ...}
   proc {P1 ... S1 ?Sn} Sn=S1+1 ... end
   proc {P2 ... T1 ?Tn} ...
      {P1 ... T1 T2} {P2 ... T2 Tn} end
   proc {P3 ... U1 ?Un} ...
      {P2 ... U1 U2} {P3 ... U2 Un} end
in ´export´(p1:P1 p2:P2 p3:P3) end
```

Chaque procédure définie par `SC` a une nouvelle interface avec deux arguments supplémentaires. La procédure `P1` est appelée comme `{P1 ... Sin Sout}`, où `Sin` est le compte à l'entrée et `Sout` est le compte à la sortie. L'accumulateur doit être enfilé entre les appels des procédures. Le module `SC` et le module qui l'appelle doivent tous les deux faire ce travail administratif.

Une autre solution est d'écrire `SC` dans un modèle avec état. Un tel modèle est défini dans le chapitre 5 ; pour le moment supposons que nous avons une nouvelle entité du langage, appelée « cellule », que nous pouvons lire et écrire (avec les opérateurs

@ et :=), comme une variable affectable dans les langages impératifs. Les cellules ont été introduites dans le chapitre 1. Avec les cellules, la définition du composant ressemblerait à ceci :

```
fun {SC ...}
   Ctr={NewCell 0}
   proc {P1 ...} Ctr:=@Ctr+1 ... end
   proc {P2 ...} ... {P1 ...} {P2 ...} end
   proc {P3 ...} ... {P2 ...} {P3 ...} end
   fun {Count} @Ctr end
in ´export´(p1:P1 p2:P2 p3:P3 count:Count) end
```

Dans cette solution, l'interface du composant a une fonction supplémentaire, `Count`, et les interfaces à `P1`, `P2` et `P3` sont inchangées. Le module qui appelle `SC` n'a aucun travail administratif à faire. Le compte est automatiquement initialisé à zéro quand le composant est instancié. On peut appeler `Count` à tout moment pour obtenir la valeur actuelle du compte. On peut aussi ignorer `Count` complètement, et dans ce cas le composant a exactement le même comportement qu'avant (sauf une toute petite différence de performance).

La figure 4.13 compare les deux approches. La figure montre le graphe d'appels d'un programme avec un composant `Main` qui appelle le sous-composant `SC`. Un graphe d'appels est un graphe orienté dans lequel chaque nœud représente une procédure et il y a une arête entre chaque procédure et les procédures qu'elle appelle. Dans la figure 4.13, `SC` est appelée à partir de trois endroits dans le composant principal.

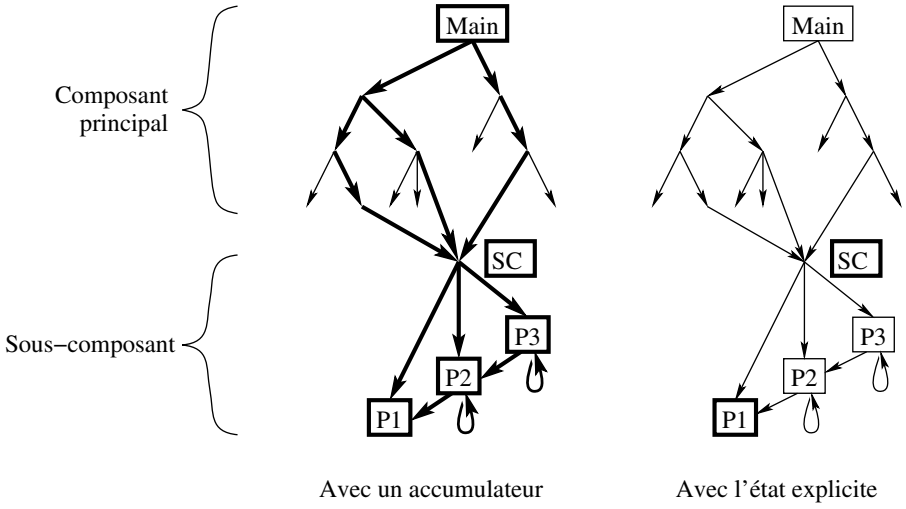


Figure 4.13 Les changements nécessaires pour compter les appels de la procédure `P1`.

Maintenant nous instrumentons `SC`. Dans l'approche déclarative (à gauche), il faut ajouter un accumulateur à chaque procédure sur le chemin de `Main` à `P1`. Dans l'approche à état (à droite), les seuls changements sont l'opération supplémentaire `Count` et le corps de `P1`. Dans les deux cas, les changements sont montrés avec des lignes épaisses. Comparons les deux approches :

- L'approche déclarative n'est pas modulaire par rapport à l'instrumentation de `P1`, parce que chaque définition et appel de procédure sur le chemin de `Main` à `P1` a besoin de deux arguments supplémentaires. Les interfaces à `P1`, `P2` et `P3` sont toutes changées. Les autres composants qui appellent `SC` doivent donc aussi être changés.
- L'approche à état est modulaire parce que la cellule est mentionnée uniquement où on en a besoin, dans l'initialisation de `SC` et dans `P1`. En particulier, les interfaces à `P1`, `P2` et `P3` sont inchangées. Comme l'opération supplémentaire `Count` peut être ignorée, les autres composants qui appellent `SC` ne doivent pas être modifiés.
- L'approche déclarative est plus lente parce qu'elle fait beaucoup de passages d'argument supplémentaires. Toutes les procédures sont ralenties à cause d'une seule. L'approche à état est efficace ; elle ne dépense du temps que lorsque c'est nécessaire pour incrémenter le compteur.

Quelle approche est la plus simple : la première ou la deuxième ? La première a un modèle plus simple mais un programme plus compliqué. La deuxième a un modèle plus compliqué mais un programme plus simple. De notre point de vue, l'approche déclarative n'est pas naturelle. L'approche à état, parce qu'elle est modulaire, est clairement la plus simple dans l'ensemble.

Le faux raisonnement du préprocesseur

Peut-être existe-t-il un moyen d'avoir les deux. Définissons un préprocesseur qui ajoute les arguments à notre place. Un préprocesseur est un programme qui prend le code source d'un autre programme comme entrée, le transforme selon certaines règles et renvoie le résultat. Nous définissons un préprocesseur qui prend la syntaxe de l'approche à état comme entrée et qui la transforme pour ressembler à l'approche déclarative. Voilà ! Il semble que nous pouvons maintenant programmer avec l'état dans le modèle déclaratif. Nous avons surmonté une limitation du modèle déclaratif. Mais l'avons-nous vraiment fait ? En réalité tout ce que nous avons réussi est de construire une implémentation inefficace d'un modèle avec état. Voici pourquoi :

- Avec le préprocesseur, nous ne voyons que les programmes qui ressemblent à l'approche à état, donc des programmes à état. Cela nous oblige à raisonner dans un modèle avec état. Nous avons donc de ce fait étendu le modèle déclaratif avec l'état explicite.

– Le préprocesseur transforme les programmes à état en programmes déclaratifs avec un état enfilé, qui sont inefficaces à cause de tous les arguments supplémentaires.

Il faut donc faire très attention à bien séparer les différents niveaux d'abstraction.

4.4.2 Le non-déterminisme

Le modèle déclaratif concurrent apparaît assez puissant pour faire des programmes concurrents. Par exemple, nous pouvons facilement construire un simulateur pour les circuits digitaux (voir [97]). Mais malgré cette puissance apparente, le modèle a une limitation qui le handicape fortement pour beaucoup d'applications concurrentes : il se comporte toujours de façon déterministe. Si un programme a un non-déterminisme observable, il ne sera pas déclaratif. Cette limitation est apparentée à la modularité : pour des composants qui sont vraiment indépendants, chacun se comporte de façon non-déterministe par rapport à l'autre. Pour montrer que ce n'est pas purement une limitation théorique, nous donnons deux exemples réalistes : une application client-serveur et une application d'affichage vidéo.

La limitation peut être enlevée par l'ajout d'une opération non-déterministe au modèle. Le modèle étendu n'est plus déclaratif. Il y a beaucoup d'opérations non-déterministes que l'on pourrait ajouter. Voici un résumé des possibilités les plus courantes :

– Une première solution est d'ajouter une opération d'attente non-déterministe, comme la fonction `{WaitTwo X Y}` qui attend jusqu'à ce que `X` ou `Y` soit déterminée, et qui pourra renvoyer 1 si `X` est déterminée et 2 si `Y` est déterminée. Si les deux sont déterminées, la fonction renverra une des deux valeurs. Sa définition est donnée dans le fichier de suppléments sur le site Web du livre. `WaitTwo` est pertinente pour l'application client/serveur.
– Une deuxième solution est d'ajouter la fonction booléenne `{IsDet X}`, qui teste si une variable dataflow `X` est déterminée ou pas et renvoie tout de suite **`true`** ou **`false`** sans attendre. Avec cette opération on peut utiliser les variables dataflow comme une forme faible d'état. `IsDet` est pertinente pour l'application d'affichage vidéo.
– Une troisième solution est d'ajouter l'état explicite au modèle, par exemple des cellules (variables affectables) ou des ports (canaux de communication).

Quelles sont les relations entre les trois solutions ? `WaitTwo` peut être programmée dans le modèle déclaratif concurrent avec l'état explicite. L'état explicite peut être programmé avec `IsDet` mais sans garder la complexité temporelle. Il semble donc que le modèle le plus expressif a besoin uniquement d'état explicite et `IsDet`.

a) Une application client/serveur

Nous construisons maintenant une application client/serveur simple. Supposons qu'il y ait deux clients indépendants. Être indépendants implique qu'ils sont concurrents. Que se passe-t-il s'ils communiquent avec le même serveur ? Parce que s'ils sont indépendants, le serveur peut recevoir des informations des deux clients dans n'importe quel ordre. C'est un comportement non-déterministe observable.

Regardons de plus près pour voir si nous pouvons exprimer ce comportement dans le modèle déclaratif concurrent. Le serveur a un flot d'entrée dont il lit ses commandes. Commençons avec un client qui envoie des commandes au serveur. Cela fonctionne parfaitement. Comment un deuxième client peut-il se connecter au serveur ? Il doit obtenir une référence à un flot qu'il peut lier et qui est lu par le serveur. Le problème est qu'un tel flot n'existe pas ! Il n'y a qu'un flot, entre le premier client et le serveur. Le deuxième client ne peut pas lier ce flot, puisque cela causera des conflits avec les liens du premier client.

Comment pouvons-nous résoudre ce problème ? Faisons une approche naïve pour voir si nous pouvons trouver une solution. Une approche pourrait être de permettre le serveur d'avoir deux flots d'entrée, comme ceci :

```
fun {Server InS1 InS2}
   ...
end
```

Mais comment le serveur lit-il ces flots ? Doit-il d'abord lire un élément de `InS1` et ensuite un élément de `InS2` ? Doit-il lire simultanément un élément des deux flots ? Aucune des deux solutions n'est correcte. En fait, il n'est pas possible d'écrire une solution dans le modèle déclaratif concurrent. La seule chose que nous pouvons faire est d'avoir deux serveurs indépendants, un par client. Mais ces serveurs ne peuvent pas communiquer entre eux, sinon nous aurions de nouveau le même problème.

La figure 4.14 montre le problème : `InS` est le flot d'entrée du serveur et `OutS1` et `OutS2` sont les deux flots de sortie des clients. Comment les messages qui apparaissent sur les deux flots clients peuvent-ils être donnés au serveur ? La réponse simple est que dans le modèle déclaratif concurrent, ils ne peuvent pas ! Dans ce modèle, un objet à flots doit toujours savoir de quel flot il lit son prochain message.

Comment pouvons-nous résoudre ce problème ? Si les clients s'exécutent de façon coordonnée et si le serveur sait toujours quel client va envoyer la commande suivante, le programme sera déclaratif. Mais ce n'est pas réaliste. Pour écrire une vraie solution, nous devons ajouter une opération non-déterministe au modèle, comme l'opération `WaitTwo` mentionnée ci-dessus. Avec `WaitTwo`, le serveur peut attendre une commande de l'un ou l'autre client.

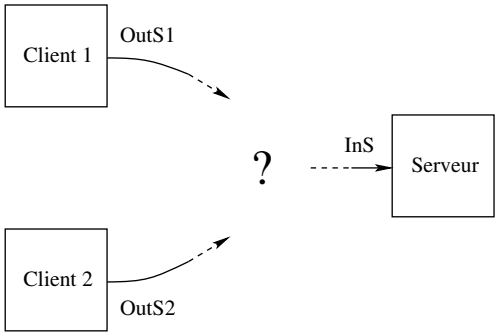


Figure 4.14 Les deux clients peuvent-ils parler au serveur ? Ils ne peuvent pas !

b) Une application d'affichage vidéo

Nous étudions une application simple d'affichage vidéo. Elle contient un composant d'affichage qui reçoit un flot d'images vidéo et les affiche sur un écran. Un certain nombre d'images par seconde arrivent au composant. Pour des raisons diverses, leur taux peut varier : les images ont des résolutions différentes, elles ont besoin d'un traitement numérique ou le réseau de transmission a une bande passante et une latence variables.

À cause du taux variable d'arrivée des images, on ne peut pas toujours afficher toutes les images. On doit parfois en laisser tomber. Par exemple, il faut parfois sauter rapidement jusqu'à la dernière image envoyée. Cette manière de gérer le flot ne peut pas être faite dans le modèle déclaratif concurrent, parce qu'il n'y a aucune manière de détecter la fin courante d'un flot. On pourra le faire si on étend le modèle avec l'opération `IsDet`. Avec `IsDet` nous pouvons définir la fonction `Skip` qui prend un flot et qui renvoie sa queue non liée :

```
fun {Skip Xs}
   if {IsDet Xs} then
      case Xs of _|Xr then {Skip Xr} [] nil then nil end
   else Xs end
end
```

`Skip` traverse le flot jusqu'à ce qu'elle trouve une queue non liée. Voici une version légèrement différente qui attend toujours d'avoir au moins un élément dans le flot :

```
fun {Skip1 Xs}
   case Xs of X|Xr then
      if {IsDet Xr} then {Skip1 Xr} else Xs end
   [] nil then nil end
end
```

Avec `Skip1`, nous pouvons construire un afficheur vidéo qui, après avoir affiché une image, saute jusqu'à la dernière image transmise :

```
proc {Display Xs}
   case {Skip1 Xs} of X|Xr then
      {DisplayFrame X} {Display Xr}
   [] nil then skip end
end
```

Cette solution fonctionnera bien même s'il y a des variations dans le taux d'arrivée des images et dans le temps nécessaire pour afficher une image.

4.5 EXERCICES

➤ **Exercice 1** — *La sémantique des fils*

Considérez la variation suivante de l'instruction utilisée dans la section 4.1.3 pour illustrer la sémantique des fils :

```
local B in
   thread B=true end
   thread B=false end
   if B then {Browse yes} end
end
```

Pour cet exercice, faites les choses suivantes :

a) Énumérez toutes les exécutions possibles de cette instruction.

b) Elles provoquent toutes une terminaison anormale du programme. Faites un petit changement au programme pour éviter ces terminaisons anormales.

➤ **Exercice 2** — *Les fils et le ramassage de miettes*

Cet exercice examine comment le ramassage de miettes se comporte avec les fils et les variables dataflow. Considérez le programme suivant :

```
proc {B _} {Wait _} end

proc {A}
   Collectible={NewDictionary}
in {B Collectible} end
```

L'opération `{Wait X}` suspend le fil courant jusqu'à ce que `X` soit déterminée. Après l'appel `{A}`, la mémoire occupée par `Collectible` est-elle récupérable ? Donnez une réponse en réfléchissant sur la sémantique. Vérifiez ensuite que le système se comporte ainsi.

➤ **Exercice 3** — *Fibonacci concurrente et séquentielle*
Prenez cette définition séquentielle de la fonction Fibonacci :

```
fun {Fib X}
   if X=<2 then 1 else {Fib X-1}+{Fib X-2} end
end
```

et comparez-la avec la définition concurrente de la section 4.2.3. Exécutez-les toutes les deux et comparez leurs performances. Quelle est la différence en temps d'exécution entre les deux ? Combien de fils sont créés par l'appel concurrent `{Fib N}` en fonction de `N` ?

➤ **Exercice 4** — *L'opération* `Wait`
Expliquez pourquoi l'opération `{Wait X}` peut être définie comme ceci :

```
proc {Wait X}
   if X==unit then skip else skip end
end
```

Utilisez vos connaissances du comportement dataflow de l'instruction **if** et de l'opération `==`.

➤ **Exercice 5** — *L'ordonnancement des fils*
La section 4.4.2 montre comment sauter au-delà des éléments déjà calculés d'un flot. Si nous utilisons cette technique pour additionner les éléments du flot d'entiers de la section 4.3.1, le résultat sera beaucoup plus petit que 11249925000, la somme de tous les entiers du flot. Expliquez pourquoi en utilisant vos connaissances de l'ordonnancement des fils.

➤ **Exercice 6** — *Le comportement dataflow dans un contexte concurrent*
Considérez la fonction `{Filter In F}` qui renvoie les éléments de `In` pour lesquels la fonction booléenne `F` renvoie **true**. Voici une définition possible de `Filter` :

```
fun {Filter In F}
   case In of X|In2 then
      if {F X} then X|{Filter In2 F}
      else {Filter In2 F} end
   else nil end
end
```

L'exécution de l'instruction suivante :

```
{Show {Filter [5 1 2 4 0] fun {$ X} X>2 end}}
```

affiche

```
[5 4]
```

(Nous utilisons la procédure `Show`, qui affiche la valeur instantanée de son argument. Contrairement à `Browse`, cet affichage n'est pas mis à jour si l'argument est lié plus tard.) `Filter` fonctionne comme on pouvait s'y attendre

quand toutes les valeurs de l'entrée sont disponibles. Explorons le comportement dataflow de `Filter`.

a) Que se passe-t-il quand nous exécutons ceci ? :

```
declare A
{Show {Filter [5 1 A 4 0] fun {$ X} X>2 end}}
```

Un des éléments de la liste est une variable `A` qui est non liée. Souvenez-vous que les instructions **`case`** et **`if`** suspendent le fil dans lequel elles s'exécutent, jusqu'à ce qu'elles puissent décider du chemin à prendre.

b) Que se passe-t-il quand nous exécutons ceci ? :

```
declare Out A
thread Out={Filter [5 1 A 4 0]
            fun {$ X} X>2 end} end
{Show Out}
```

Souvenez-vous que l'appel de `Show` affiche son argument tel il est à l'instant de l'appel. Plusieurs résultats possibles peuvent être affichés. Lesquels et pourquoi ? La fonction `Filter` est-elle déterministe ? Pourquoi oui ou pourquoi non ?

c) Que se passe-t-il quand nous exécutons ceci ? :

```
declare Out A
thread Out={Filter [5 1 A 4 0]
            fun {$ X} X>2 end} end
{Delay 1000} {Show Out}
```

Souvenez-vous que l'appel `{Delay N}` suspend son fil pour au moins `N` ms. Pendant ce temps, d'autres fils peuvent s'exécuter.

d) Que se passe-t-il quand nous exécutons ceci ? :

```
declare Out A
thread Out={Filter [5 1 A 4 0]
            fun {$ X} X>2 end} end
thread A=6 end
{Delay 1000} {Show Out}
```

Qu'est-ce qui est affiché et pourquoi ?

➤ **Exercice 7** — *La concurrence et les exceptions*

Considérez l'abstraction de contrôle suivante qui implémente **`try`**–**`finally`** :

```
proc {TryFinally S1 S2}
B Y in
   try {S1} B=false catch X then B=true Y=X end
   {S2}
   if B then raise Y end end
end
```

Avec la sémantique de la machine abstraite comme un guide, déterminez les différents résultats possibles du programme suivant :

```
local U=1 V=2 in
   {TryFinally
    proc {$}
       thread
          {TryFinally proc {$} U=V end
                      proc {$} {Browse bing} end}
    end end
    proc {$} {Browse bong} end}
end
```

Combien de résultats différents sont possibles ? Combien d'exécutions différentes sont possibles ?

Chapitre **5**

La programmation avec état explicite

L'état c'est moi.
– Louis XIV (1638-1715)

Si la programmation déclarative est comme un cristal, immuable et pratiquement éternel, alors la programmation avec état est organique : elle grandit et se développe.
– *Inspiré par* On Growth and Form*, D'Arcy Wentworth Thompson (1860-1948)*

Au premier coup d'œil, l'état explicite est une extension mineure de la programmation déclarative : le résultat d'un composant ne dépend pas seulement de ses arguments mais aussi d'un paramètre interne, appelé son « **état** ». Ce paramètre donne au composant une mémoire à long terme, un « sens de l'histoire » si vous voulez. Sans l'état, un composant n'a qu'une mémoire à court terme, qui existe pendant une invocation du composant. L'état ajoute une branche potentiellement infinie à un programme à exécution finie. Un composant qui s'exécute pendant un temps fini ne peut réunir qu'une quantité limitée d'information. Si le composant a un état, il faudra ajouter à cette information finie l'information enregistrée par l'état. Cette « histoire » peut être indéfiniment longue.

Oliver Sacks a décrit le cas d'une personne cérébrolésée n'ayant qu'une mémoire à court terme [80]. Il vit dans un « présent » interminable avec seulement une mémoire de quelques secondes dans le passé. Le mécanisme du cerveau qui « fixe » les souvenirs à court terme dans la mémoire à long terme est cassé. Ces malades utilisent peut-être le monde externe comme une sorte de mémoire à long terme ? Cette analogie montre l'importance de l'état pour les êtres humains. Nous verrons que l'état est aussi important pour la programmation.

Les degrés de la déclarativité

La programmation avec état est souvent appelée **programmation impérative** et la programmation sans état **programmation déclarative**. Ces derniers termes ne sont pas tout à fait corrects, mais la tradition a maintenu leur utilisation. La programmation déclarative, prise à la lettre, signifie programmation avec des déclarations : dire ce qu'il faut (le « quoi ») et laisser le système choisir l'algorithme (le « comment »). La programmation impérative, prise à la lettre, signifie programmation avec des commandes : dire comment faire quelque chose. Dans ce sens, le modèle déclaratif du chapitre 2 est impératif aussi, puisqu'il définit des séquences de commandes.

Le vrai problème est que « déclaratif » n'est pas une propriété absolue, mais une question de degré. Le langage Fortran, développé vers la fin des années 1950, est le premier langage populaire qui permet d'écrire les expressions arithmétiques dans une syntaxe qui ressemble à la notation mathématique [8]. Par rapport à un langage d'assemblage, c'est certainement déclaratif ! On pouvait dire à l'ordinateur de calculer `I+J` sans lui préciser où trouver `I` et `J` dans la mémoire ni quelles instructions machines utiliser pour les additionner. Dans ce sens relatif, les langages deviennent de plus en plus déclaratifs au cours des années. Après Fortran il y eut Algol-60 et la programmation structurée [19, 20, 70], qui a mené à Simula-67 et aux langages orientés objet modernes [71, 75].[1]

Ce livre reste fidèle à l'usage traditionnel du terme déclaratif pour la programmation sans état et du terme impératif pour la programmation avec état. Le modèle de calcul du chapitre 2 est appelé « déclaratif », même si les modèles postérieurs sont peut-être plus déclaratifs, parce qu'ils sont plus expressifs. L'usage traditionnel est conservé car il y a un sens important dans lequel le modèle déclaratif est vraiment déclaratif. Ce sens apparaît quand on regarde le modèle des points de vue de la programmation logique et de la programmation fonctionnelle :

- Un programme logique peut être « lu » de deux manières : soit comme un ensemble d'axiomes logiques (le « quoi ») soit comme un ensemble de commandes (le « comment »). Cette dualité est résumée par la célèbre équation de Robert Kowalski : Algorithme = Logique + Contrôle [55, 56]. Les axiomes logiques, quand ils sont supplémentés par des informations de contrôle (implicites ou données explicitement par le programmeur), définissent un programme qui peut être exécuté sur un ordinateur. Cette dualité existe aussi pour le modèle déclaratif.
- Un programme fonctionnel peut aussi être « lu » de deux manières : soit comme la définition d'un ensemble de fonctions dans le sens mathématique (le « quoi »)

1. Il est remarquable que ces trois langages aient tous été conçus dans une période de dix ans, de 1957 à 1967. Comme les langages Lisp et Absys datent aussi de cette période et que le langage Prolog est de 1972, nous pouvons parler d'un véritable âge d'or pour la conception des langages de programmation.

soit comme un ensemble de commandes pour évaluer ces fonctions (le « comment »). Les commandes seront exécutées dans un ordre particulier qui est défini par le langage. Les deux ordres les plus populaires sont l'évaluation immédiate (« *eager evaluation* ») et l'évaluation paresseuse (« *lazy evaluation* »). Quand l'ordre est connu, la définition mathématique peut être exécutée sur un ordinateur.

En pratique, la lecture déclarative d'un programme logique ou fonctionnel peut perdre de son aspect « quoi » parce que le programmeur doit expliciter le « comment » (voir l'épigraphe d'O'Keefe à la tête du chapitre 3). Par exemple, une définition déclarative d'une recherche dans un arbre doit donner presque autant de commandes qu'une définition impérative. Cependant, la programmation déclarative a toujours trois avantages capitaux. Premièrement, il est plus facile de construire des abstractions dans un contexte déclaratif, parce que les opérations déclaratives sont de nature compositionnelles. Deuxièmement, les programmes déclaratifs sont plus faciles à tester, parce que l'on peut tester chaque partie (donner les arguments et vérifier les résultats) indépendamment des autres. Tester les programmes avec état est plus difficile parce qu'il faut tester des séquences d'appels (à cause de l'histoire interne). Troisièmement, le raisonnement est plus facile avec la programmation déclarative qu'avec la programmation impérative (par exemple, le raisonnement algébrique est possible).

La structure du chapitre

Ce chapitre présente les idées et techniques de base pour l'utilisation de l'état dans la conception de programmes.

- Dans les trois premières sections, nous introduisons et définissons le concept d'état explicite.
  - La section 5.1 introduit l'état explicite : elle définit la notion générale d'état, qui est indépendant de tout modèle de calcul, et montre les différentes manières dont les modèles de calcul implémentent cette notion.
  - La section 5.2 explique les principes de base de la conception de systèmes et pourquoi l'état en est une partie essentielle. Elle donne aussi les premières définitions de la programmation par composants et de la programmation orientée objet.
  - La section 5.3 donne une définition précise du modèle de calcul avec état.
- Dans la section 5.4 nous expliquons les différentes manières de construire les abstractions de données, avec et sans état explicite. Nous expliquons également les deux principaux styles de construction des abstractions de données, le style ADT et le style objet.
- La section 5.5 concerne le polymorphisme, un des concepts les plus importants pour les abstractions de données. Le polymorphisme permet à un programme

de concentrer chaque responsabilité dans une abstraction de données, au lieu de l'étaler partout dans le programme.
– Enfin, la section 5.6 explique la programmation à grande échelle, c'est-à-dire la programmation par une équipe. C'est une extension de la présentation de la programmation à petite échelle donnée dans la section 3.8.

Le chapitre 6 poursuit la présentation de l'état avec un style de programmation particulièrement utile, à savoir la programmation orientée objet. Étant donné sa grande applicabilité, nous lui consacrons un chapitre entier.

5.1 L'ÉTAT C'EST QUOI ?

Nous avons déjà programmé avec l'état dans le modèle déclaratif du chapitre 3. Par exemple, les accumulateurs de la section 3.4.3 sont de l'état. Alors, pourquoi avons-nous besoin de tout un chapitre pour l'état ? Pour comprendre pourquoi, il faut regarder de près le concept d'état. Dans sa forme la plus simple, nous pouvons définir l'état ainsi :

> Un **état** est une séquence de valeurs dans le temps qui contient les résultats intermédiaires d'un calcul souhaité.

Nous examinerons les différentes manières dont l'état peut être présent dans un programme.

5.1.1 L'état implicite (déclaratif)

Il est possible que la séquence n'existe que dans l'esprit du programmeur. Le modèle de calcul reste déclaratif. Ce genre d'état s'appelle l'**état implicite** ou l'**état déclaratif**. Par exemple, prenez la fonction déclarative `SumList` :

```
fun {SumList Xs S}
   case Xs of nil then S
   [] X|Xr then {SumList Xr X+S} end
end
```

À chaque appel récursif, il y a deux arguments : le reste inexaminé `Xs` de la liste d'entrée, et la somme `S` de la partie examinée de la liste d'entrée. Pendant le calcul de la somme d'une liste, `SumList` s'appelle plusieurs fois. Prenons la paire `(Xs#S)` à chaque appel parce qu'elle nous donne toutes les informations dont nous avons besoin pour caractériser l'appel. Pour l'appel `{SumList [1 2 3 4] 0}`, nous obtenons la séquence suivante des paires :

```
[1 2 3 4] # 0
[2 3 4] # 1
[3 4] # 3
[4] # 6
nil # 10
```

Cette séquence est un état. Vu de cette façon, `SumList` calcule avec un état. Mais ni le programme ni le modèle de calcul « savent » cela. L'état est complètement dans l'esprit du programmeur.

5.1.2 L'état explicite

Il peut être utile pour une fonction d'avoir un état qui survit au-delà des appels de la fonction et qui est caché des appelants. Par exemple, nous pouvons étendre `SumList` pour compter le nombre de fois qu'elle est appelée. Pour les appeleurs de `SumList` il n'y a pas de raison de connaître cette extension. Nous dirions même plus : pour des raisons de modularité, les appeleurs ne doivent pas connaître l'extension. Comme nous l'avons vu dans la section 4.4, cela ne peut pas être programmé dans le modèle déclaratif. Le mieux que l'on puisse faire est d'ajouter deux arguments à `SumList` (un compte d'entrée et de sortie) et de les enfiler dans tous les appelants. Pour le faire sans arguments supplémentaires, il faut un état explicite :

> Un **état explicite** dans une procédure est un état dont l'existence s'étend au-delà d'un appel de la procédure sans être présent dans les arguments de la procédure.

L'état explicite ne peut pas être exprimé dans le modèle déclaratif. Pour l'obtenir, nous étendons le modèle avec un nouveau concept, une sorte de conteneur que nous appelons une cellule. Une cellule a un nom, une durée de vie illimitée et un contenu qui peut être changé. Si la procédure connaît le nom, elle pourra changer le contenu. Le modèle déclaratif étendu avec des cellules s'appelle le **modèle avec état**.

Contrairement à l'état déclaratif, l'état explicite n'est pas seulement dans l'esprit du programmeur. Il est visible dans le programme et le modèle de calcul. Nous pouvons utiliser une cellule pour ajouter une mémoire à long terme à `SumList`. Par exemple, nous pouvons compter le nombre de fois qu'elle est appelée :

```
local C={NewCell 0} in
   fun {SumList Xs S}
      C:=@C+1
      case Xs of nil then S
      [] X|Xr then {SumList Xr X+S} end
   end
   fun {SumCount} @C end
end
```

C'est la même situation qu'avant, mais nous définissons une cellule et nous mettons à jour son contenu dans `SumList`. Nous ajoutons aussi la fonction `SumCount` pour rendre l'état observable. Voici un résumé des nouvelles opérations sur l'état explicite. `NewCell` crée une nouvelle cellule avec un contenu initial donné (0 dans l'exemple).

@ donne accès au contenu actuel et := met un nouveau contenu. Si `SumCount` n'est pas utilisée, cette version de `SumList` ne pourra pas être distinguée de la version précédente : on l'appelle de la même façon et elle donne les mêmes résultats.[2]

La possibilité d'utiliser l'état explicite est très importante. Elle supprime les limites de la programmation déclarative (voir section 4.4). Avec l'état explicite, les abstractions de données augmentent énormément en modularité parce qu'il est possible d'encapsuler un état à l'intérieur. L'accès à l'état est limité selon les opérations de l'abstraction de données. Cette idée est au cœur de la programmation orientée objet, un style de programmation puissant qui est élaboré dans le chapitre 6.

5.2 L'ÉTAT ET LA CONSTRUCTION DE SYSTÈMES

Le principe d'abstraction

À notre connaissance, le principe de construction de systèmes le plus puissant pour des êtres intelligents avec des capacités limitées de réflection, comme les êtres humains, est le principe d'abstraction. Prenez un système quelconque. On peut considérer qu'il a deux parties : une spécification et une implémentation. La spécification est un contrat, dans le sens mathématique qui est plus fort que le sens légal. Le contrat définit comment le système doit se comporter. On dit qu'un système est correct si son comportement effectif est en accord avec le contrat. S'il se comporte autrement, on dira qu'il échoue.

La spécification définit comment le reste du monde interagit avec le système, vu de l'extérieur. L'implémentation est la construction du système, vu de l'intérieur. La propriété miraculeuse de la distinction spécification/implémentation est que la spécification est généralement bien plus simple à comprendre que l'implémentation. On ne doit pas savoir comment construire une montre pour lire l'heure. Pour paraphraser l'évolutionniste Richard Dawkins, que l'horloger soit aveugle ou non, l'important est que la montre fonctionne.

La conséquence est qu'il est possible de construire un système comme une série de couches. On peut procéder pas à pas, en construisant couche sur couche. À chaque couche, il suffit pour l'implémentation d'utiliser la couche juste en dessous pour fournir la couche juste au-dessus. Il n'est pas nécessaire de comprendre tout à la fois.

Les systèmes qui grandissent

Comment cette approche est-elle soutenue par la programmation déclarative ? Avec le modèle déclaratif du chapitre 2, tout ce que le système « sait » est à l'extérieur, sauf

2. Les seules différences sont un petit ralentissement et une petite augmentation d'utilisation de mémoire. Dans presque tous les cas pratiques, ces différences peuvent être ignorées.

pour les connaissances fixées à sa création. Pour être précis, comme une procédure n'a pas d'état, toutes ses connaissances sont dans ses arguments. La programmation déclarative est comme un organisme qui maintient toutes ses connaissances à l'extérieur, dans l'environnement. Malgré sa revendication du contraire (voir l'épigraphe à la tête du chapitre), c'était exactement la situation de Louis XIV : l'état n'était pas dans sa personne mais autour de lui, dans la France du XVII^{e} siècle. On en conclut que le principe d'abstraction n'est pas bien soutenu par la programmation déclarative, car on ne peut pas mettre de nouvelles connaissances dans un composant.

Le chapitre 4 a partiellement résolu ce problème en ajoutant la concurrence. Les objets à flots peuvent accumuler des connaissances dans leurs arguments internes. Mais l'usage de ces connaissances est limité parce qu'il n'y a pas de modularité. Dans ce chapitre, nous avons de l'état sans la concurrence, ce qui est avantageux pour la modularité.

5.2.1 Les propriétés du système

Quelles propriétés un système devrait-il avoir pour soutenir au mieux le principe d'abstraction ? En voici trois :

- *Encapsulation*. Il devrait être possible de cacher l'intérieur d'une partie du système du reste.
- *Compositionnalité*. Il devrait être possible de combiner des parties de système pour faire une nouvelle partie.
- *Instanciation/invocation*. Il devrait être possible de créer de nombreuses instances d'une partie à partir d'une définition. Ces instances se « connectent » dans leur environnement (le reste du système dans lequel ils vivront) quand elles sont créées.

Ces propriétés nécessitent un soutien du langage de programmation, par exemple la portée lexicale soutient l'encapsulation et la programmation d'ordre supérieur soutient l'instanciation. Ces propriétés ne nécessitent pas d'état ; elles peuvent aussi être utilisées dans la programmation déclarative. Par exemple, l'encapsulation est orthogonale à l'état. D'un côté, il est possible d'utiliser l'encapsulation dans les programmes déclaratifs sans état ; nous l'avons déjà fait de nombreuses fois, par exemple dans les objets à flots. De l'autre, il est possible d'utiliser l'état sans encapsulation, si on définit l'état globalement pour que tous les composants puissent y accéder.

Les invariants

L'encapsulation et l'état explicite sont plus utiles quand ils sont utilisés ensemble. Ajouter l'état à la programmation déclarative complique le raisonnement sur un programme, parce que son comportement dépend de l'état. Par exemple, une procédure

peut faire un effet de bord, c'est-à-dire qu'elle modifie un état qui est visible par le reste du programme. Les effets de bord compliquent énormément le raisonnement sur un programme. Introduire l'encapsulation est une manière de retrouver un raisonnement simple parce qu'avec l'encapsulation, les systèmes à état peuvent être conçus de façon à ce qu'une propriété bien définie, qui s'appelle **invariant**, soit toujours vraie quand on voit le système de l'extérieur. Le raisonnement sur le système devient de nouveau indépendant du raisonnement sur son environnement. Nous retrouvons en partie une des propriétés attirantes de la programmation déclarative.

Même avec les invariants, la programmation avec état n'est pas aussi simple que la programmation déclarative. Nous trouvons qu'une bonne règle empirique pour les systèmes complexes est de maintenir un maximum de composants déclaratifs. L'état ne doit pas être « étalé » sur beaucoup de composants. Il doit être concentré dans quelques composants bien choisis.

5.2.2 La programmation par composants

Les trois propriétés, l'encapsulation, la compositionnalité et l'instanciation, définissent la programmation par composants (voir section 5.6). Un composant spécifie un fragment de programme avec un intérieur et un extérieur et une interface bien définie entre les deux. L'intérieur est caché de l'extérieur, sauf pour ce qui est permis par l'interface. Les composants peuvent être composés pour faire de nouveaux composants. Un composant peut être instancié, ce qui crée une nouvelle instance dont les liens sont édités dans l'environnement du système. Les composants sont un concept omniprésent. Nous les avons déjà vus sous plusieurs apparences :

- *Les procédures* (l'abstraction procédurale). Nous avons vu un premier exemple de composants dans le modèle de calcul déclaratif. Le composant est appelé une définition de procédure et son instance, une invocation de procédure. L'abstraction procédurale est à la base des modèles plus avancés de composants qui viennent après.
- *Les foncteurs* (les unités de compilation). Un genre de composant particulièrement utile est une unité de compilation, qui peut être compilée indépendamment des autres composants. Ces composants sont appelés foncteurs et leurs instances, modules.
- *Les objets à flots*. Un système avec des entités indépendantes qui interagissent par des flots peut être vu comme un graphe de composants concurrents qui s'envoient des messages.

Dans la programmation par composants, le moyen naturel d'étendre un composant est la composition : construire un nouveau composant qui contient l'ancien. Le nouveau composant offre une nouvelle fonctionnalité et utilise l'ancien composant pour l'implémenter.

5.2.3 La programmation orientée objet

La programmation orientée objet est un ensemble de techniques pour maîtriser la programmation avec état. Tout le chapitre 6 est consacré à ces techniques. La programmation orientée objet est basée sur une façon particulière de faire l'abstraction de données qui s'appelle un **objet**. Les objets et les types abstraits (ADT) sont fondamentalement différents. Les ADT séparent les valeurs et leurs opérations. Les objets les combine en une seule entité agrégée que l'on peut invoquer. Les différences entre ADT et objets sont expliquées plus en détails dans la section 5.4. Les objets sont importants parce qu'ils facilitent l'utilisation des techniques puissantes de polymorphisme et héritage. Le polymorphisme peut aussi être utilisé dans la programmation par composants. Cela est expliqué dans la section 5.5. L'héritage est un nouveau concept qui ne fait pas partie de la programmation par composants :

– *L'héritage*. Il est possible de construire une abstraction de données de façon incrémentale, comme une extension ou une modification d'une autre abstraction de données.

Les définitions incrémentales de composants s'appellent des **classes** et leurs instances sont des **objets**.

5.3 LE MODÈLE DÉCLARATIF AVEC ÉTAT EXPLICITE

Une manière d'introduire l'état est d'avoir des composants concurrents qui communiquent avec d'autres composants, comme les objets à flots du chapitre 4. Dans ce chapitre, nous faisons autrement. Nous ajoutons le concept d'état explicite au modèle déclaratif. À la différence du chapitre 4, le modèle résultant est toujours séquentiel. Nous l'appelons le **modèle avec état**.

Chaque instance de l'état explicite est une paire de deux entités du langage. La première entité est l'identité de l'instance et la deuxième entité est le contenu actuel de l'instance. Il y a une opération qui prend l'identité et qui renvoie le contenu actuel. Cette opération définit une correspondance entre les identités de toutes les instances de l'état et toutes les entités du langage. Curieusement, quand on modifie un état aucune des deux entités du langage n'est modifiée. Il n'y a que la correspondance qui change.

5.3.1 Les cellules

Nous ajoutons l'état explicite comme un nouveau type de base au modèle de calcul. Nous l'appelons cellule. Une cellule est une paire composée d'une constante, qui est un nom, et d'une référence dans la mémoire à affectation unique. Comme les noms sont infalsifiables, les cellules sont un exemple d'ADT sécurisé. L'ensemble de toutes les cellules s'appelle la **mémoire à affectation multiple**. La figure 5.1 montre le

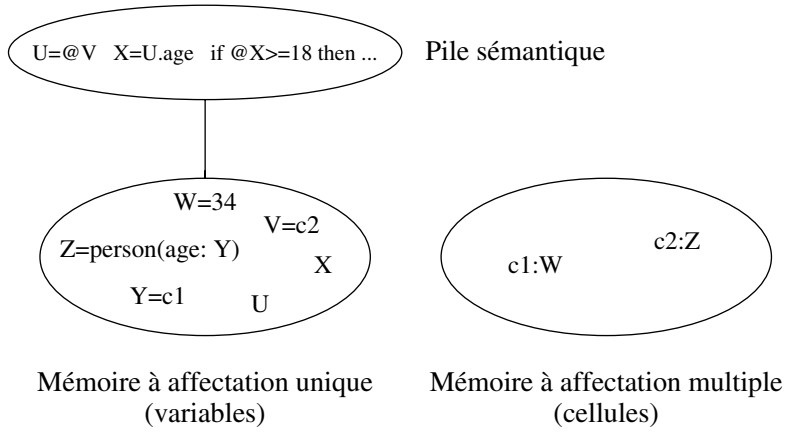


Figure 5.1 Le modèle déclaratif avec état explicite.

nouveau modèle de calcul. Il y a deux mémoires : la mémoire à affectation unique, qui contient des variables dataflow que l'on peut lier à une valeur, et la mémoire à affectation multiple, qui contient les cellules (les paires de noms et références). Le tableau 5.1 montre son langage noyau. Comparé au modèle déclaratif, il y a deux nouvelles instructions, les opérations `NewCell` et `Exchange`. Ces opérations sont définies informellement dans le tableau 5.2.

⟨s⟩ ::=	
skip	Instruction vide
⟨s⟩$_1$ ⟨s⟩$_2$	Séquence d'instructions
local ⟨x⟩ **in** ⟨s⟩ **end**	Création de variable
⟨x⟩$_1$=⟨x⟩$_2$	Lien variable-variable
⟨x⟩=⟨v⟩	Création de valeur
if ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Instruction conditionnelle
case ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Correspondance de formes
{⟨x⟩ ⟨y⟩$_1$ ⋯ ⟨y⟩$_n$}	Application de procédure
try ⟨s⟩$_1$ **catch** ⟨x⟩ **then** ⟨s⟩$_2$ **end**	Contexte d'exception
raise ⟨x⟩ **end**	Lève exception
{NewName ⟨x⟩}	Création de nom
{NewCell ⟨x⟩ ⟨y⟩}	**Création de cellule**
{Exchange ⟨x⟩ ⟨y⟩ ⟨z⟩}	**Échange de cellule**

Tableau 5.1 Le langage noyau avec état explicite.

Opération	Description
`{NewCell X C}`	Créez une nouvelle cellule `C` avec contenu initial `X`.
`{Exchange C X Y}`	Opération atomique : Liez `X` à l'ancien contenu de la cellule `C` et mettez `Y` comme nouveau contenu.
`X=@C`	Liez `X` au contenu actuel de la cellule `C`.
`C:=X`	Mettez `X` comme nouveau contenu de la cellule `C`.
`X=C:=Y`	Une autre syntaxe pour `{Exchange C X Y}`.

Tableau 5.2 Les opérations sur les cellules.

Pour le confort du programmeur, ce tableau contient deux opérations supplémentaires, `@` (l'accès) et := (l'affectation). Ces deux opérations n'offrent pas de nouvelle fonctionnalité parce qu'elles peuvent être définies avec `Exchange`. L'utilisation de `C:=Y` comme une expression a l'effet d'un `Exchange` : elle renvoie le contenu ancien comme résultat.

Il est extraordinaire que l'addition des cellules suffise pour construire tous les merveilleux concepts que l'état peut offrir. Tous les concepts sophistiqués comme les objets, les classes et les autres abstractions de données peuvent être construits avec le modèle déclaratif étendu par les cellules. La section 6.2 explique comment construire les classes et les objets. En pratique, leur sémantique est définie par cette même construction, mais le langage a un soutien syntaxique pour faciliter leur usage et l'implémentation a un soutien pour les rendre plus efficaces [38].

5.3.2 La sémantique des cellules

La sémantique des cellules demande une extension à la mémoire qui s'appelle la **mémoire affectable** ou la **mémoire à affectation multiple**. Les opérations `NewCell` et `Exchange` sont définies par rapport à cette mémoire. Nous définissons aussi comment la gestion de mémoire fonctionne pour cette mémoire.

L'extension de l'état d'exécution

À côté de la mémoire à affectation unique σ, nous ajoutons une nouvelle mémoire μ appelée **mémoire affectable**. Cette mémoire contient des cellules, qui sont des paires de la forme $x : y$, où x et y sont des variables dans la mémoire à affectation unique. Au début d'une exécution, la mémoire affectable est vide. La sémantique garantit que x est toujours liée à un nom qui représente une cellule. Par contre, y peut être n'importe quelle valeur partielle. L'état d'exécution devient un triple (MST, σ, μ).

L'opération `NewCell`

L'instruction sémantique est :

$(\{\texttt{NewCell}\ \langle \mathsf{x} \rangle\ \langle \mathsf{y} \rangle\}, E)$

L'exécution fait les actions suivantes :

- Créer un nouveau nom de cellule n.
- Lier $E(\langle \mathsf{y} \rangle)$ et n en mémoire.
- Si le lien réussit, ajouter la paire $E(\langle \mathsf{y} \rangle) : E(\langle \mathsf{x} \rangle)$ à la mémoire affectable μ.
- S'il échoue, lever une condition d'erreur.

L'opération `Exchange`

L'instruction sémantique est :

$(\{\texttt{Exchange}\ \langle \mathsf{x} \rangle\ \langle \mathsf{y} \rangle\ \langle \mathsf{z} \rangle\}, E)$

L'exécution fait les actions suivantes :

- Si la condition d'activation est vraie ($E(\langle \mathsf{x} \rangle)$ est déterminée), faites les actions suivantes :
  - Si $E(\langle \mathsf{x} \rangle)$ n'est pas liée au nom d'une cellule, lever une condition d'erreur.
  - Si la mémoire affectable contient $E(\langle \mathsf{x} \rangle) : w$, faire les actions suivantes :
    - ⋆ Modifier la paire en mémoire affectable pour devenir $E(\langle \mathsf{x} \rangle) : E(\langle \mathsf{z} \rangle)$.
    - ⋆ Lier $E(\langle \mathsf{y} \rangle)$ et w en mémoire.
- Si la condition d'activation est fausse, suspendre l'exécution.

La gestion de mémoire

Il faut deux modifications à la gestion de mémoire pour tenir compte de la mémoire affectable :

- *L'extension de la définition d'accessibilité*. Une variable y sera accessible si la mémoire affectable contient $x : y$ et si x est accessible.
- *La récupération de cellules*. Si une variable x devient inaccessible et si la mémoire affectable contient la paire $x : y$, enlevez cette paire.

5.3.3 Le partage et l'égalité

Avec l'introduction des cellules, nous avons de ce fait étendu le concept d'égalité. Nous devons distinguer l'égalité des cellules et l'égalité de leurs contenus. Cela nous mène aux concepts de partage (« *aliasing* ») et d'égalité d'identité.

Le partage (« aliasing »)

Le partage, appelé aussi *aliasing*, se produit quand deux identificateurs `X` et `Y` référencent la même cellule. On dit que les deux identificateurs sont des alias. Changer le contenu de `X` changera aussi le contenu de `Y`. Prenez par exemple une cellule :

```
X={NewCell 0}
```

Nous pouvons créer une deuxième référence `Y` à cette cellule :

```
declare Y in
Y=X
```

Changer le contenu de `Y` changera le contenu de `X` :

```
Y:=10
{Browse @X}
```

qui affiche 10. En général, quand le contenu d'une cellule est changé, tous les alias de la cellule voient le nouveau contenu. Quand on raisonne sur un programme, on doit faire attention aux alias. Cela peut être difficile car ils peuvent être répandus sur tout le programme. Le problème pourra devenir faisable si l'état est encapsulé dans une petite partie du programme sans pouvoir s'échapper. C'est une des raisons clés pour laquelle l'abstraction de données est une idée particulièrement bonne quand on l'utilise avec l'état explicite.

L'égalité d'identité et l'égalité de structure

Deux valeurs seront égales si elles ont la même structure. Par exemple :

```
X=person(age:25 name:"George")
Y=person(age:25 name:"George")
{Browse X==Y}
```

Le Browser affiche **`true`**. Nous appelons ce concept l'**égalité de structure**. C'est l'égalité que nous avons utilisée jusqu'à ce point. Mais avec les cellules, nous introduisons une nouvelle notion d'égalité, l'**égalité d'identité**. Deux cellules ne sont pas égales si elles ont le même contenu, mais elles sont égales si elles sont la même cellule ! Voici deux cellules :

```
X={NewCell 10}
Y={NewCell 10}
```

Ce sont des cellules différentes avec des identités différentes. La comparaison suivante :

```
{Browse X==Y}
```

affiche **`false`**. Il est logique que les cellules ne soient pas égales, puisque changer le contenu d'une cellule ne changera pas le contenu de l'autre. Mais nos deux cellules ont le même contenu :

```
{Browse @X==@Y}
```

Le Browser affiche **`true`**. C'est une pure coïncidence ; cette comparaison ne restera pas forcément vraie pendant toute l'exécution. Nous concluons en faisant le constat que les alias ont les mêmes identités. L'exemple suivant :

```
X={NewCell 10}
Y=X
{Browse X==Y}
```

affiche **`true`** car `X` et `Y` sont des alias et ils font donc référence à la même cellule.

5.4 L'ABSTRACTION DE DONNÉES

Une abstraction de données est une manière abstraite d'utiliser des données, c'est-à-dire que nous pouvons utiliser les données sans dépendre d'une implémentation particulière. Une abstraction de données se compose d'un ensemble d'instances qui peuvent être utilisées selon un jeu de règles, que nous appelons son **interface**. Parfois, nous utiliserons librement le terme « type » quand nous parlerons d'une abstraction de données. L'utilisation d'une abstraction a beaucoup d'avantages sur l'utilisation directe de son implémentation : l'abstraction est en général bien plus simple (car l'implémentation n'est pas limitée par la taille de son interface), le raisonnement sur les données peut être bien plus simple (car elles ont leurs propres propriétés) et le système peut garantir que les données sont utilisées correctement (car nous ne pouvons pas toucher l'implémentation sauf en passant par l'interface).

La section 3.5 a montré une forme d'abstraction de données, à savoir le type de données abstrait ou ADT : un ensemble de valeurs avec un ensemble d'opérations sur ces valeurs. Nous avons défini une pile en ADT avec des valeurs de pile et les opérations d'empiler (« *push* ») et de dépiler (« *pop* »). Mais les ADT ne sont pas la seule méthode pour le traitement abstrait des données. Maintenant que nous avons ajouté l'état explicite au modèle, nous pouvons présenter un jeu plus complet de techniques pour faire l'abstraction de données.

5.4.1 Huit manières pour organiser une abstraction de données

Une abstraction de données avec une même fonctionnalité peut être organisée de différentes façons. Par exemple, dans la section 3.5 nous avons vu qu'une abstraction simple comme une pile peut être ouverte ou sécurisée. Maintenant, nous introduisons

deux axes supplémentaires, l'agrégation et l'état explicite. L'agrégation est un concept fondamental qui a été observé en 1975 par John Reynolds [16, 34, 78]. Chaque axe a deux choix, ce qui donne huit manières en tout d'organiser une abstraction de données. Certaines sont rarement utilisées. D'autres sont plus populaires. Mais chacune présente des avantages et des inconvénients. Nous expliquerons brièvement chaque axe avec quelques exemples dans lesquels nous choisirons à chaque fois la plus appropriée des huit manières.

L'ouverture et la sécurité

Une abstraction de données est sécurisée si le langage fait respecter son encapsulation. Sinon elle est ouverte. Pour une abstraction ouverte, c'est la discipline du programmeur qui doit faire respecter l'encapsulation. À proprement parler, si l'encapsulation n'est pas imposée par le langage, l'abstraction de données ne sera plus une abstraction. Nous l'appellerons toujours une abstraction parce qu'elle pourra toujours être utilisée comme telle. La seule différence sera le responsable de l'encapsulation : le langage ou le programmeur.

L'agrégation

Une abstraction de données est non agrégée si elle définit deux sortes d'entités, appelées **valeurs** et **opérations**. Les valeurs sont passées comme arguments aux opérations et renvoyées comme résultats. Une abstraction de données non agrégée s'appelle généralement un **type abstrait de données** ou **ADT**. Elle peut être sécurisée avec une « clé ». La clé est une autorisation pour accéder aux données internes d'une valeur abstraite (et la modifier si la valeur a un état). Toutes les opérations de l'ADT connaissent la clé. Le reste du programme ne la connaît pas. La clé peut être un nom, qui est une constante infalsifiable (voir la documentation de Mozart [23]).

Le langage le mieux connu basé sur les ADT est probablement CLU, conçu et implémenté dans les années 1970 par Barbara Liskov et ses étudiants [61]. CLU a le mérite d'être le premier langage implémenté avec un soutien linguistique pour les ADT.

Une abstraction de données est agrégée si elle définit une seule sorte d'entité, appelée **agrégat** ou **objet**, qui combine les notions de valeur et d'opération. Une abstraction de données agrégée est parfois appelée abstraction de données procédurale ou PDA (« *Procedural Data Abstraction* »). Pour faire une opération, on appelle l'objet en l'informant de l'opération qu'il doit faire. C'est souvent appelé « envoyer un message à l'objet », mais l'appel est synchrone comme l'appel d'une procédure. Il retourne quand l'opération est complètement terminée.

Le style objet est devenu très populaire car il présente d'importants avantages par rapport à la modularité et à la structure de programme, grâce à la facilité avec laquelle il soutient le polymorphisme et l'héritage. Le polymorphisme est expliqué plus loin.

L'héritage est expliqué dans le chapitre 6, qui est complètement consacré au style objet.

L'état explicite

Une abstraction de données est dite avec état si elle utilise l'état explicite. Sinon elle est sans état ou déclarative. Le chapitre 3 donne deux exemples, une pile (voir section 3.5) et un dictionnaire sans état (voir section 3.8.4).

La décision de faire une abstraction avec ou sans état dépend des soucis de modularité, de concision et de facilité de raisonnement. Nous remarquons que les « valeurs » dans une abstraction de données, comme elle est définie ici, peuvent avoir de l'état. C'est un léger abus de terminologie car les valeurs de base que nous utilisons sont toutes sans état (des constantes).

5.4.2 Cinq manières pour emballer une pile

Prenons le type ⟨Stack T⟩ de la section 3.5 et voyons comment l'adapter à certaines des huit possibilités. Nous donnons cinq possibilités utiles. Nous commençons par la plus simple, la version ouverte déclarative. Nous utilisons ensuite cette version pour construire quatre versions sécurisées. La figure 5.2 montre les cinq possibilités.

Ouverte, déclarative et non agrégée	*Le style ouvert déclaratif comme dans Prolog et Scheme*
Sécurisée, déclarative et non agrégée	*Le style ADT comme dans beaucoup de langages (par ex. entiers en Java)*
Sécurisée, déclarative et agrégée	*Le style PDA avec objets déclaratifs*
Sécurisée, avec état et agrégée	*Le style PDA avec objets à état comme dans Smalltalk et Java*
Sécurisée, avec état et non agrégée	*Le style ADT avec valeurs à état (par ex. dictionnaires en Oz)*

Figure 5.2 Cinq manières pour emballer une pile.

La figure 5.3 fait une illustration graphique des quatre versions sécurisées et de leurs différences. Dans cette figure, les rectangles avec des bords épais marqués « `Pop` » sont des procédures à appeler. Les flèches entrantes sont les entrées et les flèches sortantes sont des sorties. Les rectangles avec des bords minces et trous de serrure sont les structures de données emballées qui font les entrées et sorties des procédures `Pop`. Les structures de données emballées peuvent seulement être déballées à l'intérieur des

procédures `Pop`. Deux des procédures `Pop` (la deuxième et la troisième) emballent les données elles-mêmes avec la portée lexicale.

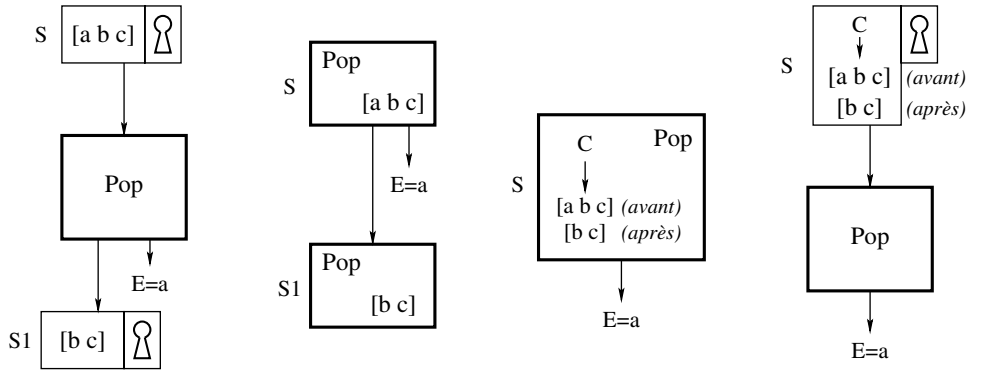


Figure 5.3 Quatre versions sécurisées d'une pile.

La pile ouverte déclarative

Définissons d'abord la fonctionnalité de la pile avec une définition simple :

```
declare
local
   fun {NewStack} nil end
   fun {Push S E} E|S end
   fun {Pop S ?E} case S of X|S1 then E=X S1 end end
   fun {IsEmpty S} S==nil end
in
   Stack=stack(new:NewStack push:Push
               pop:Pop isEmpty:IsEmpty)
end
```

`Stack` est un module qui regroupe les opérations de pile.[3] Cette version est ouverte, déclarative et non agrégée.

3. Remarquez que `Stack` est une variable globale qui a besoin de **`declare`**, même si elle est à l'intérieur de l'instruction **`local`**.

La pile sécurisée déclarative non agrégée

Sécurisons cette version avec une paire emballeur/déballeur, comme nous l'avons vu dans la section 3.5 :

```
declare
local Wrap Unwrap
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap nil} end
   fun {Push S E} {Wrap E|{Unwrap S}} end
   fun {Pop S ?E}
      case {Unwrap S} of X|S1 then E=X {Wrap S1} end end
   fun {IsEmpty S} {Unwrap S}==nil end
in
   Stack=stack(new:NewStack push:Push
               pop:Pop isEmpty:IsEmpty)
end
```

Cette version est sécurisée, déclarative et non agrégée. La pile est déballée quand on commence une opération ADT et de nouveau emballée quand l'opération se termine. À l'extérieur de l'ADT, la pile est toujours emballée.

La pile sécurisée déclarative agrégée

Faisons une version agrégée de la pile déclarative. L'idée est de cacher la pile à l'intérieur des opérations, pour qu'elle ne puisse pas en être séparée. Voici comment la programmer :

```
local
   fun {StackObject S}
      fun {Push E} {StackObject E|S} end
      fun {Pop ?E} case S of X|S1 then
         E=X {StackObject S1} end end
      fun {IsEmpty} S==nil end
   in stack(push:Push pop:Pop isEmpty:IsEmpty) end
in
   fun {NewStack} {StackObject nil} end
end
```

Cette version est sécurisée, déclarative et agrégée. Elle démontre le fait remarquable que pour sécuriser une abstraction de données il ne faut ni l'état explicite ni des noms. On peut le faire avec seulement la programmation d'ordre supérieur. La fonction `StackObject` prend une liste `S` et renvoie l'enregistrement des valeurs procédurales `stack(push:Push pop:Pop isEmpty:IsEmpty)`, dans lequel `S` est cachée

par la portée lexicale. Cet enregistrement représente l'objet et ses champs sont les méthodes. C'est très différent de la version déclarative non agrégée, où l'enregistrement représente un module et ses champs sont les opérations ADT. Voici un exemple d'utilisation :

```
declare S1 S2 S3 E in
S1={NewStack}
{Browse {S1.isEmpty}}
S2={S1.push 23}
S3={S2.pop E}
{Browse E}
```

Cette version étant à la fois agrégée et sécurisée, nous pouvons la considérer comme une forme déclarative de la programmation orientée objet. La pile `S1` est un **objet déclaratif**.

La pile sécurisée agrégée avec état

Utilisons maintenant l'état pour construire la pile. L'appel `NewStack` crée un nouvel objet pile :

```
fun {NewStack}
   C={NewCell nil}
   proc {Push E} C:=E|@C end
   fun {Pop} case @C of X|S1 then C:=S1 X end end
   fun {IsEmpty} @C==nil end
in
   stack(push:Push pop:Pop isEmpty:IsEmpty)
end
```

Cette version est sécurisée, agrégée et avec état. Cela s'appelle généralement un objet. Comme la version déclarative agrégée, l'objet est représenté par un enregistrement de valeurs procédurales. Cette version offre la fonctionnalité de la programmation orientée objet, à savoir un ensemble d'opérations (« méthodes ») avec un état caché à l'intérieur. Le résultat de l'appel de `NewStack` est une instance d'objet avec les trois méthodes `push`, `pop` et `isEmpty`.

La pile sécurisée agrégée avec état (envoi procédural)

Voici encore une autre manière d'implémenter une pile sécurisée agrégée avec état. Elle utilise une instruction **`case`** à l'intérieur d'une procédure au lieu d'un enregistrement :

```
fun {NewStack}
   C={NewCell nil}
   proc {Push E} C:=E|@C end
   fun {Pop} case @C of X|S1 then C:=S1 X end end
   fun {IsEmpty} @C==nil end
in
   proc {$ Msg}
      case Msg of push(X) then {Push X}
      [] pop(?E) then E={Pop}
      [] isEmpty(?B) then B={IsEmpty} end
   end
end
```

Cela s'appelle l'**envoi procédural** contrairement à la version précédente qui utilise l'**envoi par enregistrement**. Avec l'envoi procédural, un objet `S` est appelée comme `{S push(X)}`. Avec l'envoi par enregistrement, le même appel est écrit `{S.push X}`. L'envoi procédural sera utilisé partout dans le chapitre 6.

La pile sécurisée non agrégée avec état

Il est possible de combiner l'emballage avec les cellules pour obtenir une version qui est sécurisée, avec état et non agrégée. Ce style est peu utilisé dans la programmation orientée objet, mais il mérite d'être plus largement connu. Il n'utilise pas la programmation d'ordre supérieur directement. Chaque opération a un argument pile au lieu de deux pour la version déclarative :

```
declare
local Wrap Unwrap
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap {NewCell nil}} end
   proc {Push S E} C={Unwrap S} in C:=E|@C end
   fun {Pop S} C={Unwrap S} in
      case @C of X|S1 then C:=S1 X end end
   fun {IsEmpty S} @{Unwrap S}==nil end
in
   Stack=stack(new:NewStack push:Push
               pop:Pop isEmpty:IsEmpty)
end
```

Dans cette version, `NewStack` n'a besoin que de `Wrap` et les autres routines n'ont besoin que de `Unwrap`. Comme l'autre version déclarative non agrégée, nous regroupons les quatre opérations ensemble dans un module. Dans le système Mozart, certains types utilisent cette forme d'abstraction, comme par exemple les dictionnaires [23].

5.4.3 Le passage de paramètres

Les opérations d'une abstraction de données peuvent avoir des arguments et des résultats. Différents mécanismes ont été inventés pour passer les arguments et résultats entre un programme et l'abstraction qu'il appelle. Nous passons en revue les plus importants. Pour chaque mécanisme, nous donnons un exemple dans une syntaxe qui ressemble à Pascal et nous donnons sa définition dans le modèle avec état de ce chapitre. Cette définition peut être vue comme la sémantique du mécanisme. Nous utilisons le Pascal à cause de sa simplicité. Java est actuellement plus populaire, mais expliquer sa syntaxe plus élaborée n'est pas appropriée pour cette section. (La section 6.5 donne des exemples de syntaxe de Java.)

Le passage par référence (« call by reference »)

L'identité de l'entité du langage est passée à la procédure. La procédure peut ensuite utiliser l'entité librement. C'est le mécanisme de base utilisé par tous nos modèles de calcul pour toutes les entités du langage y compris les variables dataflow et les cellules.

Les langages impératifs donnent souvent un autre sens au passage par référence. Ils supposent que la référence est stockée dans une cellule locale à la procédure. Dans notre terminologie, c'est un passage par valeur où la référence est considérée comme une valeur (voir plus loin). Si vous étudiez un langage avec le passage par référence, nous recommandons d'examiner la définition du langage avec soin pour comprendre le sens exact.

Le passage par variable (« call by variable »)

C'est un cas spécial du passage par référence. L'identité d'une cellule est passée à la procédure. Voici un exemple :

```
procedure sqr(var a:integer);
begin
   a:=a*a
end
var c:integer;
c:=25;
sqr(c);
browse(c);
```

Nous codons cet exemple ainsi :

```
proc {Sqr A}
   A:=@A*@A
end
local C={NewCell 0} in
   C:=25
   {Sqr C}
   {Browse @C}
end
```

Pour l'appel `{Sqr C}`, le `A` à l'intérieur de `Sqr` est un synonyme du `C` à l'extérieur.

Le passage par valeur (« call by value »)

Une valeur est passée à la procédure et mise dans une cellule locale à la procédure. L'implémentation est libre de copier la valeur ou de passer une référence, pourvu que la procédure ne puisse pas changer la valeur dans l'environnement à l'appel. Voici un exemple :

```
procedure sqr(a:integer);
begin
   a:=a+1;
   browse(a*a)
end;
sqr(25);
```

Nous codons cet exemple ainsi :

```
proc {Sqr D}
A={NewCell D} in
   A:=@A+1
   {Browse @A*@A}
end
{Sqr 25}
```

La cellule `A` est initialisée avec l'argument de `Sqr`. Le langage Java utilise le passage par valeur pour les valeurs et les références aux objets (voir section 6.5).

Le passage par valeur-résultat (« call by value-result »)

C'est une modification du passage par variable. Quand la procédure est appelée, le contenu d'une cellule (une variable affectable) est mis dans une autre cellule locale à la procédure. Quand la procédure retourne, le contenu actuel de cette dernière est mis dans la première. Voici un exemple :

```
procedure sqr(inout a:integer);
begin
   a:=a*a
end
var c:integer;
c:=25;
sqr(c);
browse(c);
```

L'exemple utilise le mot clé « `inout` » pour indiquer le passage par valeur-résultat. Ce mot clé est utilisé dans le langage Ada. Nous codons cet exemple ainsi :

```
proc {Sqr A}
D={NewCell @A} in
   D:=@D*@D
   A:=@D
end
local C={NewCell 0} in
   C:=25
   {Sqr C}
   {Browse @C}
end
```

Il y a deux variables affectables : une à l'intérieur de `Sqr` (à savoir `D`) et une à l'extérieur (à savoir `C`). À l'entrée de `Sqr`, `D` est affectée avec le contenu de `C`. À la sortie, `C` est affectée avec le contenu de `D`. Pendant l'exécution de `Sqr`, les modifications de `D` sont invisibles de l'extérieur.

Le passage par nom (« call by name »)

Ce mécanisme est le plus complexe. Il crée une fonction pour chaque argument. L'appel de la fonction renvoie le nom d'une cellule, c'est-à-dire l'adresse d'une variable affectable. Chaque fois que l'argument est utilisé, la fonction est appelée. Une fonction utilisée de cette manière s'appelle un « ***thunk*** ».[4] Les thunks ont été inventés pour l'implémentation de Algol 60. Voici un exemple :

4. C'est le sens original de *thunk*. Le terme *thunk* est aussi utilisé de façon plus générale pour désigner toute fermeture à portée lexicale.

```
procedure sqr(callbyname a:integer);
begin
   a:=a*a
end;
var c:integer;
c:=25;
sqr(c);
browse(c);
```

L'exemple utilise le mot clé « `callbyname` » pour indiquer le passage par nom. Nous codons cet exemple ainsi :

```
proc {Sqr A}
   {A}:=@{A}*@{A}
end
local C={NewCell 0} in
   C:=25
   {Sqr fun {$} C end}
   {Browse @C}
end
```

L'argument `A` est une fonction dont l'invocation renvoie le nom d'une variable affectable. La fonction est appelée chaque fois que l'argument est utilisé. Le passage par nom peut donner des résultats non-intuitifs si des indices de tableau sont utilisés dans l'argument (voir exercices, section 5.7).

Le passage par besoin (« call by need »)

C'est une modification du passage par nom dans laquelle la fonction est appelée au maximum une fois. Son résultat est conservé pour les utilisations ultérieures. Voici une manière de coder le passage par besoin pour l'exemple de passage par nom :

```
proc {Sqr A}
B={A} in
   B:=@B*@B
end
local C={NewCell 0} in
   C:=25
   {Sqr fun {$} C end}
   {Browse @C}
end
```

L'argument `A` est évalué une fois si l'argument est utilisé. La variable locale `B` garde alors le résultat. Si on a encore besoin de l'argument, c'est `B` qui sera utilisée pour éviter

d'évaluer la fonction de nouveau. Dans l'exemple `Sqr`, c'est facile à implémenter parce qu'il est clair que le résultat est utilisé trois fois. S'il n'est pas clair par inspection du code que le résultat est nécessaire, alors l'évaluation paresseuse peut être utilisée pour implémenter le passage par besoin directement (voir exercices, section 5.7).

Le passage par besoin repose exactement sur le même concept que l'évaluation paresseuse. Le terme « passage par besoin » est plus souvent utilisé dans un langage avec l'état, où le résultat de l'évaluation peut être le nom d'une cellule (une variable affectable). Le passage par nom est l'évaluation paresseuse sans mémoisation. Le résultat de l'évaluation de la fonction n'est pas conservé. La fonction sera donc appelée chaque fois que l'on aura besoin du résultat.

Discussion

Quel est le « meilleur » de tous ces mécanismes ? Cette question a fait couler beaucoup d'encre (voir par exemple [62]). Le but de l'approche du langage noyau est de factoriser les langages de programmation dans un petit ensemble de concepts significatifs pour le programmeur. Pour le passage de paramètres, cela justifie l'utilisation du passage par référence comme mécanisme primitif. À la différence des autres, le passage par référence ne dépend pas des concepts supplémentaires comme les cellules ou les valeurs procédurales. Il a une sémantique formelle simple et il a une implémentation efficace. Par contre, cela ne veut pas dire que le passage par référence est toujours le bon mécanisme pour les programmes. D'autres mécanismes de passage de paramètres peuvent être codés en combinant le passage par référence avec les cellules et les valeurs procédurales. Beaucoup de langages offrent ces mécanismes comme des abstractions linguistiques.

5.5 LE POLYMORPHISME

Dans le langage courant, le polymorphisme est la capacité d'une entité à prendre plusieurs formes. Dans le contexte de l'abstraction de données, on dit qu'une opération est **polymorphe** si elle fonctionne correctement pour des arguments de plusieurs types.

Le polymorphisme est important dans l'organisation de programmes pour permettre la maintenance aisée. En particulier, il permet à un programme de répartir correctement des responsabilités entre ses composants [11]. Une responsabilité ne doit pas être étalée sur plusieurs composants. Elle devrait plutôt être concentrée dans un seul endroit.

Considérez l'analogie suivante : un patient consulte un médecin spécialiste. Le patient demande au médecin de le guérir. Selon la spécialité du médecin, ses actions peuvent être très différentes (comme faire une ordonnance ou préconiser une chirurgie). Dans un programme informatique, l'objet patient appelle l'opération `guérison` de l'objet médecin. Le polymorphisme signifie que l'objet médecin peut avoir plusieurs

types (spécialités), aussi longtemps qu'il comprend l'opération `guérison`. L'objet patient ne sait pas comment se guérir lui-même ; il ne *devrait* pas savoir. L'objet médecin comprend l'opération `guérison` et fera l'action appropriée.

Toutes les organisations des abstractions de données que nous avons expliquées auparavant peuvent soutenir le polymorphisme. L'idée de base est simple. Supposez qu'un programme fonctionne correctement avec une abstraction de données particulière. Alors il a le potentiel de fonctionner avec toute autre abstraction de données ayant la même interface. Mais le programme est-il toujours correct avec l'autre abstraction de données ? Il le sera si l'autre abstraction satisfait les mêmes propriétés mathématiques que l'abstraction originale. Le raisonnement qui a montré l'exécution correcte pour l'abstraction originale devrait être utilisable pour l'autre.

Le style objet présente un avantage sur le style ADT car le polymorphisme y est particulièrement facile à exprimer avec les objets. Bien que le polymorphisme puisse être exprimé dans le style ADT, il est plus encombrant car il nécessite des modules de première classe. Le style ADT a un avantage différent : il donne plus de liberté pour faire des implémentations efficaces. Donnons un exemple pour clarifier ces différences. Définissons d'abord un type Collection et implémentons-le dans les styles ADT et objet. Ajoutons ensuite une opération, `union`, au type Collection et étendons les deux implémentations.

5.5.1 Un exemple : le type Collection

Supposons que nous ayons un type Collection avec trois opérations : un `put` pour ajouter un élément, un `get` pour extraire un élément et un `isEmpty` pour tester si la collection est vide. Commençons l'exemple en implémentant le type Collection dans les styles ADT et objet. Implémentons la collection en ADT avec la pile avec état non agrégée (`NewWrapper` est définie dans la section 3.5) :

```
local Wrap Unwrap
   {NewWrapper Wrap Unwrap}
   fun {NewCollection} {Wrap {Stack.new}} end
   proc {Put C X} S={Unwrap C} in {Stack.push S X} end
   fun {Get C} S={Unwrap C} in {Stack.pop S} end
   fun {IsEmpty C} {Stack.isEmpty {Unwrap C}} end
in
   Collection=collection(new:NewCollection put:Put
                         get:Get isEmpty:IsEmpty)
end
```

Voici une exécution :

```
C={Collection.new}
{Collection.put C 1}
{Collection.put C 2}
{Browse {Collection.get C}}
{Browse {Collection.get C}}
```

Maintenant, implémentons la collection comme un objet avec la pile avec état agrégée :

```
fun {NewCollection}
   S={NewStack}
   proc {Put X} {S.push X} end
   fun {Get} {S.pop} end
   fun {IsEmpty} {S.isEmpty} end
in
   collection(put:Put get:Get isEmpty:IsEmpty)
end
```

Voici une exécution, qui fait les mêmes choses qu'avec l'ADT :

```
C={NewCollection}
{C.put 1}
{C.put 2}
{Browse {C.get}}
{Browse {C.get}}
```

5.5.2 Ajouter une opération `union` dans le cas ADT

Nous voulons étendre le type Collection avec une opération `union` qui prend tous les éléments d'une collection et les ajoute à une autre collection. Dans le style ADT on l'appelle comme `{Collection.union C1 C2}`, ce qui ajoute tous les éléments de `C2` dans `C1`, ce qui laisse `C2` vide. Pour implémenter `union`, nous utilisons une abstraction de contrôle :

```
proc {DoUntil BF S}
   if {BF} then skip else {S} {DoUntil BF S} end
end
```

`DoUntil` exécute `{S}` aussi longtemps que `{BF}` renvoie **`false`**. Avec `DoUntil`, nous implémentons le nouveau type Collection comme une extension de l'implémentation originale :

```
local Wrap Unwrap
   ...
   proc {Union C1 C2}
   S1={Unwrap C1} S2={Unwrap C2} in
      {DoUntil fun {$} {Stack.isEmpty S2} end
       proc {$} {Stack.push S1 {Stack.pop S2}} end}
   end
in
   Collection=collection(... union :Union)
end
```

Remarquez que cette implémentation a besoin des représentations internes de `C1` et `C2`, c'est-à-dire des deux piles. Voici une exécution :

```
C1={Collection.new} C2={Collection.new}
for I in [1 2 3] do {Collection.put C1 I} end
for I in [4 5 6] do {Collection.put C2 I} end
{Collection.union C1 C2}
{Browse {Collection.isEmpty C2}}
{DoUntil fun {$} {Collection.isEmpty C1} end
 proc {$} {Browse {Collection.get C1}} end}
```

Nous pouvons faire une deuxième implémentation qui n'utilise que les interfaces externes de `C1` et `C2` :

```
local Wrap Unwrap
   ...
   proc {Union C1 C2}
      {DoUntil fun {$} {Collection.isEmpty C2} end
       proc {$}
          {Collection.put C1 {Collection.get C2}} end}
   end
in
   Collection=collection(... union:Union)
end
```

En résumé, nous avons le choix d'utiliser ou non la représentation interne des collections. Nous avons la liberté de faire une implémentation plus efficace en utilisant les représentations internes, mais dans ce cas nous perdons le polymorphisme.

5.5.3 Ajouter une opération `union` dans le cas objet

Ajoutons maintenant une opération `union` à l'objet Collection. Dans le style objet on l'appelle comme `{C1 union(C2)}`. Voici l'implémentation :

```
fun {NewCollection}
   S1={NewStack}
   ...
   proc {Union C2}
      {DoUntil C2.isEmpty
       proc {$} {S1.push {C2.get}} end}
   end
in
   collection(... union:Union)
end
```

Cette implémentation utilise la représentation interne de `C1` mais l'interface externe de `C2`. C'est une différence cruciale avec le style ADT ! Pouvons-nous faire une implémentation en style objet qui utilise les deux représentations internes, comme nous l'avons fait dans le cas ADT ? La réponse est simplement non, pas sans casser l'encapsulation de l'objet `C2`.

Pour compléter le cas objet, voici une implémentation objet qui n'utilise que les interfaces externes :

```
fun {NewCollection}
   ...
   proc {Union C2}
      {DoUntil C2.isEmpty
       proc {$} {This.put {C2.get}} end}
   end
   This=collection(... union:Union)
in
   This
end
```

Remarquez que l'objet `C1` fait référence à lui-même avec la variable `This`.

5.5.4 Discussion

Comment choisir entre les styles ADT et objet ? Il faut les comparer :

- Le style ADT peut être plus efficace parce qu'il permet l'accès aux deux représentations internes. L'utilisation d'une interface externe peut être moins efficace si elle n'a pas toutes les opérations nécessaires.
- Parfois le style ADT est le seul qui soit bon. Supposons que nous définissions un type entier et que nous voulions définir l'addition de deux entiers. Si aucune autre opération n'est définie sur les entiers, il faudra un moyen pour accéder aux représentations internes. Par exemple, la représentation pourrait être en binaire

et nous pourrions faire une addition avec une instruction machine. Cela explique pourquoi des langages orientés objet populaires comme Java utilisent le style ADT pour les opérations primitives sur des types de base comme des entiers.

- Dans le style objet, le polymorphisme ne coûte « rien ». Supposons que nous définissions un deuxième type de collection (avec un objet `D`) ayant la même interface que le premier. Alors les deux types de collections peuvent interopérer même si leurs implémentations sont indépendantes. Bref, `{C union(D)}` est correct sans écrire une seule ligne de code en plus ![5]
- Le style objet n'est pas limité aux objets séquentiels. En particulier, les objets à flots (voir chapitre 4) sont aussi des objets comme nous les avons définis. Ils soutiennent le polymorphisme tout comme les objets séquentiels.
- Le style ADT pourra soutenir le polymorphisme si le langage a des modules de première classe. Supposons que nous définissions un deuxième type de collection comme un module `Collection2`. L'implémentation d'`union` doit alors s'assurer que `C2` utilise toujours une opération de `Collection2`. Nous pouvons le faire en ajoutant `Collection2` comme un argument à l'opération `union`, ce qui donne l'appel `{Union C1 Collection2 C2}`. La définition est :

```
proc {Union C1 Collection2 C2}
   {DoUntil fun {$} {Collection2.isEmpty C2} end
    proc {$}
       {Collection.put C1 {Collection2.get C2}} end}
end
Collection=collection(... union:Union)
```

  Cette technique est souvent utilisée dans les langages avec des modules de première classe, comme Erlang.
- Si nous utilisons le style ADT sans modules de première classe, il faudra écrire du code supplémentaire pour faire interopérer les types. Nous devons écrire une opération `union` qui connaît *les deux* représentations internes. Si nous avons trois ou plus de trois types de collection, l'implémentation deviendra encore plus embrouillée : il faudra implémenter toutes les combinaisons de deux types.

Les langages orientés objet utilisent le style objet par défaut, ce qui les rend polymorphes par défaut. C'est un des grands avantages de la programmation orientée objet. Nous l'explorerons davantage dans le chapitre 6.

5. Cela peut paraître miraculeux. L'appel fonctionne car l'implémentation en C de `union` n'appelle que l'interface externe de D. Réfléchissez-y !

5.5.5 D'autres formes de polymorphisme

La forme de polymorphisme utilisée dans cette section s'appelle le **polymorphisme universel**. Un deuxième concept, le **polymorphisme ad-hoc** [14], est aussi considéré comme une forme de polymorphisme. Dans le polymorphisme ad-hoc, du code différent est exécuté pour des arguments de types différents. Dans le polymorphisme universel, le même code est exécuté pour des arguments de tous les types admissibles. Un exemple du polymorphisme ad-hoc est la surcharge des opérateurs, où le même opérateur peut représenter plusieurs fonctions différentes. Par exemple, l'opérateur « + » est surchargé dans beaucoup de langages. Le compilateur choisit la fonction appropriée selon les types d'arguments (par exemple, des entiers ou des flottants).

5.6 LA PROGRAMMATION À GRANDE ÉCHELLE

Une administration efficace et réussie se manifeste également dans les petites affaires et dans les grandes.
– Mémorandum, 8 août 1943, Winston Churchill (1874-1965)

La programmation à grande échelle est la programmation par une équipe. Elle implique tous les aspects de développement de logiciel qui demandent de la communication et de la coordination entre personnes. La gestion d'une équipe de personnes est difficile à tous points de vue–voyez la difficulté pour entraîner une équipe de football. Pour la programmation, c'est particulièrement difficile car les programmes ne pardonnent pas des petites erreurs. Les programmes demandent une précision qui est difficile à satisfaire par les êtres humains. La programmation à grande échelle est souvent appelée **génie logiciel**.

Cette section est la suite de l'introduction à la programmation à petite échelle de la section 3.8. Nous expliquons comment développer des logiciels dans les petites équipes.

Nous vous recommandons les livres suivants pour approfondir les discussions de cette section sur la conception de programmes et le génie logiciel. Nous suggérons [74, 76] comme des textes généraux et [91] pour équilibrer la conception et la refactorisation. « *The Mythical Man-Month* » de Frederick Brooks date de 1975 mais est toujours relevant [12, 13]. « *Software Fundamentals* » est une collection d'articles de Dave Parnas qui couvrent sa carrière et sont toujours intéressants à lire [73]. « *The Cathedral and the Bazaar* » par Eric Raymond est une exposition intéressante du développement des logiciels libres [77]. Pour plus d'informations sur les composants, nous vous recommandons « *Component Software : Beyond Object-Oriented Programming* » par Clemens Szyperski [93].

5.6.1 La méthodologie de conception

Beaucoup de choses, vraies et fausses, ont été publiées sur la bonne méthodologie de conception pour la programmation à grande échelle. Une grande partie de la littérature existante est basée sur l'extrapolation d'expériences limitées, car la validation rigoureuse est assez difficile à faire. Pour valider une nouvelle idée, plusieurs équipes identiques devraient travailler dans des circonstances identiques. Cela a rarement été fait et nous ne l'essayons pas non plus.

Cette section résume les leçons que nous avons tirées de notre propre expérience de construction de systèmes. Nous avons conçu et construit plusieurs grands logiciels [42, 69, 96]. Nous avons beaucoup réfléchi à la bonne conception de logiciels dans une équipe et regardé comment d'autres bonnes équipes fonctionnent. Nous avons essayé de sélectionner les principes vraiment utiles.

La gestion de l'équipe

La première tâche, la plus importante, est de s'assurer que l'ensemble de l'équipe travaille de façon coordonnée. Il y a trois idées pour atteindre ce but :

1. Il faut compartimenter la responsabilité de chaque personne. Par exemple, chaque composant peut être affecté à une personne, qui en est responsable. Les responsabilités doivent respecter les frontières des composants et ne pas se chevaucher. Cela évite des discussions interminables sur qui aurait dû corriger un problème.
2. Les connaissances doivent être librement échangées et non compartimentées. Les membres de l'équipe doivent souvent échanger des informations sur les différentes parties du système. Dans l'idéal, il ne devrait pas y avoir de membre de l'équipe indispensable. Tous les changements majeurs au système doivent être discutés par des membres bien informés. Cela améliore grandement la qualité du système. Il est important aussi que le propriétaire d'un composant ait le dernier mot sur un changement de son composant. Les membres inexpérimentés de l'équipe doivent être en apprentissage chez des membres plus expérimentés. Les membres inexpérimentés deviennent expérimentés quand on leur donne des tâches spécifiques à faire, ce qu'ils doivent faire le plus indépendamment possible. C'est important pour la longévité du système.
3. Chaque interface de composant doit être soigneusement documentée, car c'est aussi l'interface entre le propriétaire du composant et les autres membres de l'équipe. La documentation est particulièrement importante, encore plus que pour la programmation à petite échelle. La bonne documentation est un pilier de stabilité qui peut être consultée par tous les membres de l'équipe.

La méthodologie du développement de logiciel

Il y a beaucoup de manières d'organiser le développement de logiciel. Par exemple, les techniques de développement descendant (« *top-down* ») et ascendant (« *bottom-up* ») ont été pleinement discutées. Aucune de ces techniques n'est vraiment satisfaisante pour de grands programmes, et les combiner n'aide pas beaucoup. Le principal problème vient de ce que les exigences et la spécification du système doivent être assez complètes dès le début. Cela est presque impossible à réaliser à moins que le système soit très bien compris. Selon notre expérience, une approche qui fonctionne bien est le **développement incrémental**, appelé aussi **développement itératif** ou **IID** (« *Iterative and Incremental Development* »). Elle s'appelle parfois développement évolutionnaire, bien que cette métaphore ne soit pas applicable parce qu'il n'y a pas d'évolution dans le sens darwinien [18].[6] Le développement incrémental a une longue histoire dans l'informatique et dans d'autres domaines. Il a été utilisé avec succès pour le développement de logiciel au moins depuis les années 1950 [9, 59]. L'approche comporte les étapes suivantes :

- Commencez avec un petit jeu d'exigences qui est un sous-ensemble du jeu complet, et construisez un système complet qui satisfait ce petit jeu. La spécification et l'architecture du système sont des « coquilles vides » : elles sont juste assez complètes pour construire un programme exécutable, mais elles ne résolvent pas les problèmes de l'utilisateur.
- Ensuite augmentez les exigences selon les commentaires des utilisateurs, en étendant la spécification et l'architecture du système au fur et à mesure pour satisfaire aux nouvelles exigences. Selon une métaphore organique, nous disons que l'application « grandit ». À tout moment, il y a un système exécutable qui satisfait sa spécification et qui peut être évalué par ses utilisateurs potentiels.
- N'optimisez pas pendant le processus de développement. Il est important de ne pas compliquer le système simplement pour augmenter les performances. Utilisez des algorithmes simples avec une complexité acceptable et gardez une conception simple avec de bonnes abstractions. Ne vous faites pas de soucis à propos du surcoût de ces abstractions. L'optimisation des performances peut être faite vers la fin du développement, mais seulement s'il y a des problèmes de performance. On peut faire un profilage du système, c'est-à-dire mesurer les temps d'exécution des différentes parties, afin de trouver les parties (typiquement petites) qui prennent un grand pourcentage du temps global et qui doivent donc être réécrites.

6. L'évolution darwinienne implique une population dans laquelle les individus meurent et naissent, une source de diversité dans les individus nouveau-nés et un filtre (la sélection naturelle) qui élimine les individus moins aptes. Il n'y a pas de population dans l'approche incrémentale.

– Réorganisez la conception selon les besoins pendant le développement, pour garder une bonne organisation des composants. Les composants doivent encapsuler les décisions prises lors de la conception ou implémenter des abstractions courantes. Cette réorganisation est appelée **refactorisation**. Il faut trouver une juste mesure entre une planification complète et une dépendance totale à la refactorisation. La meilleure approche est quelque part au milieu.

Le développement incrémental a beaucoup d'avantages :

– Les bugs de toute taille sont détectés rapidement et peuvent être corrigés tout de suite.
– La pression d'échéances est grandement soulagée car il existe toujours une application qui fonctionne.
– Les développeurs sont plus motivés car ils obtiennent des réactions rapides à leurs efforts.
– Les utilisateurs ont plus de chances d'obtenir ce dont ils ont vraiment besoin, car ils ont la possibilité d'utiliser l'application pendant le processus de développement.
– L'architecture a plus de chances d'être bonne car elle peut être corrigée rapidement.
– L'interface utilisateur a plus de chances d'être bonne car elle est continuellement améliorée pendant le processus de développement.

Pour la plupart des systèmes, même des petits, nous trouvons qu'il est presque impossible de trouver en avance les vraies exigences, une bonne architecture pour les réaliser et une bonne interface utilisateur. Le développement incrémental est valable, en partie parce qu'il fait peu d'hypothèses a priori. Pour une vue complémentaire, nous recommandons la programmation extrême, qui est une autre approche qui met l'accent sur le compromis entre planification et refactorisation [91]. Pour une vue plus radicale, nous recommandons le développement dirigé par les tests (« *test-driven development* »), qui est incrémental d'une façon complètement différente. Le développement dirigé par les tests prétend qu'il est possible d'écrire un programme sans aucune phase de conception, simplement en créant des tests et en faisant une refactorisation chaque fois que le programme ne réussit pas un nouveau test [10]. Il est évident que la conception de bons tests est cruciale pour la réussite de cette approche !

5.6.2 La structure hiérarchique d'un système

Comment le système doit-il être structuré pour soutenir le travail en équipe et le développement incrémental ? Une manière qui fonctionne en pratique est de structurer l'application comme un graphe hiérarchique avec des interfaces bien définies à chaque niveau (voir figure 5.4). L'application est un ensemble de nœuds où chaque nœud

interagit avec quelques autres nœuds. Chaque nœud est une instance d'un composant. Chaque nœud est lui-même une petite application et peut être décomposé en un graphe. La décomposition s'arrête quand nous arrivons aux composants primitifs qui sont fournis par la plate-forme de développement.

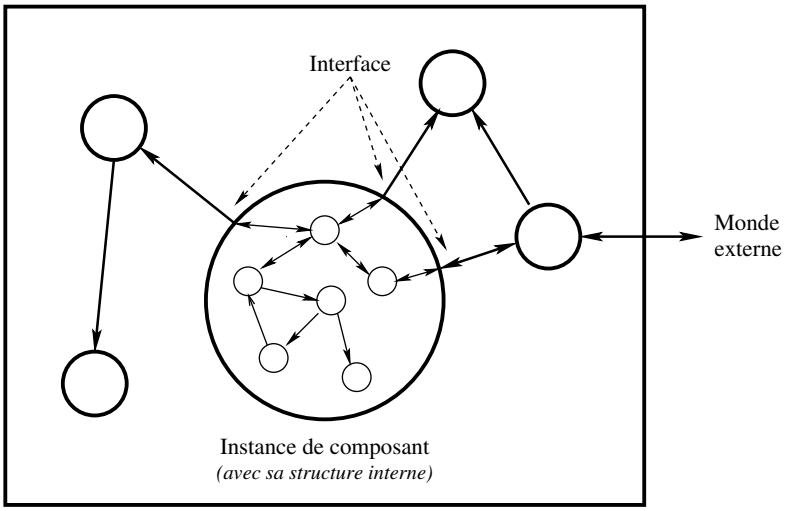


Figure 5.4 Un système structuré en graphe hiérarchique.

La connexion des composants

La première tâche dans la construction du système est la connexion des composants. Cette tâche a des aspects statique et dynamique :

- **La structure statique**. C'est le graphe de composants qui est connu quand l'application est conçue. Les liens de ces composants peuvent être édités dès le démarrage de l'application. Chaque instance de composant correspond grosso modo à un ensemble de fonctionnalité que l'on appelle une **bibliothèque** ou un **paquet**. Pour l'efficacité, nous voudrions que chaque instance de composant n'existe qu'une fois dans le système. Si une bibliothèque est utilisée par différentes parties de l'application, nous voudrons que ces parties se partagent la même bibliothèque. Par exemple, un composant peut implémenter une bibliothèque graphique ; toute l'application peut alors utiliser la même instance de ce composant.
- **La structure dynamique**. Souvent, une application fera des calculs avec des composants à l'exécution. Elle voudrait éditer les liens de nouveaux composants qui ne sont connus qu'à l'exécution. Elle voudrait calculer un nouveau composant et le sauvegarder. Les instances des composants ne sont pas forcément partagées ;

peut-être faut-il plusieurs instances d'un composant ? Par exemple, un composant peut implémenter une interface avec une base de données. S'il y a plusieurs bases de données externes, il faudra éditer les liens de plusieurs instances du composant. Cela est décidé à l'exécution, chaque fois qu'une base de données est ajoutée.

La figure 5.5 montre la structure de l'application, avec des composants dont les liens sont édités statiquement et d'autres dont les liens sont édités dynamiquement.

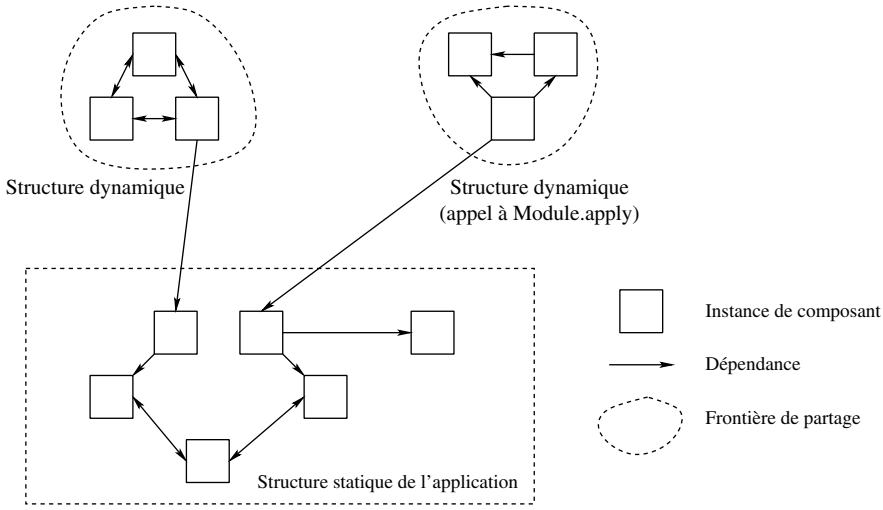


Figure 5.5 La structure statique et dynamique du système.

➤ La structure statique

Pour soutenir la structure statique, il est utile pour les composants d'être des unités de compilation qui sont stockées dans les fichiers. Nous appelons ces composants des **foncteurs** et leurs instances des **modules**. Un foncteur est une unité de compilation parce qu'il peut être compilé indépendamment d'autres foncteurs. Les dépendances du foncteur sont données comme des noms de fichier. Pour qu'il soit accessible par d'autres foncteurs, il faut stocker le foncteur dans un fichier. Cela permet aux autres foncteurs de spécifier qu'ils en ont besoin en donnant le nom du fichier.

Un foncteur a deux représentations : un code source, qui est simplement un texte, et un code compilé, qui est une valeur dans le langage. Si le code source est dans un fichier `foo.oz` alors il pourra être compilé pour donner un autre fichier `foo.ozf` qui contient le code compilé. Voici le contenu du fichier `foo.oz` :

```
functor
import OtherComp1 at File1
       OtherComp2 at File2
       ...
       OtherCompN at FileN
export op1:X1 op2:X2 ... opK:Xk
define
   % Définissez X1, ..., Xk
   ...
end
```

Ce composant dépend des autres composants `OtherComp1`, ..., `OtherCompN`, stockés respectivement dans les fichiers `File1`, ..., `FileN`. Il définit un module avec les champs `op1`, ..., `opK`, référencés par `X1`, ..., `Xk` et définis dans le corps du foncteur.

Une application est simplement un foncteur compilé. Pour exécuter l'application, tous les autres foncteurs compilés dont elle a besoin, directement ou indirectement, doivent être rassemblés et leurs liens édités. Quand l'application est exécutée, elle charge ses composants et édite leurs liens. L'édition des liens d'un composant veut dire l'évaluer avec ses modules importés comme arguments et renvoyer le résultat aux modules qui en ont besoin. L'édition des liens peut être faite au démarrage de l'application (édition statique des liens), ou au fur et à mesure selon les besoins pendant l'exécution (édition dynamique des liens). Nous trouvons que l'édition dynamique des liens est généralement préférable, si tous les foncteurs compilés sont accessibles rapidement (par exemple, s'ils sont dans le même système de fichiers). Avec ce réglage par défaut, l'application démarre rapidement et n'utilise que la quantité de mémoire dont elle a besoin.

➤ La structure dynamique

Un foncteur est une entité du langage. Si un programme contient une instruction `X=`**`functor ... end`**, alors `X` sera liée à la valeur du foncteur. Comme une procédure, le foncteur peut avoir des références externes. L'instanciation du foncteur donne un module, qui est l'ensemble d'entités du langage créé par l'initialisation du foncteur. Une interface est un enregistrement qui contient les entités du langage visibles de l'extérieur du module. Par un abus de terminologie commode, cet enregistrement est parfois appelé le module. Il existe une opération pour instancier les foncteurs :

– `Ms={Module.apply Fs}`. Avec une liste `Fs` de foncteurs (comme entités du langage), cette opération instancie tous les foncteurs et crée une liste `Ms` de modules. Dans la portée d'un appel de `Module.apply`, les foncteurs sont partagés. Si le même foncteur est mentionné plusieurs fois, ses liens seront tout de même édités une seule fois.

Chaque appel à `Module.apply` crée un ensemble de nouveaux modules. Cette opération fait partie du module `Module`. Si le foncteur est stocké dans un fichier, il faudra d'abord le charger avant d'appeler `Module.apply`.

La communication des composants

Une fois liés ensemble, les composants doivent communiquer. Nous donnons six des protocoles les plus populaires pour la communication des composants, dans l'ordre croissant du degré de l'indépendance des composants :

1. *La procédure* : l'application est séquentielle et un composant appelle l'autre comme une procédure. Le composant qui appelle n'est pas nécessairement le seul qui initie des appels ; il peut y avoir des appels imbriqués où le lieu de contrôle passe et repasse entre les composants. Mais il n'y a qu'un seul lieu de contrôle global, ce qui lie fortement les deux composants.
2. *La coroutine* : plusieurs composants s'exécutent indépendamment, mais dans un contexte séquentiel. Cela introduit le concept d'une coroutine. Chaque fois qu'un composant en appelle un autre, l'autre continue où il s'était arrêté. Il y a plusieurs lieux de contrôle, un par composant. Cette organisation est plus relâchée que la précédente, mais les composants sont toujours dépendants parce qu'ils s'exécutent en alternance.
3. *L'interaction concurrente et synchrone* : chaque composant s'exécute de façon indépendante et peut initier et terminer les communications avec un autre composant, selon un protocole accepté par les deux. Les composants sont concurrents. Il y a plusieurs lieux de contrôle, les fils, qui s'exécutent indépendamment (voir chapitre 4). Chaque composant fait des appels synchrones, c'est-à-dire que chaque appel attend une réponse avant de continuer.
4. *L'interaction concurrente et asynchrone* : un ensemble de composants concurrents qui communiquent par des canaux asynchrones. Chaque composant envoie des messages aux autres, mais ne doit pas attendre une réponse avant de continuer. Les canaux peuvent avoir un ordre premier entré premier sorti (FIFO, « *first-in, first-out* »), où les messages sont reçus dans l'ordre de leur envoi, ou ne pas avoir d'ordre. Les canaux s'appellent des flots dans le chapitre 4. Dans cette organisation, chaque composant connaît l'identité du composant avec lequel il communique.
5. *La boîte à lettres concurrente* est une variation du protocole précédent. Les canaux asynchrones se comportent comme des boîtes à lettres. Il est possible d'extraire un message d'un canal (par exemple, en utilisant la correspondance par formes) sans perturber les messages qui restent. C'est une organisation très utile pour beaucoup de programmes concurrents. Cette technique est utilisée

dans le langage Erlang, qui a des boîtes à lettres FIFO. Les boîtes à lettres sans ordre sont possibles aussi.

6. *Le modèle de coordination* : les composants peuvent communiquer sans que les expéditeurs et les destinataires connaissent les identités des autres. Une abstraction qui s'appelle un **espace de tuples** (« *tuple space* ») est placée à l'interface. Les composants sont concurrents et chacun interagit uniquement avec l'espace de tuples qui est commun entre tous. Un composant peut insérer un message asynchrone et un autre peut extraire le message.

Le principe d'indépendance des modèles

Chaque composant du système est écrit dans un modèle de calcul qui lui est propre. Pendant le développement, la structure interne d'un composant peut changer radicalement. Il n'est pas rare pour le composant de changer de modèle de calcul. Un composant sans état peut acquérir un état (ou devenir concurrent, ou réparti, etc.), ou vice versa. Si un tel changement survient à l'intérieur d'un composant, il ne sera pas nécessaire de changer son interface. L'interface doit changer seulement si la fonctionnalité visible de l'extérieur change. C'est une propriété de modularité importante des modèles de calcul. Aussi longtemps que l'interface reste la même, cette propriété garantit qu'il n'est pas nécessaire de changer quoi que ce soit dans le reste du système. Nous considérons cette propriété comme un principe de conception fondamental des modèles de calcul :

Le principe d'indépendance des modèles

L'interface d'un composant est indépendante du modèle de calcul utilisé pour implémenter le composant. L'interface dépend uniquement de la fonctionnalité visible de l'extérieur du composant.

Un bon exemple de ce principe est la mémoisation. Supposons que le composant est une fonction d'un argument. Si le calcul est long, on pourra réduire le temps d'exécution avec une mémoire cache qui contient des paires (argument, résultat). Quand la fonction est appelée, on vérifie d'abord si l'argument est dans la mémoire cache. Si oui, on renvoie le résultat directement sans faire le calcul. Si non, on fait le calcul et on ajoute une nouvelle paire (argument, résultat) à la mémoire cache. La mémoire cache est un état explicite ; ajouter la mémoisation à un composant signifie que le composant change du modèle déclaratif au modèle avec état. Le principe d'indépendance des modèles implique que l'on peut faire cela sans changer autre chose dans le programme.

5.6.3 La maintenance

Une fois que le système est construit et fonctionne bien, il faut s'assurer qu'il continue de bien fonctionner. Maintenir un système en bon état de fonctionnement après son déploiement s'appelle la **maintenance**. Quelle est la meilleure manière de structurer les systèmes pour faciliter la maintenance ? Nous donnons quelques principes importants tirés de notre expérience. Nous considérons le problème du point de vue des composants et du point de vue du système.

La conception des composants

Il y a des bonnes et des mauvaises manières pour concevoir des composants. Une mauvaise manière est de construire un organigramme et de le découper en morceaux devenant chacun un composant. Il est beaucoup mieux de concevoir un composant comme une abstraction. Par exemple, supposons que nous écrivions un programme qui utilise les listes. Il faut rassembler les opérations sur les listes dans un composant qui définit l'abstraction des listes. Avec cette conception, les listes peuvent être implémentées, déboguées, changées et étendues sans toucher le reste du programme. Si nous voulons utiliser des listes qui sont trop grandes pour la mémoire, il suffira de changer le composant des listes pour qu'il les enregistre dans des fichiers plutôt qu'en mémoire.

➤ L'encapsulation des décisions de conception

Plus généralement, nous pouvons dire qu'un composant devrait encapsuler une décision de conception.[7] Ainsi, quand la décision de conception est changée, il ne faut changer que ce composant. C'est une forme puissante de modularité. L'utilité d'un composant peut être évaluée par rapport à l'étendu des changements qu'il peut accommoder. Par exemple, considérez un programme qui calcule avec des caractères, comme l'application qui compte la fréquence des mots de la section 3.8.4. Idéalement, la décision quant au code de caractères à utiliser (par exemple, l'ASCII, le Latin-1 ou l'Unicode) doit être encapsulée dans un composant. Cela simplifie le changement du code.

➤ La stabilité des interfaces des composants

Un composant peut être modifié en changeant son implémentation ou son interface. Changer l'interface est problématique car tous les composants qui dépendent de l'interface doivent être réécrits ou recompilés. Il faut donc éviter de changer l'interface. Mais en pratique, les changements d'interface ne peuvent pas être évités pendant la conception d'un composant. Tout ce que l'on peut faire est de réduire leur fréquence.

7. De façon plus romantique, on dit parfois que le composant a un « secret ».

Cela veut dire que les interfaces des composants souvent utilisés doivent être conçues soigneusement dès le début.

Voici un exemple simple pour clarifier cette règle. Considérez un composant qui trie des listes de chaînes de caractères. Il peut changer son algorithme de tri sans changer son interface. Souvent on peut faire cela sans recompiler le reste du programme, simplement en éditant les liens du nouveau composant. Par contre, si le code de caractères est changé, il faudra peut-être changer l'interface. Par exemple, si l'espace mémoire occupé par chaque caractère change d'un à deux octets (si l'ASCII est remplacé par l'Unicode). Cela demande une recompilation de tous les composants qui utilisent le composant changé (directement ou indirectement), car le code compilé peut dépendre du code de caractères. La recompilation peut être pénible ; changer un composant de dix lignes pourra impliquer la recompilation de la plupart du programme si le composant est souvent utilisé.

La conception du système

➤ Le moins possible de dépendances externes

Un composant qui dépend d'un autre, c'est-à-dire qui a besoin de l'autre pour fonctionner, est une source de problèmes de maintenance. Si l'un est changé, l'autre devra être changé aussi. C'est une source majeure de « délabrement de logiciel » : un logiciel qui a fonctionné arrête de fonctionner. Par exemple, LaTeX 2_ε est un système populaire de typographie dans la communauté scientifique, connu pour la grande qualité de ses résultats [58]. Un document LaTeX 2_ε peut avoir des liens vers d'autres fichiers pour personnaliser et étendre ses capacités. Certains de ces fichiers, appelés paquets (« *packages* »), sont relativement standardisés et stables. D'autres sont simplement des personnalisations locales, appelées fichiers de style. Selon notre expérience, il est très mauvais pour les documents LaTeX 2_ε d'avoir des liens aux fichiers de style dans d'autres répertoires, parfois des répertoires globaux. Si ceux-ci sont changés, les documents ne pourront plus être traités. Pour aider la maintenance, il est de loin préférable d'avoir des copies des fichiers de style dans chaque répertoire de document. Cela satisfait un invariant simple : on peut toujours typographier chaque document (selon le principe : un logiciel qui fonctionne fonctionnera). Cet invariant est un énorme avantage qui l'emporte de loin sur les deux inconvénients : (1) la mémoire supplémentaire nécessaire pour les copies et (2) la possibilité qu'un document puisse utiliser un fichier de style vieilli. Si un fichier de style est mis à jour, le programmeur est libre d'utiliser la nouvelle version dans le document, mais seulement si c'est nécessaire. Entre temps, le document reste cohérent. Un deuxième avantage est qu'il est facile d'envoyer le document d'une personne à une autre, parce qu'il est indépendant du reste du système.

➤ Le moins possible de niveaux d'indirection

Cette règle est apparentée à la précédente. Quand A a un pointeur vers B, alors une mise à jour de B exige une mise à jour de A. Toute indirection est une dépendance. L'idée est d'éviter que le pointeur devienne détaché : sa destination n'a plus de sens pour la source. Toute action sur B peut causer le détachement du pointeur de A. Le pointeur de A n'est pas connu par B et il ne peut donc pas empêcher cela. Une solution provisoire est de ne jamais changer B, mais uniquement d'en faire des copies modifiées. Cela pourra être valable si le système fait de la gestion automatique de mémoire.

Deux exemples typiques des pointeurs problématiques sont des liens symboliques dans un système de fichiers Unix et les URL. Les liens symboliques sont pernicieux pour la maintenance d'un système. Ils sont commodes parce qu'ils peuvent référencer d'autres arborescences montées dans un système de fichiers, mais en fait ils sont une grande source de problèmes. Les URL sont connus pour être peu sûrs. Ils sont souvent référencés dans les documents imprimés, mais leur durée de vie est généralement beaucoup plus courte que celle du document. C'est parce qu'ils peuvent devenir détachés rapidement mais aussi parce que l'Internet n'a pas une grande qualité de service.

➤ Des dépendances prévisibles

Prenons par exemple la commande « localiser » qui garantit de récupérer un fichier sur un réseau et d'en faire une copie locale. Son comportement est simple et prévisible, ce qui n'est pas le cas pour la mémoire cache dans les navigateurs Web. Le terme cache est mal choisi parce qu'une vraie mémoire cache maintient une cohérence entre l'entité originale et la copie. Pour toute mémoire cache, la politique de remplacement des pages doit être clairement indiquée.

➤ La prise des décisions au bon niveau

Une décision temporisée (« *time out* ») doit être prise au bon niveau. Il est faux d'implémenter une décision temporisée de façon irrévocable à un bas niveau du système (un composant fortement imbriqué), qui propage jusqu'au niveau le plus haut sans aucun mécanisme pour permettre aux composants intermédiaires d'intervenir. Ce comportement court-circuite tous les efforts du concepteur de l'application pour masquer ou résoudre le problème.

➤ La documentation des violations

Quand un de ces principes est violé, peut-être pour une bonne raison (par exemple, une contrainte physique telle qu'une limitation de mémoire ou une séparation géographique qui forcent l'existence d'un pointeur), alors il faudra le documenter ! Toutes les dépendances externes, tous les niveaux d'indirection, toutes les dépendances imprévisibles et toutes les décisions irrévocables doivent être documentés.

➤ La hiérarchie de regroupement simple

Un logiciel ne doit pas être stocké de façon dispersé dans un système de fichiers, mais doit être regroupé le plus possible. Nous définissons une hiérarchie simple de regroupement des composants d'un système. Nous avons trouvé cette hiérarchie utile pour les documents et pour les applications. Nous commençons par l'élément le plus facile à maintenir. Par exemple, si l'application est stockée dans un système de fichiers, nous pourrons définir l'ordre suivant :

1. Si possible, mettez toute l'application dans un fichier. Le fichier peut être structuré en sections qui correspondent aux composants.
2. Si ce n'est pas possible (par exemple, parce qu'il y a des fichiers de types différents ou parce que différentes personnes travaillent sur différentes parties simultanément), alors mettez toute l'application dans un répertoire.
3. Si ce n'est pas possible (par exemple, parce que l'application doit être compilée pour plusieurs plate-formes), mettez l'application dans une arborescence avec une racine.

5.6.4 Les développements futurs

Les composants et l'avenir de la programmation

L'utilisation augmentée de composants est en train de changer la profession de programmeur. Il y a deux directions principales de changement. Premièrement, la programmation deviendra accessible aux utilisateurs finaux et pas seulement aux développeurs professionnels. Avec un jeu de composants à un haut niveau d'abstraction et une interface graphique intuitive, un utilisateur peut faire beaucoup de tâches de programmation lui-même. Cette possibilité existe déjà dans certains créneaux d'applications comme les statistiques, le traitement de signal et le contrôle des expériences scientifiques. La tendance incluera tôt ou tard des applications pour le grand public.

Deuxièmement, la programmation changera pour les développeurs professionnels. Comme de plus en plus de composants utiles seront développés, la granularité de la programmation augmentera. Les éléments de base utilisés par les programmeurs seront plus souvent de grands composants au lieu de petites opérations d'un langage. Cette tendance est visible dans des outils de programmation comme Visual Basic et dans les environnements de composants comme Enterprise Java Beans. Le goulot d'étranglement qui limite cette évolution est la spécification du comportement d'un composant. Les composants actuels sont souvent trop compliqués et ont des spécifications floues, ce qui limite leur utilisation potentielle. La solution, pour nous, est de simplifier les composants, de mieux factoriser leurs fonctionnalités et d'améliorer leurs connexions.

La conception non compositionnelle

La composition hiérarchique peut sembler être une manière très naturelle de structurer un système. Mais ce n'est pas « naturel » du tout ! La nature utilise une approche très différente, que l'on pourrait appeler non compositionnelle. Comparons les deux. Comparons d'abord leurs graphes de composants. Dans un graphe de composants, chaque nœud représente un composant et il y a une arête entre deux nœuds si les composants se connaissent. Dans un système compositionnel, ce graphe est hiérarchique. Chaque composant est connecté à quelques composants du même niveau, à ses enfants et à ses parents. Le résultat est que le système peut être décomposé en des parties indépendantes, de sorte que les interfaces entre les parties soient petites.

Dans un système non compositionnel, le graphe de composants n'a pas cette structure. Le graphe tend à être touffu et non local. « Touffu » signifie que chaque composant est connecté à beaucoup d'autres. « Non local » signifie que chaque composant est connecté à des endroits largement dispersés dans le graphe. La décomposition du système devient plus arbitraire. Les interfaces entre les parties tendent à être plus grandes. Cela rend la compréhension des composants difficile sans tenir compte de leurs relations avec le reste du système. Un exemple d'un graphe non compositionnel est un **graphe de petit monde** (« *small world graph* »), dont le diamètre est petit (chaque composant est à quelques pas de chaque autre composant).

La composition hiérarchique semble plus appropriée pour la conception de systèmes par les êtres humains. La contrainte principale pour les humains est la taille limitée de la mémoire à court terme. Un être humain ne peut garder qu'un petit nombre de concepts simultanément dans son esprit [66]. Une grande conception doit donc être découpée en parties suffisamment petites pour rester dans l'esprit d'un individu. Sans aide externe, cela mène les humains à construire des systèmes compositionnels. Par contre, la conception par la nature n'a pas cette limitation. Elle utilise le principe de sélection naturelle. De nouveaux systèmes sont construits en combinant et en modifiant des systèmes existants. Chaque système est jugé comme un tout selon sa capacité de survivre dans l'environnement naturel. Les systèmes les plus pérennes sont ceux avec le plus de progéniture. Les systèmes naturels tendent donc à être non compositionnels.

Il semble que chaque approche, dans sa forme pure, soit un extrême. La conception par les humains est orientée but et réductionniste. La conception par la nature est exploratoire et holistique. Nous pouvons essayer de chercher une approche mixte qui a les avantages des deux. Nous pouvons imaginer la construction d'outils qui permettent aux êtres humains d'utiliser une approche plus « naturelle » pour la conception de systèmes. Dans ce livre, nous n'explorons pas cette direction. Nous nous concentrons sur l'approche compositionnelle.

5.7 EXERCICES

➤ **Exercice 1** — *L'importance des séquences*

La section 5.1 donne une définition de l'état. Pour cet exercice, comparez et contrastez cette définition avec la définition suivante de la bande dessinée (« *comics* » en anglais) donnée par Scott McCloud [64] :

> **com·ics** (kom'iks) **n. 1.** Dessins picturaux et autres, juxtaposés en séquence délibérée, avec l'intention de transmettre des informations et/ou de produire une réaction esthétique chez le spectateur. . . .

Indices : Sommes-nous intéressés par toute la séquence ou uniquement par le résultat final ? La séquence existe-t-elle en temps ou en espace ? La transition entre les éléments de la séquence est-elle importante ?

➤ **Exercice 2** — *L'état avec des cellules*

La section 5.1 définit la fonction `SumList`, qui a un état codé comme les valeurs successives de deux arguments aux appels récursifs. Pour cet exercice, réécrivez `SumList` pour que l'état ne soit plus codé dans les arguments mais dans des cellules.

➤ **Exercice 3** — *L'émulation de l'état avec la concurrence*

Cet exercice explore si la concurrence peut être utilisée pour obtenir l'état explicite.

a) D'abord utilisez la concurrence pour créer un conteneur affectable. Nous créons un fil qui utilise une procédure récursive pour lire un flot. Le flot contient deux commandes possibles : `access(X)`, qui lie `X` au contenu actuel du conteneur, et `assign(X)`, qui met `X` comme nouveau contenu. Voici la définition :

```
fun {MakeState Init}
   proc {Loop S V}
      case S of access(X)|S2 then
         X=V {Loop S2 V}
      [] assign(X)|S2 then
         {Loop S2 X}
      else skip end
   end S
in
   thread {Loop S Init} end S
end
```

L'appel `S={MakeState 0}` crée un nouveau conteneur avec un contenu initial 0. Nous utilisons le conteneur avec les commandes dans le flot. Par exemple, voici une séquence de trois commandes pour le conteneur `S` :

```
declare S1 X Y in
S=access(X)|assign(3)|access(Y)|S1
```

X est liée à 0 (le contenu initial), 3 est mis dans le conteneur, et enfin Y est liée à 3.

b) Maintenant, réécrivez SumList pour utiliser ce conteneur pour compter le nombre d'appels. Le conteneur peut-il être encapsulé, c'est-à-dire peut-on l'ajouter sans changer les arguments de SumList ? Pourquoi oui ou pourquoi non ? Que se passe-t-il quand nous essayons d'ajouter la fonction SumCount comme dans la section 5.1.2 ?

➤ **Exercice 4** — *L'état explicite et la sécurité*

La section 5.4 montre quatre manières de construire des abstractions de données sécurisées. D'après ces constructions, il semble que la capacité de sécuriser des abstractions est une conséquence de l'utilisation d'un ou plusieurs des trois concepts suivants : les valeurs procédurales (qui permettent l'encapsulation avec la portée lexicale), les noms (qui sont infalsifiables et que l'on ne peut pas deviner) et les chunks (qui permettent un accès sélectif). En particulier, l'état explicite ne semble pas jouer un rôle par rapport à la sécurité. Pour cet exercice, réfléchissez bien sur cette assertion. Est-elle vraie ? Pourquoi oui ou pourquoi non ?

➤ **Exercice 5** — *Les objets déclaratifs et l'identité*

La section 5.4.2 montre comment construire un objet déclaratif qui combine les valeurs et les opérations de manière sécurisée. Pourtant, l'implémentation manque un aspect des objets, à savoir leur identité. Un objet doit garder la même identité après les changements d'état. Pour cet exercice, étendez les objets déclaratifs de la section 5.4.2 pour avoir une identité.

➤ **Exercice 6** — *Les abstractions et la gestion de mémoire*

Considérez l'ADT avec état suivant qui permet de collectionner des informations en une liste. L'ADT a trois opérations. L'appel C={NewCollector} crée un nouveau collecteur C. L'appel {Collect C X} ajoute X à la collection de C. L'appel L={EndCollect C} renvoie la liste finale qui contient tous les éléments collectionnés dans l'ordre de leur collection. Voici deux manières d'implémenter les collecteurs que nous comparerons :

– C est une cellule qui contient une paire H|T, où H est la tête de la liste collectionnée et T est sa queue non liée. Collect est implémentée ainsi :

```
proc {Collect C X}
H T in
   {Exchange C H|(X|T) H|T}
end
```

Implémentez les opérations `NewCollector` et `EndCollect` avec cette représentation.

– `C` est une paire `H|T`, où `H` est la tête de la liste collectionnée et `T` est une cellule qui contient sa queue non liée. `Collect` est implémentée ainsi :

```
proc {Collect C X}
T in
   {Exchange C.2 X|T T}
end
```

Implémentez les opérations `NewCollector` et `EndCollect` avec cette représentation.

– Nous comparons les deux implémentations par rapport à la gestion de mémoire. Utilisez le tableau de la section 3.6.2 pour calculer combien de mots de mémoire sont alloués par chaque version de `Collect`. Dans chaque version, combien de ces mots deviennent inactifs tout de suite ? Que cela implique-t-il pour le ramassage de miettes ? Quelle version est la meilleure ?

Cet exemple est tiré de la conception du système Mozart. La collection dans la boucle **`for`** a d'abord été implémentée avec une version. Cette implémentation a ensuite été remplacée par l'autre. (Remarquez que les deux versions fonctionnent correctement dans un contexte concurrent, c'est-à-dire si `Collect` est appelée à partir de plusieurs fils.)

➤ **Exercice 7** — *Le passage par nom*

La section 5.4.3 montre comment coder le passage par nom dans le modèle avec état. Pour cet exercice, considérez l'exemple suivant, qui vient de [26] :

```
procedure swap(callbyname x,y:integer);
var t:integer;
begin
   t:=x; x:=y; y:=t
end;
var a:array [1..10] of integer;
var i:integer;
i:=1; a[1]:=2; a[2]=1;
swap(i, a[i]);
writeln(a[1], a[2]);
```

Cet exemple montre un comportement curieux du passage par nom. L'exécution de l'exemple ne permute pas `i` et `a[i]`, comme on pourrait s'y attendre. Cela montre une interaction indésirable entre l'état explicite et l'évaluation retardée d'un argument.

– Expliquez cet exemple avec vos connaissances du passage par nom.

– Codez l'exemple dans le modèle de calcul avec état. Utilisez le codage suivant pour `array[1..10]` :

```
A={MakeTuple array 10}
for J in 1..10 do A.J={NewCell 0} end
```

Codez le tableau comme un tuple de cellules.

– Expliquez de nouveau le comportement en utilisant votre codage.

➤ **Exercice 8** — *Le passage par besoin*

Avec le passage par nom, l'argument est évalué chaque fois que l'on en a besoin. Avec le passage par besoin, il est évalué au maximum une fois.

– Pour cet exercice, reprenez l'exemple de la permutation de l'exercice précédent avec le passage par besoin au lieu du passage par nom. Le comportement curieux arrive-t-il toujours ? Si non, des problèmes similaires peuvent-ils arriver avec le passage par besoin en changeant la définition de `swap` ?

– Dans l'exemple du passage par besoin de la section 5.4.3, le corps de `Sqr` appellera toujours la fonction `A`. C'est raisonnable pour `Sqr`, puisque par inspection nous voyons que le résultat est utilisé trois fois. Mais que fera-t-on si on ne peut pas déterminer le besoin par inspection ? Nous ne voulons pas appeler `A` sans besoin. Une possibilité est d'utiliser une fonction paresseuse. Pour cet exercice, faites les choses suivantes :

  • Apprenez à écrire des fonctions paresseuses en étudiant la documentation appropriée [35, 97].

  • Modifiez le code de la section 5.4.3 avec l'exécution paresseuse pour appeler `A` uniquement si on en a besoin, même si on ne peut pas déterminer ce besoin par inspection. `A` doit être appelée au maximum une fois.

➤ **Exercice 9** — *L'instruction* `break`

Un bloc est une séquence d'instructions avec un point d'entrée et un point de sortie. Beaucoup de langages impératifs modernes, comme le Java et le C++, sont basés sur le concept de bloc. Ces langages permettent la définition de blocs imbriqués et offrent une opération pour sortir immédiatement jusqu'au point de sortie du bloc le plus imbriqué. Cette opération s'appelle `break`. Pour cet exercice, définissez une construction de bloc avec une opération `break`, qui peut être appelée ainsi :

```
{Block proc {$ Break} ⟨stmt⟩ end}
```

Cela doit avoir exactement le même comportement que l'exécution de ⟨stmt⟩, sauf que l'exécution de `{Break}` à l'intérieur de ⟨stmt⟩ doit immédiatement sortir du bloc. Votre solution doit fonctionner correctement pour les blocs imbriqués et pour les exceptions levées dans les blocs. Si ⟨stmt⟩ crée des fils, alors ceux-ci ne devront pas être affectés par l'opération `break`.

Indice : Utilisez le mécanisme de traitement d'exceptions.

➤ **Exercice 10** — (exercice avancé) *Le développement dirigé par les tests*
La section 5.6.1 explique pourquoi le développement incrémental est une bonne idée. La section se termine avec une courte mention du développement dirigé par les tests, une approche plus radicale qui est incrémentale aussi à sa manière. Pour cet exercice, explorez le développement dirigé par les tests et comparez-le avec le développement incrémental. Développez une ou plusieurs applications avec le développement dirigé par les tests et essayez de trouver une approche équilibrée qui combine les bonnes propriétés des deux techniques de développement.

Chapitre **6**

La programmation orientée objet

Le fruit est trop bien connu pour nécessiter une description de ses caractéristiques extérieures.
– De l'article « Apple » (Pomme), Encyclopædia Britannica, 11e édition

La programmation orientée objet (POO, en anglais « *Object-Oriented Programming* » ou OOP) est un des domaines les plus dynamiques de l'informatique. Depuis ses origines dans les années 1960, elle a envahi tous les domaines de l'informatique, dans la recherche scientifique et le développement technologique. Le premier langage orienté objet était Simula 67, qui a été développé en 1967 comme un descendant de Algol 60 [70, 71, 75]. Pourtant, la POO n'est pas devenue populaire dans l'industrie jusqu'à l'apparition de C++ au début des années 1980 [90]. Une autre étape importante fut Smalltalk-80, qui est sortie en 1980 comme le résultat des recherches menées dans les années 1970 [28]. Les deux langages C++ et Smalltalk sont directement influencés par Simula [49, 89]. Les langages les plus populaires actuellement, Java et C++, sont tous les deux orientés objet [90, 92]. Les aides à la conception les plus populaires, soi-disant indépendantes du langage, à savoir UML (« *Unified Modeling Language* », langage unifié de modélisation) et les motifs de conception (« *design patterns* »), font tous les deux l'hypothèse implicite que le langage sous-jacent est orienté objet [27, 79].

Avec tous ces développements, on pourrait penser que la POO est bien comprise (voir l'épigraphe de ce chapitre). Mais c'est loin d'être le cas. Le but de ce chapitre n'est pas de couvrir toute la POO en moins de 100 pages. C'est impossible. Nous donnerons par contre une introduction qui insiste sur les domaines où les autres présentations sont faibles : la relation avec d'autres modèles de calcul, la sémantique précise et les possibilités du typage dynamique. Nous motiverons aussi les choix de conception qui sont faits par la POO et les compromis implicites dans ces choix.

Les principes de la programmation orientée objet

Le modèle de calcul de la POO est le modèle avec état du chapitre 5. Le premier principe de la POO est que les programmes sont des collections d'abstractions de données qui interagissent. Dans la section 5.4 nous avons vu une diversité plutôt déconcertante des manières de construire des abstractions de données. La programmation orientée objet met de l'ordre dans cette diversité. Elle postule deux principes :

1. Les abstractions de données doivent avoir de l'état explicite par défaut. L'état est important à cause de la modularité du programme (voir section 5.2). Il rend possible le développement de programmes en tant que parties indépendantes qui peuvent être étendues sans changer leurs interfaces. Le principe opposé (déclaratif par défaut) est raisonnable aussi, parce qu'il facilite le test et le raisonnement et il est plus naturel pour la programmation répartie. Pour nous, les abstractions avec et sans état doivent être toutes les deux aussi faciles à utiliser.
2. Le style objet (PDA) de l'abstraction de données doit être le défaut. Le style objet est important parce qu'il encourage le polymorphisme et l'héritage. Le polymorphisme a été expliqué dans la section 5.4. Il permet à un programme de répartir correctement les responsabilités entre ses composants. L'héritage est un nouveau concept que nous introduisons. Il permet la construction incrémentale des abstractions. Nous ajoutons une abstraction linguistique, qui s'appelle une classe, afin de soutenir l'héritage dans le langage.

Pour résumer, nous pouvons caractériser librement la programmation orientée objet comme la programmation avec les abstractions de données dans le style objet, avec l'état explicite, le polymorphisme et l'héritage.

La structure du chapitre

- La section 6.1 explique le concept de l'héritage. Nous y introduisons et motivons le concept de manière générale et nous le situons par rapport aux autres concepts de structuration de programmes.
- Les sections 6.2 et 6.3 introduisent un modèle de calcul orienté objet avec le concept de classe. Nous définissons un système à objets simple qui permet l'héritage simple et multiple des classes avec des liens statiques et dynamiques. Le système à objets utilise le typage dynamique pour combiner la simplicité et la souplesse. Les classes sont des valeurs, les messages et les attributs sont des entités de première classe et des portées arbitraires peuvent être programmées. Cette souplesse nous permet de mieux explorer les limites de la POO et de situer les langages existants par rapport à notre système à objets. Nous donnons au système à objets une implémentation efficace et un soutien syntaxique pour faciliter son usage.

– La section 6.4 explique les concepts et techniques de base pour programmer avec l'héritage. Nous y introduisons la propriété de substitution (le principe le plus important), la conception par contrat, l'héritage multiple, les diagrammes de classe et les motifs de conception. Nous illustrons ces concepts avec des exemples réalistes. Nous donnons des références à la littérature sur la conception orientée objet.
– Enfin, la section 6.5 introduit le langage Java, un langage orienté objet populaire. Nous survolons la partie séquentielle de Java. Nous montrons comment les concepts de Java s'accordent avec le système à objets du chapitre.

Pour plus d'informations sur les techniques et principes de la programmation orientée objet, nous recommandons le livre « *Object-Oriented Software Construction* » de Bertrand Meyer [65]. Ce livre est particulièrement intéressant pour sa présentation détaillée de l'héritage, y compris de l'héritage multiple.

6.1 L'HÉRITAGE

L'héritage est basé sur l'observation suivante : les abstractions de données ont souvent beaucoup en commun. Prenons l'exemple des ensembles. Il y a beaucoup d'abstractions qui ressemblent aux ensembles, c'est-à-dire qu'elles sont des collections auxquelles nous pouvons ajouter et enlever des éléments. Parfois nous voudrions qu'elles soient comme des piles, avec un comportement dernier entré premier sorti (DEPS) (LIFO, « *last-in, first-out* »). Parfois nous voudrions qu'elles soient comme des files, avec un comportement premier entré premier sorti (PEPS) (FIFO, « *first-in, first-out* »). Parfois l'ordre des entrées et sorties n'est pas important. Et ainsi de suite, avec beaucoup d'autres possibilités. Toutes ces abstractions se partagent la propriété de base d'être une collection d'éléments. Pouvons-nous les implémenter sans dupliquer les parties communes ? Les parties dupliquées allongent le programme, mais ce n'est pas tout. Elles sont un cauchemar pour le programmeur et le mainteneur, car si une des copies est changée, il faudra changer les autres aussi. Comme les différentes copies sont généralement légèrement différentes, les relations entre tous ces changements sont obscures.

Nous introduisons le concept d'héritage pour réduire le problème de la duplication du code et clarifier les relations entre les différentes abstractions. Une abstraction de données peut « hériter » d'une ou de plusieurs autres abstractions de données, c'est-à-dire avoir substantiellement la même implémentation que les autres, avec peut-être quelques extensions et modifications. Seules les différences entre l'abstraction de données et ses ancêtres doivent être spécifiées. Une telle définition incrémentale d'une abstraction de données s'appelle une **classe**.

Une nouvelle classe est définie par une sorte de transformation : une ou plusieurs classes existantes sont combinées avec une description des extensions et modifications

nécessaires pour donner la nouvelle classe. Les langages orientés objet soutiennent cette transformation en définissant les classes comme une abstraction linguistique. La transformation peut être vue comme une manipulation syntaxique, où la syntaxe de la nouvelle classe peut être dérivée des classes originales (voir section 6.3). Dans le système à objets de ce chapitre, la transformation a aussi un sens sémantique. Comme les classes sont des valeurs, la transformation peut être vue comme une fonction qui prend des valeurs de classe comme entrées et qui renvoie un nouvelle valeur de classe comme sortie.

Quoique prometteur, l'expérience montre que l'héritage est un concept à utiliser avec beaucoup de prudence. D'abord, la transformation doit être définie avec une connaissance intime des classes ancêtres, parce qu'elle peut facilement briser un invariant de classe. Un autre problème est que l'héritage ouvre une nouvelle interface à un composant. La possibilité d'étendre une classe peut être vue comme une nouvelle manière d'interagir avec cette classe. Il faut maintenir cette interface pendant toute la durée de vie du composant. Pour cette raison, le défaut quand on définit une classe devrait être qu'elle est finale, c'est-à-dire qu'il est interdit pour d'autres classes d'en hériter. Rendre une classe (ou une partie d'une classe) extensible par l'héritage doit être une action explicite par le programmeur de la classe.

L'héritage augmente les possibilités de factoriser une application, c'est-à-dire de faire en sorte que les parties communes n'existent qu'une fois, mais le prix à payer est que l'implémentation de l'abstraction est étalée sur des parties différentes du programme. L'implémentation n'existe pas à un endroit précis ; toutes les abstractions dont elle hérite doivent être considérées ensemble. Cela rend la compréhension de l'implémentation plus difficile et paradoxalement peut rendre la maintenance plus difficile. De plus, une abstraction peut hériter d'une classe qui existe seulement en tant que code compilé, sans aucun accès au code source. L'héritage doit donc être utilisé avec parcimonie.

Une alternative est d'utiliser la programmation par composants, c'est-à-dire d'utiliser les composants directement et de les composer. L'idée est de définir un composant qui encapsule un autre composant et qui fournit une fonctionnalité modifiée. Il y a un choix entre l'héritage et la composition de composants : l'héritage est plus souple mais il peut briser un invariant de classe, tandis que la composition de composants est moins souple mais ne peut pas briser un invariant de composant. Ce choix doit être considéré soigneusement chaque fois qu'il faut étendre une abstraction.

On a pu penser que l'héritage résoudrait le problème du réemploi du logiciel. Il simplifierait la construction des bibliothèques que l'on pourrait distribuer au tiers, pour l'utilisation dans d'autres applications. Cet espoir a été partiellement réalisé avec les cadres d'applications. Un **cadre d'applications** (« *application framework* ») est une plate-forme logicielle qui est générique. Instancier le cadre veut dire donner des

valeurs effectives aux paramètres génériques. Cela peut être fait en ce qui concerne l'héritage avec les classes génériques ou les classes abstraites.

6.2 LES CLASSES COMME ABSTRACTIONS COMPLÈTES

Le cœur du concept d'objet est l'accès contrôlé aux données encapsulées. Le comportement d'un objet est spécifié par une classe. Dans le cas général, une classe est une définition incrémentale d'une abstraction de données, qui définit l'abstraction comme une modification d'autres abstractions. Il existe un grand nombre de concepts pour aider à définir des classes. Nous les classifions en deux groupes, selon qu'ils permettent la définition complète ou incrémentale d'une abstraction de données :

– *Abstraction de données complète*. Ce sont tous les concepts qui permettent à une classe de définir une abstraction de données par elle-même. Il y a deux sous-groupes :

  • Les membres de la classe (voir sections 6.2.3 et 6.2.4), à savoir les méthodes, attributs et propriétés. Il y a plusieurs manières d'initialiser les attributs, par objet ou par classe (voir section 6.2.5).
  • Les concepts qui exploitent le typage. Dans ce chapitre, nous nous limitons au typage dynamique. Il y a les messages et les attributs de première classe. Ces concepts permettent des formes puissantes de polymorphisme qui sont difficiles ou impossibles à faire dans les langages à typage statique. Cette liberté augmente la responsabilité du programmeur qui doit l'utiliser correctement.

– *Abstraction de données incrémentale*. Ce sont tous les concepts relatifs à l'héritage : ils définissent comment une classe utilise les autres. Ils sont expliqués dans la section 6.3.

Pour expliquer les classes, nous commençons par un exemple qui montre comment définir une classe et un objet. La section 6.2.1 montre en exemple la syntaxe de classe et ensuite la section 6.2.2 donne sa sémantique rigoureuse par une définition dans le modèle avec état.

6.2.1 Un exemple

Nous définissons un exemple de classe et nous l'utilisons pour créer un objet. Nous supposons que notre langage a une abstraction linguistique, **`class`**, pour définir les classes. Nous supposons que les classes sont des valeurs dans le langage. Cela nous permet d'utiliser une déclaration **`class`** comme une instruction ou une expression, comme une déclaration **`proc`**.

La figure 6.1 définit une classe référencée par la variable `Counter`. Cette classe a un attribut, `val`, qui contient la valeur actuelle d'un compteur, et trois méthodes, `init`,

```
class Counter
   attr val
   meth init(Value)
      val:=Value
   end
   meth browse
      {Browse @val}
   end
   meth inc(Value)
      val:=@val+Value
   end
end
```

Figure 6.1 Un exemple de classe `Counter` (avec la syntaxe de **`class`**).

`browse` et `inc`, pour l'initialisation, l'affichage et l'incrémentation du compteur. L'attribut est modifié avec l'opérateur := et lu avec l'opérateur `@`. À première vue, c'est très proche d'autres langages, avec une syntaxe légèrement différente. Mais il y a quelques différences importantes !

La déclaration de la figure 6.1 est une instruction exécutable : elle crée une valeur de classe et la lie à `Counter`. Si on remplace « `Counter` » par « `$` », la déclaration pourra être utilisée dans une expression. Mettre cette déclaration à la tête d'un programme déclarera la classe avant le reste, ce qui est un comportement familier. Mais ce n'est pas la seule possibilité. La déclaration peut se mettre partout où l'on peut mettre une instruction. Par exemple, mettre la déclaration à l'intérieur d'une procédure créera une nouvelle classe distincte chaque fois que la procédure sera appelée.

Nous pouvons créer un objet de la classe `Counter` et faire quelques opérations avec lui :

```
C={New Counter init(0)}
{C inc(6)} {C inc(6)}
{C browse}
```

Ces instructions créent l'objet compteur `C` avec une valeur initiale `0`, l'incrémentent deux fois par `6` et affichent sa valeur. L'instruction `{C inc(6)}` s'appelle une invocation d'objet. Le message `inc(6)` est utilisé par l'objet pour identifier la méthode correspondante. Essayez le code suivant :

```
local X in {C inc(X)} X=5 end
{C browse}
```

Rien n'est affiché ! C'est parce que l'invocation de l'objet

```
{C inc(X)}
```

bloque à l'intérieur de la méthode `inc`. Voyez-vous l'endroit exact du blocage ? Essayez maintenant la variation suivante :

```
declare S in
local X in thread {C inc(X)} S=unit end X=5 end
{Wait S} {C browse}
```

Les choses fonctionnent comme prévu. L'exécution dataflow garde son comportement habituel avec les objets.

6.2.2 La sémantique de l'exemple

Avant d'expliquer les autres possibilités des classes, nous donnons la sémantique de l'exemple `Counter`. Cette définition utilise la programmation d'ordre supérieur avec l'état explicite. La sémantique que nous donnons ici est légèrement simplifiée ; elle omet les possibilités de **`class`** qui ne sont pas utilisées dans l'exemple (comme l'héritage et **`self`**). La sémantique complète est donnée dans [97].

```
local
   proc {Init M S}
      init(Value)=M in (S.val):=Value
   end
   proc {Browse2 M S}
      {Browse @(S.val)}
   end
   proc {Inc M S}
      inc(Value)=M in (S.val):=@(S.val)+Value
   end
in
   Counter=c(attrs:[val]
             methods:m(init:Init browse:Browse2 inc:Inc))
end
```

Figure 6.2 La définition de la classe `Counter` (sans soutien syntaxique).

La figure 6.2 donne la même définition que la figure 6.1 sans utiliser la syntaxe **`class`**. Une classe est simplement un enregistrement qui contient un ensemble de noms d'attributs (représenté comme une liste) et un ensemble de méthodes (représenté comme un enregistrement). Le nom d'un attribut est un **littéral** (« *literal* »),

c'est-à-dire un atome ou un nom. Une méthode est une procédure avec deux arguments, le message et l'état de l'objet. Dans chaque méthode, l'affectation à un attribut (« `val:=` ») est faite par une affectation de cellule et l'accès à un attribut (« `@val` ») est fait par un accès de cellule.

La figure 6.3 définit la fonction `New` pour créer des objets à partir des classes. Cette fonction crée l'état `S` de objet, définit une procédure d'un argument `Obj` qui sera l'objet et enfin initialise l'objet avant de le renvoyer. L'état est un enregistrement qui a un champ pour chaque attribut et le champ contient une cellule. L'état est caché à l'intérieur de `Obj` par la portée lexicale.

```
fun {New Class Init}
   Fs={Map Class.attrs fun {$ X} X#{NewCell _} end}
   S={List.toRecord state Fs}
   proc {Obj M}
      {Class.methods.{Label M} M S}
   end
in
   {Obj Init}
   Obj
end
```

Figure 6.3 La création d'un objet `Counter`.

6.2.3 La définition des classes et objets

Une classe est une structure de données qui définit l'état interne d'un objet (les attributs), son comportement (les méthodes), les classes dont il hérite et plusieurs autres propriétés et opérations que nous verrons plus loin. De façon plus générale, on peut voir une classe comme une structure de données qui définit une abstraction de données et qui lui donne son implémentation partielle ou complète. Le tableau 6.1 définit la syntaxe des classes.

Une classe peut donner lieu à un nombre arbitraire d'instances, appelées des objets. Chaque objet a une identité différente et peut avoir des valeurs différentes pour son état interne. Sinon, tous les objets de la même classe se comportent selon la définition de la classe. Un objet est créé avec l'opération `New` :

```
MyObj={New MyClass Init}
```

Cet appel crée un nouvel objet `MyObj` de la classe `MyClass` et invoque l'objet avec le message `Init`. Ce message est utilisé pour initialiser l'objet. L'objet `MyObj` est appelé avec la syntaxe `{MyObj M}`. Il se comporte comme une procédure à un argument, où l'argument est le message. Les messages `Init` et `M` sont représentés

comme des enregistrements. Un appel d'objet ressemble à un appel de procédure. Il retourne quand la méthode a terminé son exécution.

```
⟨statement⟩   ::= class ⟨variable⟩ { ⟨classDesc⟩ }
                  { meth ⟨methHead⟩ [ ´=´ ⟨variable⟩ ]
                    ( ⟨inExpression⟩ | ⟨inStatement⟩ ) end }
                  end
                | lock [ ⟨expression⟩ then ] ⟨inStatement⟩ end
                | ⟨expression⟩ ':=' ⟨expression⟩
                | ⟨expression⟩ ´,´ ⟨expression⟩
                | ...
⟨expression⟩  ::= class ´$´ { ⟨classDesc⟩ }
                  { meth ⟨methHead⟩ [ ´=´ ⟨variable⟩ ]
                    ( ⟨inExpression⟩ | ⟨inStatement⟩ ) end }
                  end
                | lock [ ⟨expression⟩ then ] ⟨inExpression⟩ end
                | ⟨expression⟩ ':=' ⟨expression⟩
                | ⟨expression⟩ ´,´ ⟨expression⟩
                | ´@´ ⟨expression⟩
                | self
                | ...
⟨classDesc⟩   ::= from { ⟨expression⟩ }+ | prop { ⟨expression⟩ }+
                | attr { ⟨attrInit⟩ }+
⟨attrInit⟩    ::= ( [ ´!´ ] ⟨variable⟩ | ⟨atom⟩ | unit | true | false )
                  [ ':' ⟨expression⟩ ]
⟨methHead⟩    ::= ( [ ´!´ ] ⟨variable⟩ | ⟨atom⟩ | unit | true | false )
                  [ ´(´ { ⟨methArg⟩ } [ ´...´ ] ´)´ ]
                  [ ´=´ ⟨variable⟩ ]
⟨methArg⟩     ::= [ ⟨feature⟩ ':' ] ( ⟨variable⟩ | ´_´ | ´$´ )
                  [ ´<=´ ⟨expression⟩ ]
```

Tableau 6.1 La syntaxe des classes.

6.2.4 Les membres des classes

Une classe définit les parties constitutives de chacun de ses objets. Dans la terminologie orientée objet, ces parties s'appellent les **membres**. Il y a trois sortes de membres :

– *Les attributs* (déclarés avec le mot clé « **attr** »). Un attribut est une cellule qui contient une partie de l'état de l'instance. Dans la terminologie orientée objet, un attribut s'appelle souvent une **variable d'instance**. L'attribut peut contenir

toute entité du langage. L'attribut est visible seulement dans la définition de la classe et dans toutes les classes qui héritent de cette classe. Chaque instance a un ensemble distinct d'attributs. L'instance peut modifier l'attribut avec les opérations suivantes :

- Une instruction d'affectation : ⟨expr⟩$_1$:=⟨expr⟩$_2$. Le résultat de l'évaluation de ⟨expr⟩$_2$ est affecté à l'attribut dont le nom est obtenu par l'évaluation de ⟨expr⟩$_1$.
- Une opération d'accès : `@`⟨expr⟩. Le résultat est le contenu de l'attribut dont le nom est obtenu par l'évaluation de ⟨expr⟩. L'opération d'accès peut être utilisée dans toute expression qui est lexiquement à l'intérieur de la définition de la classe. En particulier, elle peut être utilisée à l'intérieur des procédures qui sont définies dans la classe.
- Une opération d'échange. Si l'affectation ⟨expr⟩$_1$:=⟨expr⟩$_2$ est utilisée comme une expression, elle aura l'effet d'un échange. Par exemple, considérez l'instruction ⟨expr⟩$_3$=⟨expr⟩$_1$:=⟨expr⟩$_2$. Les trois expressions sont évaluées d'abord. Ensuite, ⟨expr⟩$_3$ est liée avec le contenu de l'attribut ⟨expr⟩$_1$ et le nouveau contenu de l'attribut est ⟨expr⟩$_2$. Cette opération est atomique.

– *Les méthodes* (déclarées avec le mot clé « **`meth`** »). Une méthode est semblable à une procédure qui est appelée dans le contexte d'un objet et qui peut accéder aux attributs de l'objet. La méthode a un en-tête et un corps. L'en-tête a une étiquette, qui doit être un atome ou un nom, et un ensemble d'arguments. (On appelle parfois l'en-tête la méthode ou le nom de la méthode.) Les arguments doivent être des variables distinctes, sinon il y a une erreur de syntaxe. Pour augmenter l'expressivité, un en-tête peut être une forme et un message peut être un enregistrement.
– *Les propriétés* (déclarées avec le mot clé « **`prop`** »). Une propriété modifie le comportement de la classe ou de l'objet. Par exemple, avec la propriété `final`, la classe est une classe finale, qui ne peut pas être étendue avec l'héritage. C'est une bonne habitude de donner la propriété `final` à chaque classe sauf si elle est spécifiquement conçue pour l'héritage.

Les attributs et les étiquettes des méthodes sont des littéraux. S'ils sont définis avec une syntaxe d'atome, ils sont des atomes. S'ils sont définis avec une syntaxe d'identificateur (par exemple, la première lettre est une majuscule), alors le système créera de nouveaux noms pour eux. La portée des identificateurs est la définition de la classe. L'utilisation de noms donne un contrôle fin sur la sécurité de l'objet.

En plus de ces trois sortes de membres, la section 6.3 montre comment une classe peut hériter des membres d'autres classes.

6.2.5 L'initialisation des attributs

Il y a trois manières d'initialiser un attribut : par instance, par classe ou par marque.

– *Par instance*. Un attribut peut avoir une valeur initiale différente pour chaque instance. Pour faire cela, on l'initialise dans une méthode. Par exemple :

```
class OneApt
   attr streetName
   meth init(X) @streetName=X end
end
Apt1={New OneApt init(drottninggatan)}
Apt2={New OneApt init(rueNeuve)}
```

Chaque instance, y compris `Apt1` et `Apt2`, contient initialement une autre variable non liée. Chacune de ces variables peut donc être liée à une autre valeur.

– *Par classe*. Un attribut peut avoir une valeur initiale qui est la même pour toutes les instances de la classe. Pour faire cela on l'initialise avec « : » dans la définition de la classe. Par exemple :

```
class Apartment
   attr
      streetName:york
      streetNumber:100
      wallColor:_
      floorSurface:wood
   meth init skip end
end
Apt3={New Apartment init}
Apt4={New Apartment init}
```

Toutes les instances, y compris `Apt3` et `Apt4`, auront les mêmes valeurs initiales pour les quatre attributs. Attention, cela inclut aussi `wallColor`, même si la valeur initiale est une variable non liée. Toutes les instances feront référence à la même variable non liée. Elle peut être liée dans une des instances, par exemple en faisant `@wallColor=white`. Toutes les instances ont alors cette même valeur. Il ne faut pas confondre les deux opérations `@wallColor=white` et `wallColor:=white`. La première opération garde la même variable dans `wallColor`. La deuxième opération y met une nouvelle variable.

– *Par marque*. C'est une autre manière d'utiliser l'initialisation par classe. Une marque (« *brand* ») est un ensemble de classes qui sont apparentées mais pas par l'héritage. On peut donner à un attribut une valeur qui est la même pour tous les

membres d'une marque en les initialisant avec la même variable pour tous les membres. Par exemple[1] :

```
L=linux
class RedHat attr ostype:L end
class SuSE attr ostype:L end
class Debian attr ostype:L end
```

Chaque instance de chaque classe sera initialisée à la même valeur.

6.2.6 Les techniques de programmation

Le concept de classe que nous avons introduit jusque-là nous donne une syntaxe commode pour définir des abstractions de données avec l'état encapsulé et plusieurs opérations. L'instruction **`class`** définit une valeur de classe qui peut être instanciée pour obtenir des objets. En plus d'une syntaxe commode, les valeurs de classe définies ici gardent tous les avantages des valeurs procédurales. Les techniques de programmation pour les procédures sont aussi applicables aux classes. Les classes peuvent avoir des références externes tout comme les valeurs procédurales. Les classes sont compositionnelles : on peut imbriquer les classes dans les classes. Elles sont compatibles avec les valeurs procédurales : on peut imbriquer les classes dans les procédures et vice versa. Les classes ne sont pas aussi souples dans tous les langages orientés objet ; souvent il y a quelques limitations.

6.3 LES CLASSES COMME ABSTRACTIONS INCRÉMENTALES

Comme nous l'avons expliqué, l'héritage est le concept majeur que la POO ajoute à la programmation par composants. La POO permet la définition incrémentale d'une classe comme une extension des classes existantes. Il y a trois concepts importants pour utiliser l'héritage :

- Le premier est le graphe d'héritage (voir section 6.3.1), qui définit quelles classes sont étendues. Notre modèle permet l'héritage simple et multiple.
- Le deuxième est le contrôle d'accès aux méthodes (voir section 6.3.3), qui définit comment accéder aux méthodes dans les classes. Il utilise les liens statiques et dynamiques et le concept de **`self`**.
- Le troisième est le contrôle d'encapsulation (voir section 6.3.4), qui définit quelle partie du programme peut voir les attributs et les méthodes d'une classe. Nous définissons les portées les plus populaires et nous montrons comment les autres portées peuvent être programmées.

1. Avec nos excuses à toutes les distributions Linux omises.

6.3.1 Le graphe d'héritage

L'héritage est une manière de construire de nouvelles classes à partir des classes existantes. Il définit les attributs et les méthodes qui sont disponibles dans la nouvelle classe. Nous restreignons notre discussion de l'héritage aux méthodes. Les mêmes règles s'appliquent aux attributs. Les méthodes disponibles dans une classe `C` sont définies par une relation de précédence sur les méthodes qui apparaissent dans la hiérarchie de classes. Nous appelons cette relation la relation de redéfinition :

– Une méthode dans une classe `C` redéfinit toute les méthodes avec la même étiquette dans toutes les superclasses de `C`.

Une classe peut hériter d'une ou plusieurs classes, qui apparaissent après le mot clé **`from`** dans sa déclaration. Une classe qui hérite exactement d'une classe utilise l'héritage simple. Une classe qui hérite de plusieurs classes utilise l'héritage multiple. Une classe `B` est une superclasse d'une classe `A` si :

– `B` apparaît dans la déclaration **`from`** de `A`, ou
– `B` est une superclasse d'une classe qui apparaît dans la déclaration **`from`** de `A`.

Une hiérarchie de classe avec la relation de superclasse peut être vue comme un graphe orienté dont la classe définie est la racine. Les arêtes sont dirigées vers les sous-classes. Il y a deux conditions pour que l'héritage soit légal. D'abord, la relation d'héritage doit être orientée et acyclique. La hiérarchie suivante n'est pas admise :

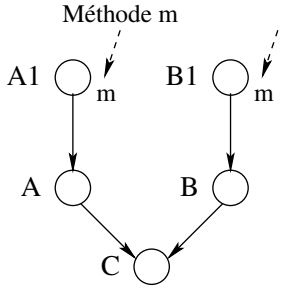
```
class A from B ... end
class B from A ... end
```

Ensuite, si on barre toutes les méthodes redéfinies, chaque méthode qui reste devra avoir une étiquette unique et être définie dans une seule classe de la hiérarchie. Donc, la classe `C` dans l'exemple suivant est illégale parce qu'il reste deux méthodes avec étiquette `m` :

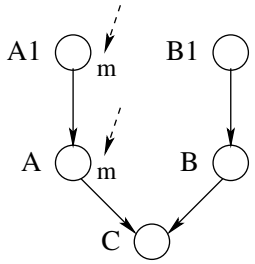
```
class A1 meth m(...) ... end end
class B1 meth m(...) ... end end
class A from A1 end
class B from B1 end
class C from A B end
```

La figure 6.4 montre cette hiérarchie et une autre légèrement différente qui est légale. La classe `C` ci-dessous est illégale aussi, parce qu'il y a deux méthodes `m` disponibles dans `C` :

```
class A meth m(...) ... end end
class B meth m(...) ... end end
class C from A B end
```



Hiérarchie de classe illégale
(deux «m» visibles à partir de C)



Hiérarchie de classe légale
(un «m» visible à partir de C)

Figure 6.4 Des hiérarchies de classe illégale et légale.

6.3.2 Tout est exécution

Si un programme qui déclare `C` est compilé, le système ne se plaindra pas. C'est seulement quand le programme exécute la déclaration que le système lève une exception. Si le programme ne l'exécute pas, aucune exception n'est levée. Par exemple, un programme qui contient le code suivant :

```
fun {StrangeClass}
   class A meth foo(X) X=a end end
   class B meth foo(X) X=b end end
   class C from A B end
in C end
```

peut être compilé et exécuté avec succès. Son exécution a pour effet de définir la fonction `StrangeClass`. C'est seulement pendant l'appel `{StrangeClass}` qu'une exception sera levée. Cette « détection tardive d'erreurs » n'est pas seulement une propriété des déclarations de classe. C'est une propriété générale du système Mozart qui est une conséquence de la nature dynamique du langage Oz : il n'y a aucune distinction entre une opération faite à la compilation et la même opération faite à l'exécution. Le système à objets partage cette nature dynamique. Par exemple, il est possible de définir des classes dont les étiquettes des méthodes sont calculées à l'exécution.

On peut dire que tout est exécution dans le système Mozart. Le compilateur fait partie du système à l'exécution. Une déclaration de classe est une instruction exécutable. Son exécution crée une classe, qui est une valeur dans le langage (voir figure 6.5). Cette valeur peut être passée à `New` pour créer un objet.

Un système de programmation n'a pas besoin de faire la distinction entre la compilation et l'exécution. Cette distinction est simplement une manière d'aider le compilateur

à accomplir certaines optimisations. Certains langages courants, comme C++ et Java, font cette distinction. D'autres, comme Ruby et Python, ne la font pas. Quelques opérations (comme les déclarations) ne peuvent être exécutées que pendant la compilation, et toutes les autres opérations ne peuvent être exécutées que pendant l'exécution. Le compilateur peut ainsi exécuter toutes les déclarations en une fois, sans aucune interférence du reste du programme. Cela lui permet de faire des optimisations plus poussées pour la génération du code machine. Mais cela réduit largement la souplesse du langage. Par exemple, la généricité et l'instanciation ne sont plus disponibles au programmeur comme des outils généraux.

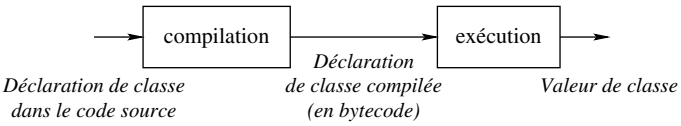


Figure 6.5 Une déclaration de classe est une instruction exécutable.

À cause de la nature dynamique du système Mozart, le compilateur a un tout petit rôle. Comme le compilateur n'exécute pas véritablement les déclarations (il les convertit simplement en code exécutable), il n'a pas besoin de connaître beaucoup de la sémantique du langage. Le compilateur Mozart a quelques connaissances mais seulement pour permettre une détection plus tôt de certaines erreurs et un code compilé plus efficace. On pourrait ajouter plus de connaissances au compilateur, par exemple pour détecter des erreurs dans les hiérarchies de classe.

6.3.3 Les liens statiques et dynamiques

Pendant l'exécution d'un objet, il faut souvent appeler une autre méthode dans le même objet, c'est-à-dire faire une sorte d'invocation récursive de l'objet. Cela semble simple, mais devient un peu plus compliqué quand l'héritage entre en jeu. L'héritage est utilisé pour définir une nouvelle classe qui étend une classe existante. Deux classes sont impliquées dans cette définition : la nouvelle classe et la classe existante. Les deux peuvent avoir des méthodes avec le même nom, et la nouvelle classe voudrait appeler l'une ou l'autre. Cela veut dire que nous avons besoin de deux manières pour faire un appel récursif. Elles s'appellent le lien statique et le lien dynamique. Nous les introduisons avec un exemple.

Un exemple

Regardez la classe `Account` définie dans la figure 6.6. Cette classe modélise un compte en banque simple avec un solde. Nous pouvons transférer de l'argent dans le compte avec `transfer`, obtenir le solde actuel avec `getBal` et faire une série

de transferts avec `batchTransfer`. Remarquez que `batchTransfer` appelle `transfer` pour chaque transfert.

```
class Account
   attr balance:0
   meth transfer(Amt)
      balance:=@balance+Amt
   end
   meth getBal(Bal)
      Bal=@balance
   end
   meth batchTransfer(AmtList)
      for A in AmtList do {self transfer(A)} end
   end
end
```

Figure 6.6 Un exemple de classe `Account`.

Nous étendons `Account` pour tenir un journal, c'est-à-dire pour garder une trace de toutes les transactions. Une solution est d'utiliser l'héritage pour redéfinir la méthode `transfer` :

```
class LoggedAccount from Account
   meth transfer(Amt)
      {LogObj addEntry(transfer(Amt))}
      ...
   end
end
```

où `LogObj` est un objet qui garde le journal. Nous créons un compte avec un journal qui a un solde initial de 100 :

```
LogAct={New LoggedAccount transfer(100)}
```

Que se passe-t-il quand nous appelons `batchTransfer` ? Appelle-t-elle l'ancienne `transfer` dans `Account` ou la nouvelle `transfer` dans `LoggedAccount` ? Nous pouvons déduire la réponse, si nous supposons qu'une classe définit une abstraction de données en style objet. L'abstraction de données a un ensemble de méthodes. Pour `LoggedAccount`, cet ensemble contient les méthodes `getBal` et `batchTransfer` définies dans `Account` et la nouvelle méthode `transfer` définie dans `LoggedAccount` elle-même. Par conséquent, la réponse est que `batchTransfer` doit appeler la nouvelle `transfer` dans `LoggedAccount`. Ce choix s'appelle un lien dynamique. On l'écrit comme un appel à **`self`**, c'est-à-dire `{self transfer(A)}`.

Quand `Account` a été définie, il n'y avait pas de classe `LoggedAccount`. Avec un lien dynamique, nous gardons la possibilité d'extension d'`Account` avec l'héritage, tout en s'assurant que la nouvelle classe est une abstraction de données qui fait une extension correcte de l'ancienne.

Le lien dynamique est nécessaire mais pas suffisant pour implémenter l'abstraction étendue. Pour comprendre pourquoi, regardons la définition de la nouvelle `transfer`. Voici la définition complète :

```
class LoggedAccount from Account
   meth transfer(Amt)
      {LogObj addEntry(transfer(Amt))}
      Account,transfer(Amt)
   end
end
```

À l'intérieur de la nouvelle `transfer`, il faut appeler l'ancienne `transfer`. Nous ne pouvons pas utiliser un lien dynamique, parce qu'il appelle toujours la nouvelle `transfer`. Nous utilisons une autre technique, le lien statique. Avec cette technique, nous appelons une méthode dans une classe spécifique de la hiérarchie. La notation `Account,transfer(Amt)` appelle la méthode `transfer` dans la classe `Account` et non dans `LoggedAccount`.

Discussion

Les liens statiques et dynamiques sont nécessaires tous les deux quand on utilise l'héritage pour redéfinir des méthodes. Le lien dynamique permet à la nouvelle classe d'étendre correctement l'ancienne en permettant aux anciennes méthodes d'en appeler des nouvelles, même si la nouvelle méthode n'existait pas quand l'ancienne a été définie. Le lien statique permet aux nouvelles méthodes d'appeler les anciennes. Nous résumons les deux concepts :

- **Le lien dynamique**. On l'écrit `{self M}`. Il choisit la méthode qui correspond à `M` qui est visible dans l'objet actuel. Cela prend en compte la redéfinition.
- **Le lien statique**. On l'écrit `C, M` (avec une virgule), où `C` est une classe qui définit une méthode qui correspond à `M`. Il choisit cette méthode. Cela prend en compte la redéfinition jusqu'à la classe `C` mais pas plus loin. Si l'objet est d'une sous-classe de `C` qui a redéfinie `M` de nouveau, cela ne sera pas pris en compte.

Pour les attributs, le lien dynamique est le seul comportement possible. Le lien statique n'est pas possible pour eux car les anciens attributs n'existent simplement pas, ni dans un sens logique (le seul objet qui existe est l'instance de la classe qui résulte après que tout l'héritage soit fait) ni dans un sens pratique (l'implémentation n'alloue aucune mémoire pour eux).

6.3.4 Le contrôle d'encapsulation

Le principe du contrôle d'encapsulation est de limiter l'accès aux membres d'une classe selon les besoins de l'architecture de l'application. Chaque membre est défini avec une portée. La **portée** est la partie du texte du programme dans laquelle le membre est visible, c'est-à-dire que l'on peut y accéder simplement en mentionnant son nom. Dans la plupart des cas, la portée est statique, définie par la structure du programme. Si on utilise des noms, elle pourra aussi être dynamique.

Un langage donne en général une portée par défaut à chaque membre quand il est déclaré. Ce défaut peut être modifié avec des mots clés spéciaux. Certains mots clés utilisés sont `public`, `private` et `protected`. Il se trouve que différents langages utilisent ces mots pour des portées légèrement différentes. Dans l'esprit de [25], nous essaierons de mettre de l'ordre dans ce chaos.

Les portées privées et publiques

Les deux portées les plus fondamentales sont la portée privée (« *private* ») et la portée publique (« *public* ») :

- Un membre privé est visible seulement dans l'objet lui-même. L'objet peut voir tous les membres privés définis dans sa classe et ses superclasses. Privée définit une sorte de visibilité verticale.
- Un membre public est visible partout dans le programme.

Dans les langages Smalltalk et Oz, les attributs sont privés et les méthodes sont publiques.

Ces définitions de privé et public sont naturelles si les classes sont utilisées pour construire des abstractions de données :

- D'abord, une classe n'est pas la même chose que l'abstraction qu'elle définit ! La classe est un incrément ; elle définit une abstraction de données comme une modification incrémentale des superclasses. La classe est nécessaire seulement pendant la construction de l'abstraction. L'abstraction de données n'est pas cependant un incrément ; elle contient tous ses attributs et méthodes. Ses attributs et méthodes peuvent venir de la classe ou d'une superclasse.
- Ensuite, les attributs sont internes à l'abstraction et doivent être invisibles de l'extérieur. C'est exactement la définition de portée privée.
- Enfin, les méthodes sont l'interface externe de l'abstraction, elles doivent donc être visibles pour toutes les entités qui référencent l'abstraction. C'est exactement la définition de portée publique.

La construction d'autres portées

Les techniques pour écrire des programmes qui contrôlent l'encapsulation sont basées essentiellement sur deux concepts : la portée lexicale et les valeurs de nom. Les portées privées et publiques définies auparavant, tout comme beaucoup d'autres portées, peuvent être implémentées avec ces deux concepts. Par exemple, il est possible d'exprimer les portées privée et protégée de C++ et Java, et d'écrire des programmes qui ont des politiques de sécurité bien plus élaborées. On utilise un nom au lieu d'un atome comme étiquette pour les méthodes. Un nom est une constante infalsifiable ; la seule manière de connaître un nom est que quelqu'un vous donne une référence (voir la documentation de Mozart [23]). Ainsi, un programme peut passer cette référence de façon contrôlée, aux parties du programme qui ont le droit de la connaître.

Dans les exemples précédents, nous avons utilisé des atomes comme étiquettes des méthodes. Mais les atomes ne sont pas sécurisés : si un tiers trouve la représentation imprimée de l'atome (en la devinant ou par un autre moyen), alors il pourra appeler la méthode. Les noms sont une manière simple de supprimer ce genre de fuite de sécurité. C'est important pour un projet de développement de logiciel avec des interfaces bien définies entre différents composants. C'est encore plus important pour les programmes répartis ouverts, qui contiennent du code écrit par différents groupes.

Les méthodes privées (dans le sens de C++ et Java)

Quand une étiquette de méthode est un nom, sa portée est limitée aux instances de la classe, mais pas aux sous-classes ou leurs instances. C'est exactement une portée privée dans le sens de C++ et Java. À cause de son utilité, le système à objets de ce chapitre donne un soutien syntaxique à cette technique. Il y a deux manières de l'écrire, avec un nom défini dans la classe ou à l'extérieur :

– Avec un identificateur comme étiquette de méthode. Cela crée implicitement un nom quand la classe est définie et lie le nom à l'identificateur. Par exemple :

```
class C
   meth A(X)
      % Corps de la méthode
   end
end
```

L'étiquette `A` est liée à un nom. L'identificateur `A` est visible seulement dans la définition de la classe. Une instance de `C` peut appeler la méthode `A` dans toute autre instance de `C`. La méthode `A` est invisible pour les sous-classes. C'est une sorte de visibilité horizontale. Elle correspond au concept de méthode privée comme il existe dans C++ et Java (mais pas dans Smalltalk). Comme la figure 6.7 le montre, « private » dans C++ et Java est très différente de « private » dans Smalltalk et Oz. Dans Smalltalk et Oz, « private » est relative à un objet et ses

superclasses, comme I3 dans la figure. Dans C++ et Java, « private » est relative à une classe et ses instances, comme SousSousC dans la figure.

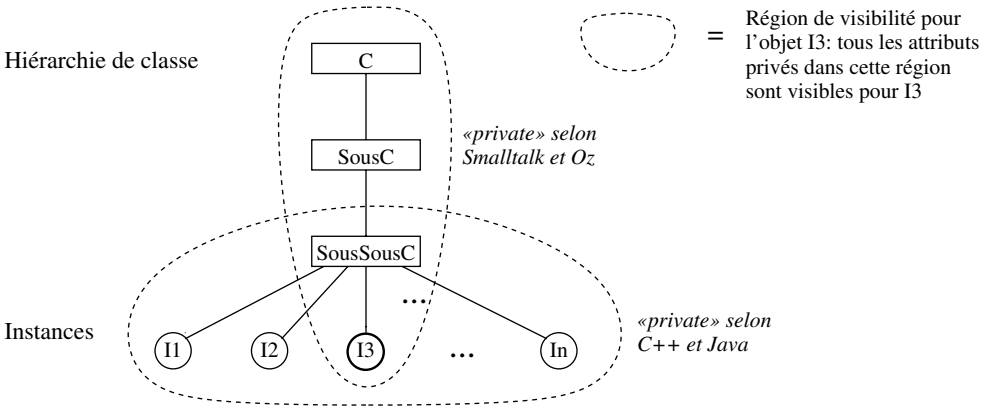


Figure 6.7 Le sens de « `private` ».

– Avec un identificateur « échappé » comme étiquette. Un identificateur échappé est marqué avec un point d'exclamation (comme `!A`). Cela indique que nous déclarons et lions la variable en dehors de la classe. Quand la classe est définie, la valeur de la variable devient l'étiquette de la méthode. C'est un mécanisme très général qui peut être utilisé pour protéger des méthodes de plusieurs façons. Il peut aussi être utilisé pour d'autres buts que la sécurité. Voici un exemple qui utilise ce mécanisme pour faire exactement la même chose que le cas précédent :

```
local A={NewName} in
   class C
      meth !A(X)
         % Corps de la méthode
      end
   end
end
```

Cet exemple crée un nom au moment de la définition de la classe, comme l'exemple précédent, et lie l'étiquette de la méthode avec le nom. En fait, la définition précédente est simplement un raccourci pour cet exemple.

Laisser le programmeur définir l'étiquette de la méthode permet la définition d'une politique de sécurité à une granularité très fine. Le programme peut donner l'étiquette aux entités qui doivent la connaître et pas aux autres.

Les méthodes protégées (dans le sens de C++)

Par défaut, les méthodes dans le système à objets sont publiques. Avec les noms, nous pouvons construire le concept de méthode protégée, y compris la version C++ et la version Java. En C++, une méthode sera protégée si elle est accessible seulement dans la classe qui la définit et dans les sous-classes (et tous les objets instances de ces classes). Ce concept de protection est une combinaison du concept de privé dans Smalltalk avec le concept de privé dans C++/Java : il y a des composantes horizontale et verticale. Nous montrons comment exprimer le concept de protection de C++. Le concept de protection en Java est un peu différent ; nous le laissons pour un exercice. Dans la classe suivante, la méthode `A` est protégée :

```
class C
   attr pa:A
   meth A(X) skip end
   meth foo(...) {self A(5)} end
end
```

Elle est protégée parce que l'attribut `pa` garde une référence vers `A`. Maintenant créez une sous-classe `C1` de `C`. Nous pouvons accéder à la méthode `A` dans la sous-classe avec le code suivant :

```
class C1 from C
   meth b(...) A=@pa in {self A(5)} end
end
```

La méthode `b` accède à la méthode `A` en utilisant l'attribut `pa`, qui existe dans la sous-classe. L'étiquette `A` est stockée dans l'attribut `pa`.

Les portées des attributs

Les attributs sont toujours privés. La seule manière de les rendre publics consiste à utiliser des méthodes. À cause du typage dynamique, il est possible de définir des méthodes génériques qui permettent de lire et d'écrire tous les attributs. Les attributs qui sont des atomes ne sont pas sécurisés parce qu'ils peuvent être devinés. Les attributs qui sont des noms sont sécurisés.

Les atomes ou les noms comme étiquettes des méthodes ?

Quand faut-il utiliser un atome ou un nom comme étiquette d'une méthode ? Par défaut, les atomes sont visibles partout dans le programme et les noms ne sont visibles que dans la portée lexicale de leur création. Il y a donc une règle simple pour les classes : pour les méthodes internes utilisez des noms et pour les méthodes externes utilisez des atomes.

La plupart des langages orientés objet populaires (comme Smalltalk, C++ et Java) ne soutiennent que des atomes comme étiquettes des méthodes, pas des noms. Pour restreindre la visibilité des atomes, ces langages utilisent des déclarations (comme `private` et `protected`). C'est une approche syntaxique : pour comprendre ces déclarations il faut connaître leur sémantique. Par contre, l'utilisation des noms est une approche sémantique : on peut les comprendre tout de suite parce qu'ils font partie du langage noyau. L'utilisation des noms est pratique parce que les visibilités que l'on peut obtenir ne sont pas limitées aux déclarations imaginées par les concepteurs du langage. Le programmeur peut définir d'autres visibilités. Mais pour l'instant, l'approche capacité exemplifiée par les noms n'est pas encore populaire dans les langages courants.

Regardons de plus près les compromis entre les noms et les atomes. Les atomes sont identifiés par leurs représentations imprimées. Ils peuvent donc être stockés dans les fichiers texte, dans les courriels, dans les pages Web, etc. En particulier, ils peuvent être stockés dans la tête du programmeur ! Quand on écrit un grand programme, une méthode peut être appelée de partout simplement en donnant sa représentation imprimée. Par contre, la dissémination des noms est très différente : le programme lui-même doit passer le nom au code qui appelle la méthode. Cela ajoute de la complexité au programme et c'est un fardeau pour le programmeur. Les atomes l'emportent alors quant à la simplicité du programme et au confort psychologique pendant le développement.

Par contre, avec les noms il n'y a pas de conflits à propos de l'héritage (simple ou multiple). Ils ont peu de problèmes de sécurité : il est impossible d'utiliser un nom sans l'autorisation du programme. Quelqu'un qui a une référence à un objet n'a pas forcément le droit d'appeler toutes les méthodes de l'objet. Le programme sera donc moins enclin aux erreurs et mieux structuré. Un dernier point est que l'on peut donner un soutien syntaxique pour simplifier l'utilisation des noms. Par exemple, dans le système à objets de ce chapitre, il suffit que l'étiquette de la méthode commence avec une majuscule.

6.4 LA PROGRAMMATION AVEC L'HÉRITAGE

Toutes les techniques de programmation des chapitres précédents sont toujours valables dans le système à objets. Les techniques particulièrement utiles sont basées sur l'encapsulation et l'état pour modulariser les programmes.

Cette section se concentre sur les nouvelles techniques qui deviennent possibles par la POO. Ces techniques sont basées sur l'utilisation de l'héritage : d'abord l'utiliser correctement et ensuite exploiter sa puissance.

6.4.1 L'utilisation correcte de l'héritage

Il y a deux manières de regarder l'héritage :

- **La vue de type**. Dans cette vue, les classes sont des types et les sous-classes sont des sous-types. Par exemple, prenez une classe `LabeledWindow` qui hérite d'une classe `Window`. Toutes les fenêtres étiquetées sont aussi des fenêtres. La vue de type est cohérente avec le principe selon lequel les classes doivent modéliser des entités qui existent en dehors du programme. Dans la vue de type, les classes satisfont la **propriété de substitution** : chaque opération qui est valable pour un objet de la classe `C` l'est aussi pour les objets d'une sous-classe de `C`. La plupart des langages orientés objet, comme Java et Smalltalk, sont conçus pour la vue de type [28, 29].
- **La vue de structure**. Dans cette vue, l'héritage est simplement un outil pour structurer les programmes. Cette vue est *fortement découragée* parce que les classes ne satisfont plus la propriété de substitution. La vue de structure est une source inépuisable de bugs et de mauvaises conceptions. Des projets commerciaux majeurs qui resteront anonymes ont échoué pour cette raison.

Certains langages orientés objet, notamment Eiffel, sont conçus pour permettre les deux vues [65]. Dans la vue de type, chaque classe définit une vraie abstraction de données. Dans la vue de structure, les classes ne sont parfois que de l'échafaudage, qui n'existe que pour son rôle de structuration.

Un exemple

Dans la grande majorité des cas, l'héritage doit respecter la vue de type. Faire autrement engendre des bugs subtiles et pernicieux qui peuvent empoisonner tout le système. Par exemple, prenez la classe `Account` que nous avons vue auparavant, définie dans la figure 6.6. Un objet `A` de cette classe satisfait la règle algébrique suivante :

```
{A getBalance(b)} {A transfer(s)} {A getBalance(b')}
```

avec $b + s = b'$. En mots, avec un solde initial de b et un transfert de s, le solde final sera $b + s$. Cette règle algébrique peut être vue comme une spécification de `Account`, ou comme un contrat entre `Account` et le programme qui utilise ses objets. (Dans une définition pratique de `Account`, il y aurait d'autres règles. Nous les omettons pour simplifier l'exemple.) Selon la vue de type, les sous-classes de `Account` doivent aussi implémenter ce contrat. Nous étendons maintenant `Account` deux fois. La première extension est conservatrice, c'est-à-dire qu'elle respecte la vue de type :

```
class VerboseAccount from Account
   meth verboseTransfer(Amt)
      {self transfer(Amt)}
      {Browse ´Balance :´#@balance}
   end
end
```

Nous ajoutons simplement une nouvelle méthode, `verboseTransfer`. Comme les méthodes existantes ne sont pas changées, le contrat sera toujours respecté. Un objet de `VerboseAccount` sera valable dans tous les cas où un objet de `Account` l'est. Voici une deuxième extension plus dangereuse :

```
class AccountWithFee from VerboseAccount
   attr fee:5
   meth transfer(Amt)
      VerboseAccount,transfer(Amt-@fee)
   end
end
```

La figure 6.8 montre la nouvelle hiérarchie. La flèche ouverte dans cette figure est la notation habituelle d'un lien d'héritage. `AccountWithFee` redéfinit la méthode `transfer`. La redéfinition n'est pas un problème en soi. Le problème est qu'un objet `AccountWithFee` ne fonctionne pas correctement en tant qu'objet `Account`.

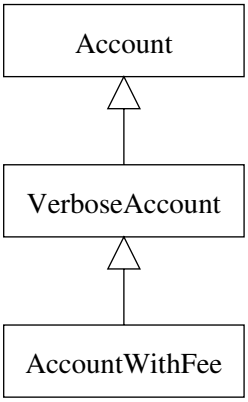


Figure 6.8 Une hiérarchie simple avec trois classes.

Considérez la séquence des trois appels suivants :

```
{A getBalance(B)} {A transfer(S)} {A getBalance(B2)}
```

Si `A` est un objet `AccountWithFee`, cela implique `B+S-@fee=B2`. Si `@fee`$\neq 0$ le contrat n'est plus satisfait. Cela cassera tout programme qui dépend du comportement des objets `Account`. Quand cette erreur arrive dans un grand programme, son origine n'est généralement pas évidente parce qu'elle est cachée à l'intérieur d'une méthode. Elle apparaîtra comme un léger déséquilibre dans les comptes, longtemps après que l'on ait oublié la classe `AccountWithFee`. Le débogage de ce genre de « petits » problèmes est extraordinairement difficile et souvent interminable.

À partir de maintenant nous allons nous concentrer sur la vue de type. Quoi qu'il en soit, la vue de structure est parfois utile. Son utilisation principale est de changer le comportement du système à objets lui-même. Dans ce but, elle doit être utilisée uniquement par des experts qui comprennent clairement les ramifications de ce qu'ils font. Pour plus d'informations, nous recommandons [65] pour une comparaison plus approfondie de la vue de type et de la vue de structure.

La conception par contrat

Nous disons qu'un programme est **correct** ou **exact** s'il s'exécute selon sa spécification. Une manière de prouver l'exactitude d'un programme est un raisonnement basé sur une sémantique formelle. Par exemple, avec un programme à état nous pouvons raisonner avec la sémantique axiomatique (voir un des nombreux livres sur la sémantique axiomatique). Nous pouvons aussi raisonner avec des règles algébriques, comme dans l'exemple `Account`. En se basant sur ces deux techniques, Bertrand Meyer a développé une méthode pour la conception de programmes corrects qui s'appelle la **conception par contrat** (« *design by contract* ») et il l'a implémentée dans le langage Eiffel [65].

L'idée principale de la conception par contrat est qu'une abstraction de données implique un contrat entre le concepteur de l'abstraction et ses utilisateurs. Les utilisateurs doivent garantir que l'abstraction est appelée correctement ; en échange l'abstraction se comportera correctement. Il y a une analogie avec les contrats dans la société humaine. Le contrat peut être formulé avec des règles algébriques, comme nous l'avons fait dans l'exemple `Account`, ou avec les préconditions et les postconditions. L'utilisateur s'assure que les préconditions sont toujours vraies avant d'appeler une opération. L'implémentation de l'abstraction de données s'assure alors que les postconditions seront toujours vraies quand l'opération se terminera. Il y a une répartition des responsabilités : l'utilisateur est responsable pour les préconditions et l'abstraction de données est responsable pour les postconditions.

L'abstraction de données vérifie que l'utilisateur respecte le contrat. C'est particulièrement facile si on utilise les préconditions et les postconditions. Les préconditions sont vérifiées à la frontière entre l'utilisateur et l'abstraction. Une fois à l'intérieur de l'abstraction, nous pouvons supposer que les préconditions sont satisfaites. Aucun test n'est fait à l'intérieur de l'abstraction, ce qui simplifie son implémentation. C'est

analogue à la société humaine. Par exemple, pour entrer dans un pays, on est contrôlé à la frontière (contrôle des passeports, sécurité, douane). Une fois à l'intérieur, il n'y a presque plus de vérifications.

Une histoire vraie

Nous terminons la discussion sur la bonne utilisation de l'héritage avec un conte basé sur une histoire vraie. Il y a quelque temps, une entreprise connue a initié un projet ambitieux basé sur la POO. Malgré un budget de plusieurs milliards de dollars, le projet a échoué. Parmi les nombreuses raisons de cet échec, il y avait une mauvaise utilisation de la POO, en particulier en ce qui concernait l'héritage. Deux erreurs majeures avaient été commises :

– La propriété de substitution a été régulièrement enfreinte. Des routines qui marchaient bien avec les objets d'une classe ne marchaient plus avec les objets d'une sous-classe. Cela compliquait le programme : il fallait plusieurs routines là où une seule aurait pu suffire.
– Les classes ont été étendues par l'héritage pour corriger de petits problèmes. Au lieu de corriger la classe elle-même, une sous-classe est définie comme correctif (« *patch* »). Il y avait des correctifs pour les correctifs et ainsi de suite. Le résultat était une grande profondeur inutile de la hiérarchie qui compliquait le système et ralentissait les invocations des objets.

Il faut donc faire attention à l'usage correct de l'héritage. Respectez la propriété de substitution. Utilisez l'héritage pour ajouter de la nouvelle fonctionnalité et non pour corriger une classe erronée. Étudiez des bons motifs de conception pour l'héritage.

6.4.2 L'héritage multiple pour une bibliothèque graphique

L'héritage multiple est utile quand un objet doit être deux choses différentes dans un même programme. Prenons une bibliothèque graphique qui peut afficher des figures géométriques diverses, comme des cercles, des lignes et des figures plus complexes. Nous voudrions définir une opération de « regroupement » qui peut combiner plusieurs figures pour faire une seule figure composite. Comment pouvons-nous modéliser cela avec la POO ? Nous concevrons un programme simple et complet qui fonctionne. Nous utiliserons l'héritage multiple pour ajouter la possibilité de regrouper les figures. L'idée pour cette conception vient de Bertrand Meyer [65]. Ce programme simple peut facilement être étendu pour devenir une vraie bibliothèque graphique performante.

Les figures géométriques

Nous définissons d'abord la classe `Figure` pour modéliser les figures géométriques, avec les méthodes `init` (initialiser la figure), `move(X Y)` (déplacer la figure) et `display` (afficher la figure) :

```
class Figure
   meth otherwise(M)
      raise undefinedMethod({Label M}) end
   end
end
```

`Figure` est une classe abstraite ; toute tentative d'invoquer ces méthodes lèvera une exception. La méthode `otherwise` a une sémantique particulière : elle est invoquée avec le message `M` quand aucune autre méthode n'est trouvée pour `M`. Une classe abstraite est une classe dans laquelle certaines méthodes ne sont pas définies. Pour utiliser une classe abstraite, on définit une autre classe qui hérite de la classe abstraite et qui ajoute les méthodes manquantes. Cela donne une classe concrète, c'est-à-dire une classe qui définit toutes ses méthodes et qui peut donc être instanciée. Les figures sont des instances des sous-classes de `Figure`. Par exemple, voici une classe `Line` (ligne) :

```
class Line from Figure
   attr canvas x1 y1 x2 y2
   meth init(Can X1 Y1 X2 Y2)
      canvas:=Can
      x1:=X1 y1:=Y1
      x2:=X2 y2:=Y2
   end
   meth move(X Y)
      x1:=@x1+X y1:=@y1+Y
      x2:=@x2+X y2:=@y2+Y
   end
   meth display
      {@canvas create(line @x1 @y1 @x2 @y2)}
   end
end
```

Voici une classe `Circle` (cercle) :

```
class Circle from Figure
   attr canvas x y r
   meth init(Can X Y R)
      canvas:=Can
      x:=X y:=Y r:=R
   end
   meth move(X Y)
      x:=@x+X y:=@y+Y
   end
   meth display
      {@canvas create(oval @x-@r @y-@r @x+@r @y+@r)}
   end
end
```

La figure 6.9 montre comment `Line` et `Circle` héritent de `Figure`. Ce genre de diagramme s'appelle un **diagramme de classe**. Ce diagramme fait partie de l'UML (« *Unified Modeling Language* »), un ensemble standardisé de techniques pour modéliser les programmes orientés objet [25]. Les diagrammes de classe sont utiles pour visualiser la structure de classe d'un programme orienté objet. Chaque classe est représentée par un rectangle avec trois parties, qui contiennent le nom de la classe, ses attributs et ses méthodes. Ces rectangles sont connectés par des lignes qui représentent les liens d'héritage.

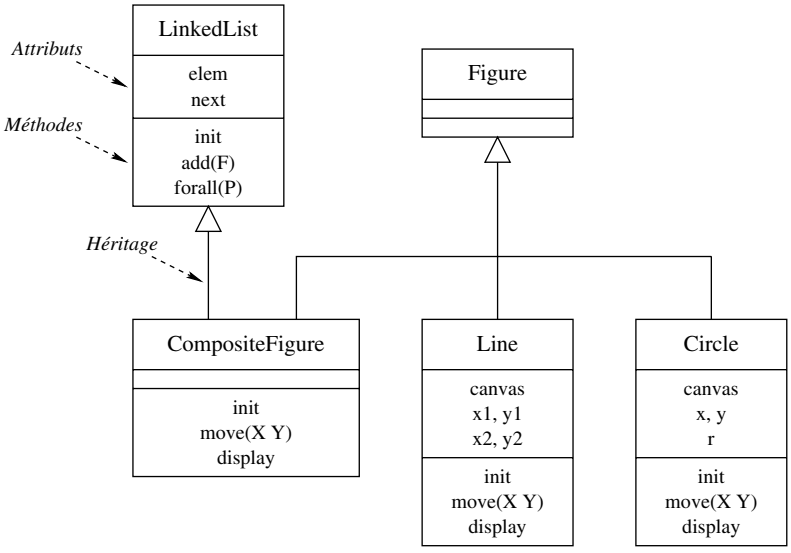


Figure 6.9 Le diagramme de classe de la bibliothèque graphique.

Les listes enchaînées

Nous définissons la classe `LinkedList` pour regrouper les figures ensemble, avec les méthodes `init` (initialiser la liste enchaînée), `add(F)` (ajouter la figure `F`) et `forall(M)` (exécuter `{F M}` pour toutes les figures `F`) :

```
class LinkedList
   attr elem next
   meth init(elem:E<=null next:N<=null)
      elem:=E next:=N
   end
   meth add(E)
      next:={New LinkedList init(elem:E next:@next)}
   end
   meth forall(M)
      if @elem\=null then {@elem M} end
      if @next\=null then {@next forall(M)} end
   end
end
```

La méthode `forall(M)` est particulièrement intéressante parce qu'elle utilise les messages de première classe. Une liste enchaînée est représentée comme une séquence d'instances de `LinkedList`. Le champ `next` de chaque instance référence la suivante dans la liste. Le dernier élément a le champ `next` égal à `null`. Il y a toujours au moins un élément dans la liste, qui s'appelle l'en-tête. L'en-tête n'est pas un élément qui est vu par les utilisateurs de la liste ; il est nécessaire uniquement pour l'implémentation. L'en-tête a toujours le champ `elem` égal à `null`. Une liste enchaînée qui est vide correspond donc à un en-tête avec les deux champs `elem` et `next` égal à `null`.

Les figures composites

Une figure composite est en même temps une figure et une liste enchaînée de figures. Nous définissons donc une classe `CompositeFigure` qui hérite des deux classes `Figure` et `LinkedList` :

```
class CompositeFigure from Figure LinkedList
   meth init
      LinkedList,init
   end
   meth move(X Y)
      {self forall(move(X Y))}
   end
   meth display
      {self forall(display)}
   end
end
```

La figure 6.9 montre l'héritage multiple de cet exemple. L'héritage multiple est correct dans ce cas parce que les deux fonctionnalités sont complètement différentes sans aucune interaction indésirable. La méthode `init` initialise la liste enchaînée. La figure n'a pas besoin d'initialisation. Comme dans toutes les figures, il y a une méthode `move` et une méthode `display`. La méthode `move(X Y)` déplace toutes les figures de la liste enchaînée. La méthode `display` affiche toutes les figures de la liste enchaînée.

Voyez-vous la beauté de cette conception ? Avec cette conception, une figure peut être faite d'autres figures, dont certaines sont faites d'autres figures, et ainsi de suite pour tout niveau de profondeur. La structure de l'héritage garantit que le déplacement et l'affichage fonctionneront toujours correctement. C'est un bel exemple de polymorphisme : les classes `CompositeFigure`, `Line` et `Circle` comprennent tous les messages `move(X Y)` et `display`.

Un exemple d'exécution

Nous pouvons exécuter cet exemple. D'abord, nous créons une fenêtre avec un champ d'affichage graphique :

```
declare
W=250 H=150 Can
Wind={QTk.build
      td(title:"Simple graphics package"
         canvas(width:W height:H bg:white handle:Can))}
{Wind show}
```

Cela utilise l'outil `QTk`, qui est expliqué dans la section 3.7.2. Pour cet exemple, nous introduisons un gadget de plus, un canevas (« *canvas* ») qui contient la surface d'affichage des figures géométriques. Ensuite, nous définissons une figure composite `F1` qui contient un triangle et un cercle :

```
declare
F1={New CompositeFigure init}
{F1 add({New Line init(Can 50 50 150 50)})}
{F1 add({New Line init(Can 150 50 100 125)})}
{F1 add({New Line init(Can 100 125 50 50)})}
{F1 add({New Circle init(Can 100 75 20)})}
```

Nous pouvons l'afficher avec `{F1 display}`. Nous pouvons faire une série de déplacements et d'affichages :

```
for I in 1..10 do {F1 display} {F1 move(3 ~2)} end
```

La figure 6.10 montre le résultat.

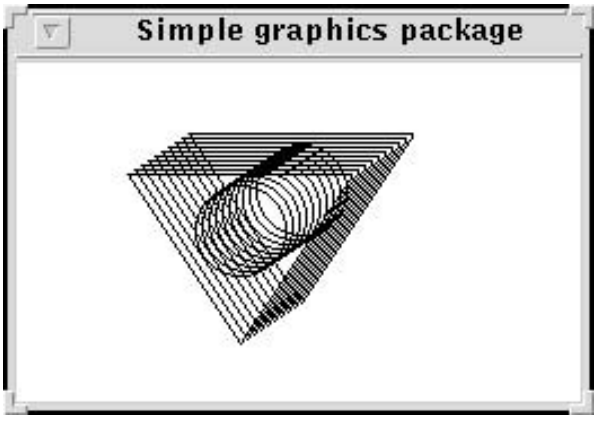


Figure 6.10 L'affichage dans la bibliothèque graphique.

Les figures composites avec l'héritage simple

Au lieu de définir `CompositeFigure` avec l'héritage multiple, nous pouvons la définir avec l'héritage simple en mettant la liste des figures dans un attribut. Cela donne

```
class CompositeFigure from Figure
   attr figlist
   meth init
      figlist:={New LinkedList init}
   end
   meth add(F)
      {@figlist add(F)}
   end
   meth move(X Y)
      {@figlist forall(move(X Y))}
   end
   meth display
      {@figlist forall(display)}
   end
end
```

La figure 6.11 montre le diagramme de classe pour cette version. Le lien entre `CompositeFigure` et `LinkedList` s'appelle une **association**. Il représente une relation entre les deux classes. Les nombres attachés aux deux bouts sont des cardinalités ; chaque nombre dit combien d'éléments il y a dans chaque association. Le nombre 1 sur le côté de la liste enchaînée signifie qu'il y a exactement une liste enchaînée par figure composite, et de même pour l'autre côté. Le lien d'association est une spécification ; il ne dit pas comment il est implémenté. Dans notre cas, chaque figure composite a un attribut `figlist` qui référence une liste enchaînée.

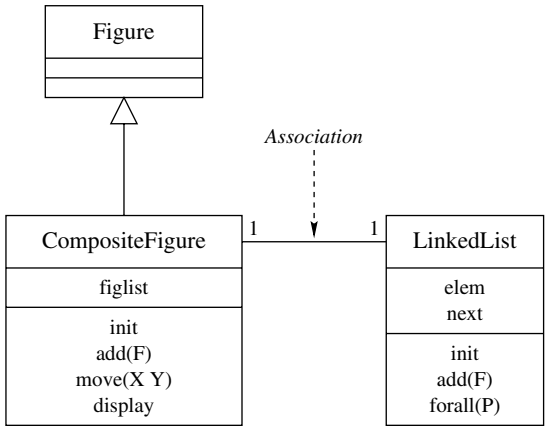


Figure 6.11 Un diagramme de classe avec une association.

L'exemple d'exécution que nous avons donné auparavant sera valable aussi dans le cas de l'héritage simple. Quelles sont les différences entre l'utilisation de l'héritage

simple et multiple pour cet exemple ? Dans les deux cas, les figures qui font la figure composite sont encapsulées. La différence principale est que l'héritage multiple place les opérations de la liste enchaînée au même niveau que la figure :

– Avec l'héritage multiple, une figure composite est aussi une liste enchaînée. Toutes les opérations de la classe `LinkedList` peuvent être utilisées directement sur les figures composites. C'est important si nous voulons faire des calculs de liste enchaînée sur les figures composites.
– Avec l'héritage simple, une figure composite cache complètement sa structure. C'est important si nous voulons protéger la figure composite de tous les calculs sauf ceux définis dans sa classe.

6.4.3 Quelques règles de conception pour l'héritage multiple

L'héritage multiple est une technique puissante à utiliser avec prudence. Nous vous recommandons de l'utiliser selon les règles suivantes :

– L'héritage multiple fonctionne bien pour combiner deux abstractions qui sont complètement indépendantes. Par exemple, les figures et les listes enchaînées ne partagent rien, elles peuvent donc être combinées avec succès.
– L'héritage multiple est bien plus difficile à utiliser correctement quand les abstractions partagent quelque chose. Par exemple, la création d'une classe `WorkStudy` à partir de `Student` et `Employee` est douteuse, parce que les étudiants et les employés sont tous les deux des êtres humains. Ils peuvent tous les deux hériter d'un ancêtre commun, une classe `Person` ! Même s'ils n'ont pas d'ancêtre commun, il peut y avoir des problèmes s'ils partagent des concepts.
– Que se passe-t-il quand deux superclasses frères partagent (directement ou indirectement) une classe ancêtre en commun qui spécifie un objet avec état (avec des attributs) ? Cette situation s'appelle le **problème d'implémentation partagée**. Elle peut mener à des opérations dupliquées sur l'ancêtre commun, par exemple quand on initialise un objet. La méthode d'initialisation doit normalement initialiser ses superclasses, l'ancêtre commun est donc initialisé deux fois. Le seul remède est de bien comprendre la hiérarchie pour éviter une telle duplication. Une alternative est de n'hériter que des classes qui ne partagent pas un ancêtre commun avec état.
– Quand il y a un conflit de nom entre deux méthodes, c'est-à-dire que la même étiquette est utilisée dans des superclasses adjacentes, alors le programme doit définir une méthode locale qui redéfinit les méthodes conflictuelles. Sinon le système à objets donne un message d'erreur. Une manière simple d'éviter ces conflits est d'utiliser des valeurs de nom comme étiquettes de méthodes. C'est une bonne technique pour certaines classes, comme les classes mixin, qui sont souvent utilisées dans l'héritage multiple.

6.4.4 L'utilisation des diagrammes de classe

Le diagramme de classe est un excellent outil pour visualiser la structure de classe d'une application. Il est au cœur de l'approche populaire UML pour la modélisation des applications orientées objet. Cette popularité a souvent masquée ses limites. Il y a trois limites évidentes :

– Il ne spécifie pas la fonctionnalité des classes. Par exemple, si les méthodes d'une classe maintiennent un invariant, alors cet invariant ne sera pas visible dans le diagramme de classe.
– Il ne modélise pas le comportement dynamique. Le comportement dynamique existe à grande échelle et à petite échelle. Les applications passent souvent par plusieurs phases d'exécution, avec un autre diagramme de classe valable pour chaque phase. Les applications sont souvent concurrentes, avec des parties indépendantes qui interagissent de manière coordonnée.
– Il ne modélise qu'un niveau dans la hiérarchie des composants. Comme la section 5.6 l'explique, les applications bien structurées ont une décomposition hiérarchique. Les classes et les objets sont proches de la base de cette hiérarchie. Un diagramme de classe montre la décomposition à ce niveau.

L'approche UML reconnaît ces limites et fournit des outils pour les alléger partiellement, comme le diagramme d'interaction (« *interaction diagram* ») et le diagramme de paquet (« *package diagram* »). Les diagrammes d'interaction modélisent une partie du comportement dynamique. Les diagrammes de paquet modélisent les composants à un niveau d'abstraction plus haut.

6.4.5 Les motifs de conception (« *design patterns* »)

Pendant la conception d'un système logiciel, il arrive souvent de rencontrer plusieurs fois les mêmes problèmes. L'approche des motifs de conception reconnaît ce fait et propose des solutions générales à ces problèmes. Un motif de conception est une technique qui résout un problème commun et qui peut être réutilisée facilement. Ce livre contient beaucoup de motifs de conception dans ce sens :

– Dans la programmation déclarative, la section 3.4.2 introduit la technique de construction d'une fonction en suivant la structure d'un type. Un programme qui utilise une structure de données récursive peut être écrit facilement en regardant le type de cette structure. La structure du programme reflète la définition du type.
– La section 5.4.2 introduit une série de techniques pour sécuriser une abstraction de données en l'emballant dans une couche sécurisée. Ces techniques sont indépendantes de la fonctionnalité de l'abstraction ; elles sont valables pour toute abstraction.

Les motifs de conception ont été popularisés dans un livre important par Gamma, Helm, Johnson et Vlissides [27], qui contient un catalogue des motifs de conceptions pour la POO et qui explique comment les utiliser. Le catalogue contient beaucoup de motifs basés sur l'héritage avec la vue de type. Pour fixer les idées, nous examinons un motif de conception typique de ce catalogue du point de vue d'un programmeur habitué à penser avec les modèles de calcul.

Le motif Composite

Composite est un exemple typique d'un motif de conception. Le but de Composite est de construire des hiérarchies d'objets. Si on donne une classe qui définit une feuille, le motif nous montrera comment utiliser l'héritage pour définir des arbres. La figure 6.12, inspirée par Gamma *et al.* [27], représente le diagramme d'héritage du motif Composite. On utilise ce diagramme en insérant une classe pour les feuilles, `Leaf`. Le motif définit ensuite les deux classes `Composite` et `Component`. `Component` est une classe abstraite.

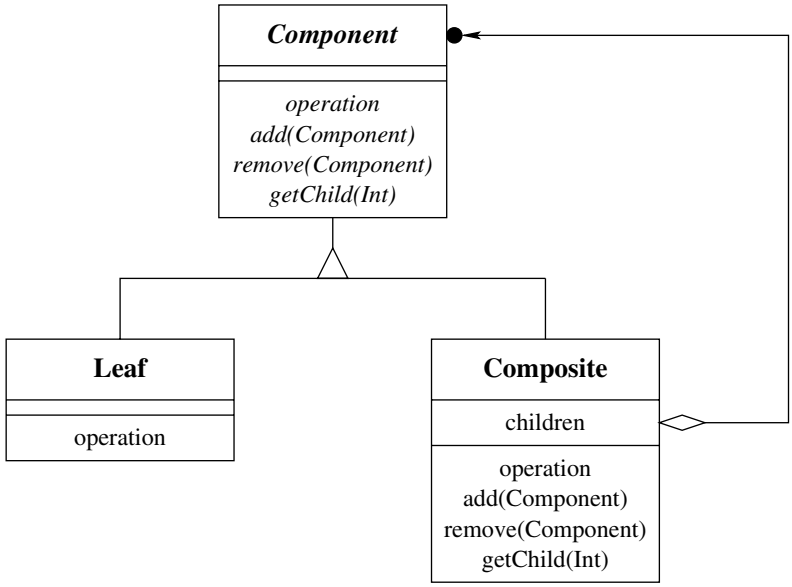


Figure 6.12 Le motif Composite.

Nous pouvons utiliser le motif Composite pour définir des figures graphiques composées. La section 6.4.2 résout le problème avec la combinaison d'une figure et d'une liste enchaînée (avec l'héritage simple ou multiple). Le motif Composite est une solution plus abstraite, parce qu'elle ne suppose pas que le regroupement soit fait par une liste enchaînée. La classe `Composite` a les opérations `add` et `remove` mais ne dit

pas comment elles sont implémentées. Elles peuvent être implémentées comme une liste enchaînée, mais elles peuvent avoir une autre implémentation, par exemple un dictionnaire ou une liste déclarative.

En donnant une classe qui définit une feuille de l'arbre, le motif Composite renvoie une classe qui définit l'arbre. Dit de cette façon, cela ressemble à la programmation d'ordre supérieur : nous voudrions définir une fonction qui accepte une classe et qui renvoie une autre classe. La plupart des langages courants, comme C++ et Java, ne permettent pas de définir cette fonction. Il y a deux raisons pour cela. D'abord, ces langages ne considèrent pas les classes comme des valeurs. Ensuite, la fonction définit une nouvelle superclasse de la classe d'entrée. Ces langages permettent la définition de nouvelles sous-classes mais pas de nouvelles superclasses. Pourtant, malgré ces limitations nous voudrions toujours utiliser le motif Composite dans nos programmes.

La solution habituelle à ce problème est de considérer les motifs de conception essentiellement comme une façon d'organiser ses pensées, sans pour cela avoir un soutien du langage de programmation. Un motif peut exister seulement dans l'esprit du programmeur. Les motifs de conception peuvent alors être utilisés dans les langages comme C++ ou Java, même s'ils ne peuvent pas être implémentés comme des abstractions dans ces langages. Cela peut être facilité avec un préprocesseur du code source. Le programmeur peut ainsi programmer directement avec des motifs de conception et le préprocesseur génère le code source pour le langage cible.

6.5 LE LANGAGE JAVA (PARTIE SÉQUENTIELLE)

Java est un langage concurrent orienté objet avec une syntaxe qui ressemble à C++. Cette section donne une brève introduction à la partie séquentielle de Java. Nous expliquons comment écrire un programme simple, comment définir des classes et comment utiliser l'héritage. Nous ne parlons pas de la réflection, qui permet de faire beaucoup de ce que notre système à objets sait faire (mais de manière plus verbeuse).

Java est presque un langage orienté objet pur : presque toutes les entités sont des objets. Il n'y a qu'un petit ensemble de types primitifs, les entiers, flottants, booléens et caractères, qui utilisent le style ADT et qui ne sont donc pas des objets. Java est un langage relativement propre avec une sémantique relativement simple. Malgré la ressemblance syntaxique, il y a une grande différence en philosophie entre Java et C++ [7, 90]. C++ donne accès à la représentation machine des données et fait une traduction directe aux instructions machine. Il a aussi une gestion de mémoire manuelle. À cause de ces propriétés, C++ peut souvent être utilisé à la place du langage d'assemblage. En revanche, Java cache la représentation des données et fait une gestion automatique de mémoire. Il soutient la programmation répartie. Il a un système à objets plus sophistiqué. Ces propriétés font que Java est plus commode pour le développement d'applications générales.

6.5.1 Le modèle de calcul

Java fait de la programmation orientée objet statiquement typé avec des classes, des objets passifs (qui ne contiennent pas de fils) et des fils. Le modèle de calcul de Java est un modèle avec état et concurrence, sans les variables dataflow, le calcul paresseux et les noms. Le passage de paramètres est toujours par valeur, pour les types primitifs et les références aux objets. Les variables déclarées ont une valeur initiale par défaut qui dépend de leur type. Il y a un soutien pour l'affectation unique : les variables et les attributs d'objets peuvent être déclarés comme `final`, ce qui veut dire que la variable peut être affectée exactement une fois. Les variables finales doivent être affectées avant leur utilisation.

Java introduit sa propre terminologie pour certains concepts. Les classes contiennent des champs (attributs, dans notre terminologie), des méthodes, d'autres classes ou des interfaces, qui sont connus collectivement sous le nom de membres de classe. Les variables sont des champs ou des variables locales (déclarées dans les blocs locaux aux méthodes) ou des paramètres des méthodes. Les variables sont déclarées avec leur type, leur identificateur et un jeu facultatif de modificateurs (comme par exemple `final`). Le concept de **`self`** s'appelle `this`.

Les interfaces

Java a une solution élégante aux problèmes de l'héritage multiple (voir sections 6.4.2 et 6.4.3). Java introduit le concept d'interface, qui ressemble syntaxiquement à une classe avec seulement des déclarations de méthodes. Une interface n'a pas d'implémentation. Une classe peut implémenter une interface, ce qui veut simplement dire qu'elle définit toutes les méthodes spécifiées par l'interface. Java permet l'héritage simple pour les classes, ce qui évite les problèmes de l'héritage multiple. Mais pour garder les avantages de l'héritage multiple, Java permet l'héritage multiple pour les interfaces.

Java peut faire de la programmation d'ordre supérieur par un codage (les objets sont codés comme valeurs procédurales). Java a aussi un soutien plus direct pour la programmation d'ordre supérieur avec les classes internes (« *inner classes* »). Une classe interne est une définition de classe qui est imbriquée dans une autre classe ou dans un bloc de code (par exemple un corps de méthode). Une instance d'une classe interne peut sortir d'une méthode ou d'un bloc de code. Une classe interne peut avoir des références externes, mais il y a une restriction si elle est imbriquée dans un bloc de code : dans ce cas elle ne pourra pas référencer des variables non finales. Nous pouvons dire qu'une instance d'une classe interne est presque une valeur procédurale. La restriction existe vraisemblablement parce que les concepteurs du langage voulaient que les variables non finales dans les blocs de code puissent être implémentées sur une pile, et enlevées à la sortie de la méthode. Sans la restriction, cela pourrait créer des pointeurs détachés.

6.5.2 Une introduction à la programmation en Java

Nous donnons ici une brève introduction à la programmation en Java. Nous expliquons comment écrire un programme simple, comment définir des classes et comment utiliser l'héritage. Cette section n'explique qu'une petite partie de ce qui est possible en Java. Pour plus d'informations, nous vous recommandons un des nombreux bons livres sur la programmation en Java [7, 24].

Un programme simple

Nous voulons calculer la fonction factorielle. En Java, les fonctions sont définies comme des méthodes qui renvoient un résultat :

```
class Factorial {
   public long fact(long n)  {
      long f=1;
      for (int i=1; i<=n; i++) f=f*i;
      return f;
   }
}
```

Les instructions se terminent avec un point-virgule « ; » et les instructions composées sont entourées d'accolades `{...}`. Les identificateurs sont déclarés en mettant leur type d'abord, comme dans `long f`. L'affectation est désignée par un signe d'égalité « = ». Dans notre système à objets cela devient :

```
class Factorial
   meth fact(N ?X)
   F={NewCell 1} in
      for I in 1..N do F:=@F*I end
      X=@F
   end
end
```

Remarquez que `i` est une variable affectable (une cellule) qui est modifiée à chaque itération, tandis que `I` est un identificateur qui est déclaré de nouveau à chaque itération. La factorielle peut aussi être définie récursivement :

```
class Factorial {
    public long fact(long n) {
       if (n==0) return 1;
       else return n*this.fact(n-1);
    }
}
```

Dans notre système à objets cela devient :

```
class Factorial
    meth fact(N ?F)
       if N==0 then F=1
       else F=N*{self fact(N-1 $)} end
    end
end
```

Il y a quelques différences avec notre système à objets. Le mot clé `this` en Java a le même sens que **`self`** dans notre système à objets. Java est statiquement typé : le type de toutes les variables est déclaré à la compilation. Notre modèle est dynamiquement typé : une variable peut être liée à une entité de tout type. En Java, la visibilité de `fact` est déclarée comme publique. Dans notre modèle, `fact` est publique par défaut ; pour avoir une autre visibilité, il faut la déclarer comme un nom.

L'entrée/sortie

Tout programme réaliste en Java doit faire de l'entrée/sortie. Java a un sous-système compliqué d'entrée/sortie basé sur la notion de flot (« *stream* »). Un flot en Java est une séquence ordonnée de données qui a une source (pour un flot d'entrée) ou une destination (pour un flot de sortie). Il ne faut pas confondre avec les flots du chapitre 4, qui sont des listes avec une queue non liée. Le concept de flot en Java généralise le concept Unix d'entrée/sortie standard : l'entrée standard (« *standard input* ») (`stdin`) et la sortie standard (« *standard output* ») (`stdout`).

Un flot peut coder beaucoup de types, y compris des types primitifs, des objets et des graphes d'objets (un graphe d'objets est un objet avec les autres objets qu'il référence, directement ou indirectement). Un flot peut contenir des octets ou des caractères. Un caractère prend plus de place qu'un octet parce que Java soutient l'Unicode. Un octet en Java est un entier positif ou zéro de 8 bits. Un caractère en Java est un caractère Unicode 2.0, qui a un code de 16 bits.

La définition des classes

La classe `Factorial` est assez atypique. Elle n'a qu'une méthode et aucun attribut. Voici une classe plus réaliste. Cette classe définit des points dans un espace bidimensionnel :

```
class Point {
   public double x, y;
}
```

Les attributs `x` et `y` sont publics, ce qui signifie qu'ils sont visibles à l'extérieur de la classe. Les attributs publics ne sont généralement pas une bonne idée ; il est presque toujours mieux de les rendre privés et d'utiliser des méthodes d'accès :

```
class Point {
   double x, y;
   Point(double x1, y1) { x=x1; y=y1; }
   public double getX() { return x; }
   public double getY() { return y; }
}
```

La méthode `Point` s'appelle un **constructeur** ; elle est utilisée pour initialiser de nouveaux objets créés avec `new`, comme dans :

```
Point p=new Point(10.0, 20.0);
```

qui crée le nouvel objet `p` de classe `Point`. Nous ajoutons quelques méthodes pour calculer avec des points :

```
class Point {
   double x, y;
   Point(double x1, y1) { x=x1; y=y1; }
   public double getX() { return x; }
   public double getY() { return y; }
   public void origin() { x=0.0; y=0.0; }
   public void add(Point p) { x+=p.getX(); y+=p.getY(); }
   public void scale(double s) { x*=s; y*=s; }
}
```

L'argument `p` de `add` est une variable locale dont la valeur initiale est une référence à l'argument effectif. Dans notre système à objets, on peut définir `Point` ainsi :

```
class Point
   attr x y
   meth init(X Y) x:=X y:=Y end
   meth getX(X) X=@x end
   meth getY(Y) Y=@y end
   meth origin x:=0.0 y:=0.0 end
   meth add(P) x:=@x+{P getX($)} y:=@y+{P getY($)} end
   meth scale(S) x:=@x*S y:=@y*S end
end
```

Cette définition est similaire à la définition Java. La différence principale est que les arguments de `add` et `scale` ne sont pas dans des cellules locales (ce n'est pas nécessaire dans l'exemple). Il y a quelques différences mineures de syntaxe, comme les opérateurs `+=` et `*=`. Les deux définitions ont des attributs privés. Il y a une différence subtile dans la visibilité des attributs. En Java, les attributs privés sont visibles pour tous les objets de la même classe. On peut donc écrire la méthode `add` différemment :

```
public void add(Point p) { x+=p.x; y+=p.y; }
```

Vous trouverez des explications plus détaillées dans la section 6.3.4.

Le passage de paramètres et le programme principal

Dans les méthodes, Java utilise le passage par valeur. Une copie de la valeur est passée à la méthode et peut être modifiée à l'intérieur de la méthode sans changer la valeur originale. Pour les valeurs primitives comme les entiers et flottants, c'est clair. Pour les objets, Java passe les références aux objets (pas les objets eux-mêmes) par valeur. Les objets utilisent donc presque le passage par référence. La différence est que, à l'intérieur de la méthode, le champ peut être modifié pour référencer un autre objet.

La figure 6.13 donne un exemple. Cet exemple est un programme complet autonome ; on peut le compiler et exécuter tel quel. Chaque programme Java a une méthode, `main`, qui est appelée quand le programme est démarré. La référence à l'objet `c` est passée par valeur à la méthode `sqr`. À l'intérieur de `sqr`, l'affectation `a=null` n'a aucun effet sur `c`.

```
class MyInteger {
   public int val;
   MyInteger(int x) { val=x; }
}

class CallExample {
   public static void sqr(MyInteger a) {
      a.val=a.val*a.val;
      a=null;
   }

   public static void main(String[] args) {
      MyInteger c=new MyInteger(25);
      CallExample.sqr(c);
      System.out.println(c.val);
   }
}
```

Figure 6.13 Le passage de paramètres en Java.

L'argument de `main` est un tableau de chaînes de caractères qui contient les arguments de la ligne de commande du programme lorsqu'il est appelé par le système d'exploitation. L'appel de méthode `System.out.println` imprime son argument sur la sortie standard.

L'héritage

Nous pouvons utiliser l'héritage pour étendre la classe `Point`. Par exemple, elle peut être étendue pour représenter un pixel, qui est la surface indépendamment affichable la plus petite sur un dispositif de sortie graphique bidimensionnelle comme un écran d'ordinateur. Les pixels ont des coordonnées, comme les points, mais ils ont aussi une couleur.

```
class Pixel extends Point {
   Color color;
   public void origin() {
      super.origin();
      color=null;
   }
   public Color getC() { return color; }
   public void setC(Color c) { color=c; }
}
```

Le mot clé `extends` désigne l'héritage ; il correspond au mot clé **`from`** dans notre système à objets. Nous supposons que la classe `Color` est définie ailleurs. La classe `Pixel` redéfinit la méthode `origin`. La nouvelle méthode `origin` initialise le point et sa couleur. Elle utilise `super` pour accéder à la méthode redéfinie dans la classe ancêtre immédiate. Par rapport à la classe de base, cette classe s'appelle la superclasse en Java. Dans notre système à objets, nous pouvons définir `Pixel` ainsi :

```
class Pixel from Point
   attr color
   meth origin
      Point,origin
      color:=null
   end
   meth getC(C) C=@color end
   meth setC(C) color:=C end
end
```

6.6 EXERCICES

➤ **Exercice 1** — *Les objets non initialisés*

La fonction `New` crée un nouvel objet quand on lui donne une classe et un message initial. Écrivez une autre fonction `New2` qui n'a pas besoin d'un message initial. L'appel `Obj={New2 Class}` doit créer un nouvel objet sans l'initialiser.

Indice : Écrivez `New2` en utilisant `New`.

➤ **Exercice 2** — *Les méthodes protégées dans le sens de Java*
Une méthode protégée en Java a deux aspects : elle est accessible partout dans le paquet (« *package* ») qui définit la classe et aussi par les descendants de la classe. Pour cet exercice, définissez une abstraction linguistique qui permet l'annotation d'une méthode ou attribut comme `protected` dans le sens Java. Montrez comment la coder dans le modèle de la section 6.3.4 avec les noms. Utilisez les foncteurs pour représenter les paquets Java. Par exemple, une approche pourrait consister à définir le nom globalement dans le foncteur et aussi de le stocker dans un attribut qui s'appelle `setOfAllProtectedAttributes`. Comme l'attribut est hérité, le nom de la méthode sera visible dans toutes les sous-classes. Élaborez les détails de cette approche.

➤ **Exercice 3** — (exercice avancé) *L'héritage sans état explicite*
L'héritage n'a pas besoin d'état explicite ; les deux concepts sont orthogonaux. Pour cet exercice, concevez et implémentez un système à objets avec des classes et l'héritage mais sans état explicite. Un point de départ possible est l'implémentation des objets déclaratifs dans la section 5.4.2.

➤ **Exercice 4** — (projet de recherche) *La programmation des motifs de conception*
Pour cet exercice, concevez un langage orienté objet qui permet l'héritage « ascendant » (la définition d'une nouvelle superclasse d'une classe) et la programmation d'ordre supérieur. L'héritage ascendant est parfois appelé la généralisation. Implémentez et évaluez l'utilité de votre langage. Montrez comment programmer les motifs de conception de Gamma *et al.* [27] comme des abstractions dans votre langage. Avez-vous besoin d'autres nouvelles opérations en plus de la généralisation ?

Annexe **A**

L'environnement de développement du système Mozart

Prenez garde aux ides de mars.
– Augure à Julius Caesar, William Shakespeare (1564-1616)

Le système Mozart a un environnement de développement interactif (IDE, « *Interactive Development Environment* »). Mozart a aussi des outils pour la ligne de commande. Cette annexe donne un résumé de l'interface interactive et de l'interface de commande. Nous vous conseillons de voir la documentation de Mozart pour plus d'informations [53].

Pour l'exécution des exemples de ce livre, nous vous conseillons de regarder aussi le *Labo interactif*, un outil qui est expliqué sur la page xiii. Cet outil est conçu pour accompagner le livre. Il est plus facile à utiliser que le système Mozart pour l'exécution des exemples du livre.

A.1 L'INTERFACE INTERACTIVE

Le système Mozart a une interface interactive basée sur l'éditeur de texte Emacs. L'interface interactive s'appelle l'OPI, l'interface de programmation Oz (« *Oz Programming Interface* »). L'OPI a plusieurs mémoires tampon (« *buffers* ») : le bloc-notes (« *scratch pad* »), l'émulateur Oz, le compilateur Oz et un tampon pour chaque fichier ouvert. L'interface donne accès à plusieurs outils : le compilateur incrémental (qui peut compiler tout fragment de programme légal), le Browser (pour visualiser la mémoire à affectation unique), le Panel (pour voir l'utilisation des ressources en temps et en espace), le Compiler Panel (pour paramétrer l'environnement du compilateur),

l'Explorer (pour la résolution interactive graphique des problèmes à contraintes) et le Distribution Panel (pour la programmation répartie). Ces outils peuvent aussi être manipulés à partir des programmes, par exemple le module `Compiler` peut compiler les chaînes de caractères qui représentent du code source.

A.1.1 Les commandes de l'interface interactive

Vous pouvez accéder à toutes les commandes OPI importantes avec les menus en haut de la fenêtre. La plupart de ces commandes ont un équivalent clavier. Les commandes les plus importantes sont énumérées dans le tableau A.1. La notation « `CTRL-x` » signifie de garder appuyée la touche Control (ctrl) et ensuite d'appuyer sur la touche `x` une fois. La commande `CTRL-g` est particulièrement utile pour corriger les erreurs de frappe. Une région est une partie contiguë d'un tampon. Elle peut être sélectionnée en glissant la souris et en appuyant sur le premier bouton (à gauche) de la souris. Un paragraphe est un ensemble de lignes de texte non vides entourées par des lignes vides ou le début ou la fin du tampon.

Commande	Effet
`CTRL-x CTRL-f`	Lire un fichier dans un nouveau tampon de l'éditeur
`CTRL-x CTRL-s`	Sauvegarder le tampon actif dans son fichier
`CTRL-x i`	Insérer un fichier dans le tampon actif
`CTRL-. CTRL-l`	Exécuter la ligne sélectionnée dans Mozart
`CTRL-. CTRL-r`	Exécuter la région sélectionnée dans Mozart
`CTRL-. CTRL-p`	Exécuter le paragraphe sélectionné dans Mozart
`CTRL-. CTRL-b`	Exécuter le tampon actif dans Mozart
`CTRL-. h`	Arrêter l'exécution de Mozart (mais garder l'éditeur)
`CTRL-x CTRL-c`	Arrêter tout le système (y compris l'éditeur)
`CTRL-. e`	Basculer le tampon de l'émulateur
`CTRL-. c`	Basculer le tampon du compilateur
`CTRL-x 1`	Maximiser le tampon actif
`CTRL-g`	Annuler la commande en cours

Tableau A.1 Quelques commandes de l'interface interactive de Oz.

Le tampon de l'émulateur affiche les messages de l'exécution. Elle affiche la sortie de `Show` et les erreurs à l'exécution, comme des exceptions non capturées. Le tampon du compilateur affiche les messages du compilateur. Elle affiche les confirmations du code source compilé avec succès et les erreurs de compilation.

A.1.2 L'utilisation interactive des foncteurs

Les foncteurs sont des spécifications de modules utilisés pour la construction des programmes. Instancier un foncteur crée un module. Un module est une entité qui regroupe d'autres entités. Les modules peuvent contenir des enregistrements, des procédures, des objets, des classes, des fils en exécution et toute autre entité qui existe à l'exécution.

Les foncteurs sont des unités de compilation : leur code source, mis dans un fichier, peut être compilé en une fois. Les foncteurs peuvent aussi être utilisés dans l'interface interactive, selon le principe dans Mozart que tout peut être fait interactivement.

– Un foncteur compilé peut être chargé et ses liens édités interactivement. Par exemple, supposez que le module `Set`, qui est sur le site Web du livre, est compilé dans le fichier `Set.ozf`. Il sera chargé et ses liens édités interactivement avec le code suivant :

```
declare
[Set]={Module.link ["Set.ozf"]}
```

Cela crée et édite les liens du module `Set`. La fonction `Module.link` prend une liste de noms de fichier ou d'URL et renvoie une liste de modules.

– Un foncteur est une valeur, comme une classe. Il peut être défini interactivement avec une syntaxe analogue à la syntaxe des classes :

```
F=functor $ define skip end
```

Cela définit un foncteur et le lie à `F`. Nous pouvons créer un module à partir de `F` comme ceci :

```
declare
[M]={Module.apply [F]}
```

Cela crée et édite les liens du module `M`. La fonction `Module.apply` prend une liste de valeurs de foncteur et renvoie une liste de modules.

Pour plus d'opérations sur les foncteurs, consultez la documentation du module `Module` [22].

A.2 L'INTERFACE DE COMMANDE

Le système Mozart peut être utilisé à partir d'une ligne de commande. Les fichiers Oz peuvent être compilés et leurs liens édités. Chaque fichier source à compiler doit contenir un foncteur. Par exemple, supposez que nous ayons le fichier source `Set.oz`. Nous créons le foncteur compilé `Set.ozf` avec la commande suivante sur la ligne de commande :

```
ozc -c Set.oz
```

Nous pouvons créer un fichier exécutable autonome `Set` avec la commande suivante :

```
ozc -x Set.oz
```

(Dans le cas de `Set.oz`, le fichier autonome fait très peu : il va simplement définir les opérations sur les ensembles.) Le choix par défaut dans Mozart est d'utiliser l'édition dynamique des liens, c'est-à-dire que les modules sont chargés et leurs liens édités au moment où on en a besoin dans une application. Cela réduit la taille des fichiers compilés. Mais il est possible d'éditer les liens de tous les modules importés pendant la compilation (édition statique des liens) pour éviter l'édition dynamique des liens.

Annexe **B**

La syntaxe du langage Oz

Le diable est dans les détails.
– Proverbe traditionnel.

Dieu est dans les détails.
– Proverbe traditionnel.

Je ne sais pas ce qu'il y a dans ces détails,
mais cela doit être quelque chose d'important !
– Proverbe impertinent.

Cette annexe définit la syntaxe complète du langage Oz utilisé dans ce livre et dans [97], y compris tous les raccourcis syntaxiques. Le langage est un sous-ensemble du langage Oz implémenté par le système Mozart. L'annexe contient six sections :

- La section B.1 définit la syntaxe des instructions interactives, c'est-à-dire les instructions qui peuvent être utilisées dans l'interface interactive.
- La section B.2 définit la syntaxe des instructions et des expressions.
- La section B.3 définit la syntaxe des non terminaux dont on a besoin pour la syntaxe des instructions et expressions.
- La section B.4 donne une liste des opérateurs du langage avec leur précédence et associativité.
- La section B.5 donne une liste des mots clés du langage.
- La section B.6 définit la syntaxe lexicale du langage, c'est-à-dire comment une séquence de caractères est transformée en séquence de jetons.

Cette annexe définit une syntaxe hors-contexte pour le langage Oz. Cela rend la syntaxe simple et facile à lire. Le désavantage d'une syntaxe hors-contexte est qu'elle ne contient pas toutes les conditions syntaxiques nécessaires pour les programmes légaux. Par exemple, prenez l'instruction **`local`** `X` **`in`** ⟨statement⟩ **`end`**. L'instruction qui contient celle-ci doit déclarer tous les identificateurs libres de ⟨statement⟩ sauf `X`. Ce n'est pas une condition hors-contexte.

Cette annexe définit une syntaxe pour un sous-ensemble du langage défini dans [21, 39]. Cette annexe fait quelques changements par rapport à [39] : elle introduit les constructions imbriquables (« *nestable constructs* »), les déclarations imbriquables (« *nestable declarations* ») et les termes, pour factoriser les parties communes de la syntaxe des instructions et des expressions ; elle définit des instructions interactives et des boucles **`for`** ; elle omet la traduction vers le langage noyau (qui est donné dans le corps du livre pour chaque abstraction linguistique) ; et elle fait quelques autres petites simplifications pour la clarté (mais sans sacrifier la précision).

B.1 LES INSTRUCTIONS INTERACTIVES

Le tableau B.1 donne la syntaxe des instructions interactives. Une instruction interactive est un sur-ensemble d'une instruction ; en plus de toutes les instructions normales, elle peut contenir l'instruction **`declare`**. Il faut toujours donner des instructions interactives à l'interface interactive. Tous les identificateurs libres dans une instruction interactive doivent exister dans l'environnement global ; sinon le système donne une erreur « *variable not introduced* » (variable non introduite).

⟨interStmt⟩	::=	⟨statement⟩
	|	**`declare`** { ⟨declPart⟩ }+ [⟨interStmt⟩]
	|	**`declare`** { ⟨declPart⟩ }+ **`in`** ⟨interStmt⟩

Tableau B.1 Les instructions interactives.

B.2 LES INSTRUCTIONS ET LES EXPRESSIONS

Les tableau B.2 donne la syntaxe des instructions et des expressions. La plupart des constructions du langage peuvent être utilisées comme instruction ou comme expression. Nous appelons ces constructions imbriquables. Nous écrivons les règles de grammaire pour définir leur syntaxe une seule fois, à la fois pour l'utilisation en tant qu'instruction et en tant qu'expression. Le tableau B.3 donne la syntaxe pour toutes les constructions imbriquables sauf les déclarations. Le tableau B.4 donne la syntaxe pour les déclarations imbriquables. Les règles de grammaire pour les constructions et déclarations imbriquables sont des gabarits avec un argument. Le

gabarit est instancié chaque fois qu'il est utilisé. Par exemple, ⟨nestCon(α)⟩ définit le gabarit pour les constructions imbriquables sauf les déclarations. Ce gabarit est utilisé deux fois, comme ⟨nestCon(statement)⟩ et comme ⟨nestCon(expression)⟩, ce qui donne chaque fois une règle de grammaire.

⟨statement⟩	: :=	⟨nestCon(statement)⟩ | ⟨nestDec(⟨variable⟩)⟩
	|	**skip** | ⟨statement⟩ ⟨statement⟩
⟨expression⟩	: :=	⟨nestCon(expression)⟩ | ⟨nestDec(´$´)⟩
	|	⟨unaryOp⟩ ⟨expression⟩
	|	⟨expression⟩ ⟨evalBinOp⟩ ⟨expression⟩
	|	´$´ | ⟨term⟩ | **self**
⟨inStatement⟩	: :=	[{ ⟨declPart⟩ }+ **in**] ⟨statement⟩
⟨inExpression⟩	: :=	[{ ⟨declPart⟩ }+ **in**]
		[⟨statement⟩] ⟨expression⟩
⟨in(statement)⟩	: :=	⟨inStatement⟩
⟨in(expression)⟩	: :=	⟨inExpression⟩

Tableau B.2 Les instructions et les expressions.

⟨nestCon(α)⟩ : :=
⟨expression⟩ (´=´ | ':=' | ´,´) ⟨expression⟩
| ´{´ ⟨expression⟩ { ⟨expression⟩ } ´}´
| **local** { ⟨declPart⟩ }+ **in** [⟨statement⟩] ⟨α⟩ **end**
| ´(´ ⟨in(α)⟩ ´)´
| **if** ⟨expression⟩ **then** ⟨in(α)⟩
{ **elseif** ⟨expression⟩ **then** ⟨in(α)⟩ }
[**else** ⟨in(α)⟩] **end**
| **case** ⟨expression⟩ **of** ⟨pattern⟩ [**andthen** ⟨expression⟩]
then ⟨in(α)⟩
{ ´[]´ ⟨pattern⟩ [**andthen** ⟨expression⟩] **then** ⟨in(α)⟩ }
[**else** ⟨in(α)⟩] **end**
| **for** { ⟨loopDec⟩ }+ **do** ⟨in(α)⟩ **end**
| **try** ⟨in(α)⟩
[**catch** ⟨pattern⟩ **then** ⟨in(α)⟩
{ ´[]´ ⟨pattern⟩ **then** ⟨in(α)⟩ }]
[**finally** ⟨inStatement⟩] **end**
| **raise** ⟨inExpression⟩ **end**
| **thread** ⟨in(α)⟩ **end**
| **lock** [⟨expression⟩ **then**] ⟨in(α)⟩ **end**

Tableau B.3 Les constructions imbriquables (sans les déclarations).

```
⟨nestDec(α)⟩ : :=
    proc ´{´ α { ⟨pattern⟩ } ´}´ ⟨inStatement⟩ end
  | fun [ lazy ] ´{´ α { ⟨pattern⟩ } ´}´ ⟨inExpression⟩ end
  | functor α
    [ import { ⟨variable⟩ [ at ⟨atom⟩ ]
             | ⟨variable⟩ ´(´
                 { (⟨atom⟩ | ⟨int⟩) [ ':' ⟨variable⟩ ] }+ ´)´
             }+ ]
    [ export { [ (⟨atom⟩ | ⟨int⟩) ':' ] ⟨variable⟩ }+ ]
    define { ⟨declPart⟩ }+ [ in ⟨statement⟩ ] end
  | class α { ⟨classDesc⟩ }
    { meth ⟨methHead⟩ [ ´=´ ⟨variable⟩ ]
      ( ⟨inExpression⟩ | ⟨inStatement⟩ ) end }
    end
```

Tableau B.4 Les déclarations imbriquables.

B.3 LES AUTRES SYMBOLES NON TERMINAUX

Les tableaux B.5 et B.6 définissent les symboles non terminaux nécessaires pour la syntaxe des instructions et des expressions. Le tableau B.5 définit la syntaxe des termes et des formes (« *patterns* »). Il y a une relation proche entre termes et formes. Les deux sont utilisés pour définir les valeurs partielles. Il n'y a que deux différences : (1) les termes peuvent contenir toutes les expressions mais les formes ne le peuvent pas, et (2) les formes peuvent être partielles (avec ´...´), mais les termes ne le peuvent pas.

```
⟨term⟩    : := [ ´!´ ] ⟨variable⟩ | ⟨int⟩ | ⟨float⟩ | ⟨character⟩
             | ⟨atom⟩ | ⟨string⟩ | unit | true | false
             | ⟨label⟩ ´(´ { [ ⟨feature⟩ ':' ] ⟨expression⟩ } ´)´
             | ⟨expression⟩ ⟨consBinOp⟩ ⟨expression⟩
             | ´[´ { ⟨expression⟩ }+ ´]´
⟨pattern⟩ : := [ ´!´ ] ⟨variable⟩ | ⟨int⟩ | ⟨float⟩ | ⟨character⟩
             | ⟨atom⟩ | ⟨string⟩ | unit | true | false
             | ⟨label⟩ ´(´ { [ ⟨feature⟩ ':' ] ⟨pattern⟩ } [ ´...´ ] ´)´
             | ⟨pattern⟩ ⟨consBinOp⟩ ⟨pattern⟩
             | ´[´ { ⟨pattern⟩ }+ ´]´
```

Tableau B.5 Les termes et les formes.

```
⟨declPart⟩    : := ⟨variable⟩ | ⟨pattern⟩ ´=´ ⟨expression⟩ | ⟨statement⟩
⟨loopDec⟩     : := ⟨variable⟩ in ⟨expression⟩ [ ´..´ ⟨expression⟩ ]
                    [ ´;´ ⟨expression⟩ ]
                  | ⟨variable⟩ in
                    ⟨expression⟩ ´;´ ⟨expression⟩ ´;´ ⟨expression⟩
                  | break ':' ⟨variable⟩ | continue ':' ⟨variable⟩
                  | return ':' ⟨variable⟩ | default ':' ⟨expression⟩
                  | collect ':' ⟨variable⟩
⟨unaryOp⟩     : := ´~´ | ´@´ | ´!!´
⟨binaryOp⟩    : := ⟨consBinOp⟩ | ⟨evalBinOp⟩
⟨consBinOp⟩   : := ´#´ | ´|´
⟨evalBinOp⟩   : := ´+´ | ´-´ | ´*´ | ´/´ | div | mod | andthen | orelse
                  | ':=' | ´,´ | ´=´ | ´==´ | ´\=´ | ´<´ | ´=<´ | ´>´ | ´>=´
                  | ´.´ | '::' | '=:' | '\=:' | '=<:'
⟨label⟩       : := unit | true | false | ⟨variable⟩ | ⟨atom⟩
⟨feature⟩     : := unit | true | false | ⟨variable⟩ | ⟨atom⟩ | ⟨int⟩
⟨classDesc⟩   : := from { ⟨expression⟩ }+ | prop { ⟨expression⟩ }+
                  | attr { ⟨attrInit⟩ }+
⟨attrInit⟩    : := ( [ ´!´ ] ⟨variable⟩ | ⟨atom⟩ | unit | true | false )
                    [ ':' ⟨expression⟩ ]
⟨methHead⟩    : := ( [ ´!´ ] ⟨variable⟩ | ⟨atom⟩ | unit | true | false )
                    [ ´(´ { ⟨methArg⟩ } [ ´...´ ] ´)´ ]
                    [ ´=´ ⟨variable⟩ ]
⟨methArg⟩     : := [ ⟨feature⟩ ':' ] ( ⟨variable⟩ | ´_´ | ´$´ )
                    [ ´<=´ ⟨expression⟩ ]
```

Tableau B.6 Les autres symboles non terminaux.

Le tableau B.6 définit les non terminaux pour les déclarations dans les instructions **declare**, **local** et **define**, pour les opérateurs unaires, pour les opérateurs binaires (les opérateurs « constructifs » ⟨consBinOp⟩ et les opérateurs « évaluatifs » ⟨evalBinOp⟩), pour les enregistrements (étiquettes et traits) et pour les classes (descripteurs, attributs, méthodes, etc.).

B.4 LES OPÉRATEURS

Le tableau B.7 donne la précédence et l'associativité de tous les opérateurs utilisés dans ce livre et dans [97]. Tous les opérateurs sont binaires et infixés sauf dans trois cas. Le signe minus « ~ » est un opérateur unaire préfixé. Le symbole dièse « # » est

un opérateur n-aire mixfixé. Le symbole « `. :=` » est un opérateur ternaire infixé qui est expliqué dans la section suivante. Il n'y a pas d'opérateurs postfixés. Les opérateurs sont classés dans l'ordre croissant de précédence. Plus la précédence augmente, plus ils sont liés étroitement. Il y a quatre associativités :

- Gauche. Pour les opérateurs binaires, cela signifie que les opérateurs répétés sont groupés à gauche. Par exemple, `1+2+3` a le même sens que `((1+2)+3)`.
- Droite. Pour les opérateurs binaires, cela signifie que les opérateurs répétés sont groupés à droite. Par exemple, `a|b|X` a le même sens que `(a|(b|X))`.
- Mixfix. Les opérateurs répétés sont en réalité un seul opérateur, et toutes les expressions sont des arguments de l'opérateur. Par exemple, `a#b#c` est un raccourci syntaxique pour le tuple `´#´(a b c)`.
- Aucune. Pour les opérateurs binaires, cela signifie que l'opérateur ne peut pas être répété. Par exemple, `1<2<3` est une erreur.

Les parenthèses peuvent être utilisées pour changer la précédence.

Opérateur	Associativité
`=`	droite
`:=` « `. :=` »	droite
`orelse`	droite
`andthen`	droite
`== \= < =< > >= =: \=: =<:`	aucune
`::`	aucune
`|`	droite
`#`	mixfix
`+ -`	gauche
`* /` **`div mod`**	gauche
`,`	droite
`~`	gauche
`.`	gauche
`@ ! !`	gauche

Tableau B.7 Les opérateurs avec leur précédence et leur associativité.

B.4.1 L'opérateur ternaire

Il y a un opérateur ternaire (à trois arguments), « `. :=` », qui est conçu pour les affectations des dictionnaires et tableaux (« *arrays* »). Il a la même précédence et associativité que `:=`. Il peut être utilisé dans la position d'une expression comme `:=`, où il a l'effet d'un échange (`Exchange`). L'instruction `S.I:=X` contient un opérateur ternaire avec les arguments `S`, `I` et `X`. Cette instruction est utilisée pour affecter les

dictionnaires et les tableaux. Il ne faut pas la confondre avec `(S.I):=X`, qui contient deux opérateurs binaires imbriqués, `.` et `:=`. Cette dernière instruction est utilisée pour affecter une cellule à l'intérieur d'un dictionnaire. Les parenthèses sont très importantes ! La figure B.1 montre la différence en syntaxe abstraite entre `S.I:=X` et `(S.I):=X`. Dans la figure, *(cellule)* signifie toute cellule ou attribut d'objet et *(dictionnaire)* signifie tout dictionnaire ou tableau.

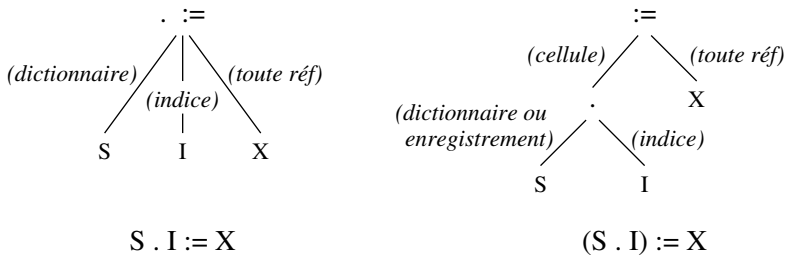


Figure B.1 L'opérateur ternaire « `.` `:=` ».

L'opérateur ternaire est important parce qu'un dictionnaire peut contenir des cellules. Pour affecter un dictionnaire `D`, nous écrivons `D.I:=X`. Pour affecter une cellule dans un dictionnaire qui contient des cellules, nous écrivons `(D.I):=X`. Cela a le même effet que **`local`** `C=D.I` **`in`** `C:=X` **`end`** mais c'est plus concis. Le premier argument de l'opérateur binaire `:=` doit être une cellule ou un attribut d'objet.

B.5 LES MOTS CLÉS

Le tableau B.8 contient les mots clés du langage en ordre alphabétique.

andthen	default	**false**	**lock**	**require** (*)
at	**define**	**feat** (*)	**meth**	return
attr	**dis** (*)	**finally**	**mod**	**self**
break	**div**	**for**	**not** (*)	**skip**
case	**do**	**from**	**of**	**then**
catch	**else**	**fun**	**or** (*)	**thread**
choice	**elsecase** (*)	**functor**	**orelse**	**true**
class	**elseif**	**if**	otherwise	**try**
collect	**elseof** (*)	**import**	**prepare** (*)	**unit**
cond (*)	**end**	**in**	**proc**	
continue	**export**	lazy	**prop**	
declare	**fail**	**local**	**raise**	

Tableau B.8 Les mots clés.

Les mots clés marqués par (*) existent en Oz mais ne sont pas utilisés dans le livre CTM [97] ni dans le présent livre. Les mots clés en gras peuvent être utilisés comme des atomes si on les met entre guillemets simples. Par exemple, ´then´ est un atome et **then** est un mot clé. Les mots clés qui ne sont pas en gras peuvent être utilisés comme atomes directement, sans guillemets.

B.6 LA SYNTAXE LEXICALE

Cette section définit la syntaxe lexicale de Oz, c'est-à-dire la façon dont une séquence de caractères est transformée en séquence de jetons.

B.6.1 Les jetons

Les identificateurs, les atomes, les chaînes et les caractères

Le tableau B.9 définit la syntaxe lexicale pour les identificateurs, les atomes, les chaînes et les caractères dans les chaînes. Contrairement aux sections précédentes, qui définissent des séquences de jetons, cette section définit des séquences de caractères. Un caractère alphanumérique est une lettre (en majuscule ou minuscule), un chiffre ou un caractère de soulignement. Les guillemets simples entourent les représentations des atomes qui peuvent contenir des caractères non alphanumériques et les guillemets arrières sont utilisés pareillement pour les identificateurs. Remarquez qu'un atome ne peut pas avoir la même séquence de caractères qu'un mot clé sauf si l'atome est entre guillemets. Le tableau B.10 définit les non terminaux nécessaires pour le tableau B.9. « Tout caractère » signifie aussi les caractères de contrôle et les caractères avec des accents. Le caractère NUL a le code 0 (zéro).

⟨variable⟩	:=	(caract. en majuscule) { (caract. alphanumérique) }
	|	´`´ { ⟨variableChar⟩ | ⟨pseudoChar⟩ } ´`´
⟨atom⟩	:=	(caract. en minuscule) { (caract. alphanum.) } (sauf mot clé)
	|	´´´ { ⟨atomChar⟩ | ⟨pseudoChar⟩ } ´´´
⟨string⟩	:=	´"´ { ⟨stringChar⟩ | ⟨pseudoChar⟩ } ´"´
⟨character⟩	:=	(tout entier dans l'intervalle 0 . . 255)
	|	´&´ ⟨charChar⟩ | ´&´ ⟨pseudoChar⟩

Tableau B.9 La syntaxe lexicale des identificateurs, atomes, chaînes et caractères.

⟨variableChar⟩	: :=	(tout caractère sauf `, \ et NUL)
⟨atomChar⟩	: :=	(tout caractère sauf ´, \ et NUL)
⟨stringChar⟩	: :=	(tout caractère sauf ", \ et NUL)
⟨charChar⟩	: :=	(tout caractère sauf \ et NUL)
⟨pseudoChar⟩	: :=	´\´ ⟨octdigit⟩ ⟨octdigit⟩ ⟨octdigit⟩
		| (´\x´ | ´\X´) ⟨hexdigit⟩ ⟨hexdigit⟩
		| ´\a´ | ´\b´ | ´\f´ | ´\n´ | ´\r´ | ´\t´
		| ´\v´ | ´\\´ | ´\´´ | ´\"´ | ´\`´ | ´\&´

Tableau B.10 Les non terminaux nécessaires pour la syntaxe lexicale.

Les entiers et les nombres à virgule flottante

Le tableau B.11 définit la syntaxe lexicale des entiers et des nombres à virgule flottante. Remarquez que le signe moins est écrit avec un tilde ´~´.

⟨int⟩	: :=	[´~´] ⟨nzdigit⟩ { ⟨digit⟩ }
	|	[´~´] 0 { ⟨octdigit⟩ }+
	|	[´~´] (´0x´ | ´0X´) { ⟨hexdigit⟩ }+
	|	[´~´] (´0b´ | ´0B´) { ⟨bindigit⟩ }+
⟨float⟩	: :=	[´~´] { ⟨digit⟩ }+ ´.´ { ⟨digit⟩ }
		[(´e´ | ´E´) [´~´] { ⟨digit⟩ }+]
⟨digit⟩	: :=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨nzdigit⟩	: :=	1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨octdigit⟩	: :=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨hexdigit⟩	: :=	⟨digit⟩ | ´a´ | ´b´ | ´c´ | ´d´ | ´e´ | ´f´
	|	´A´ | ´B´ | ´C´ | ´D´ | ´E´ | ´F´
⟨bindigit⟩	: :=	0 | 1

Tableau B.11 La syntaxe lexicale des entiers et nombres à virgule flottante.

B.6.2 L'espace blanc et les commentaires

Les jetons peuvent être séparés par de l'espace blanc (« *white space* ») et des commentaires. L'espace blanc est un des caractères suivants : tabulation horizontale (code de caractère 9), saut de ligne (« *newline* ») (code 10), tabulation verticale (code 11), saut de page (« *form feed* ») (code 12), retour de chariot (« *carriage return* ») (code 13) et l'espace (code 32). Un commentaire est une des trois possibilités suivantes :

– Une séquence de caractères qui commence avec le caractère % (pourcentage) jusqu'à la fin de la ligne ou la fin du fichier (celle qui arrive en premier).

– Une séquence de caractères qui commence avec `/*` et s'arrête avec `*/`, inclus. Ce commentaire peut être imbriqué.
– Le caractère `?` (point d'interrogation). L'intention est de marquer les arguments de sortie des procédures, comme dans

```
proc {Max A B ?C} ... end
```

où `C` est une sortie. Un argument de sortie est un argument qui est lié dans la procédure.

Bibliographie

[1] ABELSON H., SUSSMAN G. J. & SUSSMAN J., *Structure and interpretation of computer programs*, MIT Press, Cambridge, MA, 1985.

[2] ———, *Structure and interpretation of computer programs,* 2nd edition, MIT Press, Cambridge, MA, 1996.

[3] AÏT-KACI H., « An introduction to LIFE-programming with logic, inheritance, functions, and equations », in *Logic Programming - Proceedings of the 1993 International Symposium* (Vancouver, Canada) (Miller D., ed.), MIT Press, 1993, p. 52-68.

[4] AÏT-KACI H. & LINCOLN P., « LIFE, a natural language for natural language », *T. A. Informations, revue internationale du traitement automatique du langage* **30** (1991), no. 1-2, p. 37-67 (ISSN 0039-8217).

[5] AÏT-KACI H. & NASR R., « LOGIN : A logic programming language with built-in inheritance », *Journal of Logic Programming* **3** (1986), no. 3, p. 185-215.

[6] AÏT-KACI H. & PODELSKI A., « Towards a meaning of LIFE », *Journal of Logic Programming* **16** (1993), no. 3-4, p. 195-234.

[7] ARNOLD K. & GOSLING J., *The Java programming language,* 2nd edition, Addison-Wesley, 1998.

[8] BACKUS J., « The history of FORTRAN I, II, and III », *ACM SIGPLAN Notices* **13** (1978), no. 8, p. 165-180.

[9] BASILI V. R. & TURNER A. J., « Iterative enhancement : A practical technique for software development », *IEEE Transactions on Software Engineering* **1** (1975), no. 4, p. 390-396.

[10] BECK K., *Test-driven development : by example*, Addison-Wesley, 2003.

[11] BERGIN J. & WINDER R., « Understanding object-oriented programming », 2000, Available at `csis.pace.edu/~bergin`.

[12] BROOKS F. P., JR., *The mythical man-month : Essays on software engineering*, Addison-Wesley, 1975.

[13] ———, *The mythical man-month : Essays on software engineering, anniversary edition*, Addison-Wesley, 1995.

[14] CARDELLI L. & WEGNER P., « On understanding types, data abstraction, and polymorphism », *Computing Surveys* **17** (1985), no. 4, p. 471-522.

[15] COLMERAUER A., « The birth of Prolog », *ACM SIGPLAN Notices* **28** (1993), no. 3, p. 37-52, Originally appeared in History of Programming Languages Conference (HOPL-II), 1993.

[16] COOK W. R., « Object-oriented programming versus abstract data types », in *REX Workshop/School on the Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science, vol. 173, Springer-Verlag, 1990, p. 151-178.

[17] CORMEN T. H., LEISERSON C. E. & RIVEST R. L., *Introduction to algorithms*, MIT Press, McGraw-Hill, Cambridge, MA, 1990.

[18] DARWIN C., *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*, Harvard University Press (originally John Murray, London, 1859).

[19] DIJKSTRA E. W., *A primer of Algol 60 programming*, Academic Press, 1962.

[20] ———, « Go To statement considered harmful », **11** (1968), no. 3, p. 147-148.

[21] DUCHIER D., « Loop support », Tech. report, Mozart Consortium, 2003, Available at `www.mozart-oz.org`.

[22] DUCHIER D., KORNSTAEDT L., MÜLLER T., SCHULTE C. & VAN ROY P., « System modules », Tech. report, Mozart Consortium, 2003, Available at `www.mozart-oz.org`.

[23] DUCHIER D., KORNSTAEDT L. & SCHULTE C., « The Oz base environment », Tech. report, Mozart Consortium, 2003, Available at `www.mozart-oz.org`.

[24] ECKEL B., *Thinking in Java,* 3rd edition, Prentice Hall PTR, 2002.

[25] FOWLER M. & SCOTT K., *UML distilled : A brief guide to the standard object modeling language*, Addison-Wesley Longman, 2000.

[26] FRIEDMAN D. P., WAND M. & HAYNES C. T., *Essentials of programming languages*, MIT Press, Cambridge, MA, 1992.

[27] GAMMA E., HELM R., JOHNSON R. & VLISSIDES J., *Design patterns : Elements of reusable object-oriented software*, Addison-Wesley, 1994.

[28] GOLDBERG A. & ROBSON D., *Smalltalk-80 : The language and its implementation*, Addison-Wesley, 1983.

[29] GOSLING J., JOY B. & STEELE G., *The Java language specification*, Addison-Wesley, 1996, Available at `www.javasoft.com`.

[30] GRAHAM P., *On Lisp*, Prentice Hall, Englewood Cliffs, NJ, 1993, Available for download from the author.

[31] GROLAUX D., « QTk : Graphical user interface design for Oz », 2003, Available at `www.mozart-oz.org/mozart-stdlib/index.html`.

[32] GROLAUX D., VAN ROY P. & VANDERDONCKT J., « `QTk`—a mixed declarative/procedural approach for designing executable user interfaces », in *8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01)* (Toronto, Canada), Lecture Notes in Computer Science, vol. 2254, Springer-Verlag, 2001, p. 109-110.

[33] ———, « `QTk`—an integrated model-based approach to designing executable user interfaces », in *8th Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS 2001)* (Glasgow, Scotland), 2001, GIST Technical Report G-2001-1, p. 77-91.

[34] GUNTER C. A. & MITCHELL J. C. (eds.), *Theoretical aspects of object-oriented programming*, MIT Press, Cambridge, MA, 1994.

[35] HARIDI S., « An Oz 2.0 tutorial », Available at `www.sics.se/~seif/oz.html`, 1996.

[36] HARIDI S. & JANSON S., « Kernel Andorra Prolog and its computation model », in *7th International Conference on Logic Programming*, MIT Press, 1990, p. 31-48.

[37] HARIDI S., VAN ROY P., BRAND P. & SCHULTE C., « Programming languages for distributed applications », *New Generation Computing* **16** (1998), no. 3, p. 223-261.

[38] HENZ M., *Objects for concurrent constraint programming*, vol. 426, International Series in Engineering and Computer Science, no. ISBN 0-7923-8038-X, Kluwer Academic Publishers, Boston, 1997.

[39] HENZ M. & KORNSTAEDT L., « The Oz notation », Tech. report, Mozart Consortium, 2003, Available at `www.mozart-oz.org`.

[40] HENZ M., SMOLKA G. & WÜRTZ J., « Oz—a programming language for multi-agent systems », in *13th International Joint Conference on Artificial Intelligence* (Chambéry, France) (Bajcsy R., ed.), Morgan Kaufmann, 1993, p. 404-409.

[41] ———, « Object-oriented concurrent constraint programming in Oz », in *Principles and Practice of Constraint Programming* (Cambridge, MA) (Van Hentenryck P. & Saraswat V., eds.), MIT Press, 1995, p. 29-48.

[42] INTELLIGENT SYSTEMS LABORATORY, SWEDISH INSTITUTE OF COMPUTER SCIENCE , « SICStus Prolog user's manual », 2003, Available at `www.sics.se/sicstus`.

[43] JAIN R., *The art of computer systems performance analysis : Techniques for experimental design, measurement, simulation, and modeling*, John Wiley & Sons, New York, 1991.

[44] JANSON S., « AKL—a multiparadigm programming language », PhD Thesis, Uppsala University and SICS, 1994.

[45] JANSON S. & HARIDI S., « Programming paradigms of the Andorra Kernel Language », in *International Symposium on Logic Programming*, 1991, p. 167-183.

[46] JANSON S., MONTELIUS J. & HARIDI S., « Ports for objects in concurrent logic programs », in *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

[47] JONES R. & LINS R., *Garbage collection : Algorithms for automatic dynamic memory management*, John Wiley & Sons, New York, 1996.

[48] KÅGEDAL A., VAN ROY P. & DUMANT B., « Logical State Threads 0.1 », 1997, Available at `www.info.ucl.ac.be/people/PVR/implementation.html`.

[49] KAY A. C., « The early history of Smalltalk », *ACM SIGPLAN Notices* **28** (1993), no. 3, p. 69-95, Originally appeared in History of Programming Languages Conference (HOPL-II), 1993.

[50] KNUTH D. E., *The art of computer programming : Fundamental algorithms*, vol. 1, Addison-Wesley, Reading, MA, 1973.

[51] ———, « Structured programming with **go to** statements », *Computing Surveys* **6** (1974), no. 4, p. 261-301.

[52] KORNSTAEDT L., « Gump–a front-end generator for Oz », Tech. report, Mozart Consortium, 2003, Available at `www.mozart-oz.org`.

[53] KORNSTAEDT L. & DUCHIER D., « The Oz Programming Interface », Tech. report, Mozart Consortium, 2003, Available at `www.mozart-oz.org`.

[54] KOSARAJU S. R., « Analysis of structured programs », *Journal of Computer and System Sciences* **9** (1974), no. 3, p. 232-255.

[55] KOWALSKI R. A., « Algorithm = logic + control », **22** (1979), no. 7, p. 424-436.

[56] ———, *Logic for problem solving*, North-Holland, 1979.

[57] KURZWEIL R., *The singularity is near : When humans transcend biology*, Penguin Books, 2006.

[58] LAMPORT L., *LaTeX : A document preparation system,* 2nd edition, Addison-Wesley, 1994.

[59] LARMAN C. & BASILI V. R., « Iterative and incremental development : A brief history », *IEEE Computer* **36** (2003), no. 6, p. 47-56.

[60] LEVESON N. & TURNER C. S., « An investigation of the Therac-25 accidents », *IEEE Computer* **26** (1993), no. 7, p. 18-41.

[61] LISKOV B., « A history of CLU », 1992, Technical Report, Laboratory for Computer Science, MIT.

[62] MACLENNAN B. J., *Principles of programming languages,* 2nd edition, WB Saunders, Philadelphia, 1987.

[63] MCCARTHY J., *LISP 1.5 programmer's manual*, MIT Press, Cambridge, MA, 1962.

[64] MCCLOUD S., *Understanding comics : The invisible art*, Kitchen Sink Press, 1993.

[65] MEYER B., *Object-oriented software construction,* 2nd edition, Prentice Hall, Englewood Cliffs, NJ, 2000.

[66] MILLER G. A., « The magical number seven, plus or minus two : Some limits on our capacity for processing information », *The Psychological Review* **63** (1956), p. 81-97.

[67] MILLER M., STIEGLER M., CLOSE T., FRANTZ B., YEE K.-P., MORNINGSTAR C., SHAPIRO J. & HARDY N., « E : Open source distributed capabilities », 2001, Available at `www.erights.org`.

[68] MORRISON J. P., *Flow-based programming : A new approach to application development*, Van Nostrand Reinhold, New York, 1994.

[69] MOZART CONSORTIUM , « The Mozart Programming System, version 1.3.2 », 2006, Available at `www.mozart-oz.org`.

[70] NAUR P., BACKUS J. W., BAUER F. L., GREEN J., KATZ C., MCCARTHY J. L., PERLIS A. J., RUTISHAUSER H., SAMELSON K., VAUQUOIS B., WEGSTEIN J. H., VAN WIJNGAARDEN A. & WOODGER M., « Revised report on the algorithmic language ALGOL 60 », **6** (1963), no. 1, p. 1-17.

[71] NYGAARD K. & DAHL O. J., « The development of the SIMULA languages », p. 439-493, Academic Press, 1981.

[72] OKASAKI C., *Purely functional data structures*, Cambridge University Press, Cambridge, UK, 1998.

[73] PARNAS D. L., *Software fundamentals*, Addison-Wesley, 2001.

[74] PFLEEGER S. L., *Software engineering : The production of quality software,* 2nd edition, Macmillan, 1991.

[75] POOLEY R. J., *An introduction to programming in SIMULA*, Blackwell Scientific Publishers, 1987.

[76] PRESSMAN R. S., *Software engineering,* 6th edition, Addison-Wesley, 2000.

[77] RAYMOND E., *The cathedral and the bazaar : Musings on linux and open source by an accidental revolutionary*, O'Reilly & Associates, 2001.

[78] REYNOLDS J. C., « User-defined types and procedural data structures as complementary approaches to data abstraction », in *Programming Methodology, A Collection of Papers by Members of IFIP WG 2.3* (Gries D., ed.), Springer-Verlag, 1978, Originally published in New Directions in Algorithmic Languages, INRIA Rocquencourt, 1975., p. 309-317.

[79] RUMBAUGH J., JACOBSON I. & BOOCH G., *The Unified Modeling Language reference manual*, Addison-Wesley, 1999.

[80] SACKS O., *The man who mistook his wife for a hat and other clinical tales*, Harper & Row, Publishers, 1987.

[81] SCHULTE C., *Programming constraint services : High-level programming of standard and new constraint services*, Lecture Notes in Computer Science, vol. 2302, Springer-Verlag, 2002.

[82] SCHULTE C. & SMOLKA G., « Encapsulated search for higher-order concurrent constraint programming », in *1994 International Symposium on Logic Programming*, MIT Press, 1994, p. 505-520.

[83] SIEWIOREK D. P., BELL C. G. & NEWELL A., *Computer structures : Principles and examples*, McGraw-Hill, New York, 1982.

[84] SMOLKA G., « The definition of Kernel Oz », in *Constraints : Basics and Trends* (Podelski A., ed.), Lecture Notes in Computer Science, vol. 910, Springer-Verlag, Berlin, 1995, p. 251-292.

[85] ———, « The Oz programming model », in *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, Springer-Verlag, Berlin, 1995, p. 324-343.

[86] SPIESSENS A., COLLET R. & VAN ROY P., « Declarative laziness in a concurrent constraint language », in *2nd International Workshop on Multiparadigm Constraint Programming Languages (MultiCPL 2003)* (Kinsale, Ireland), 2003, Workshop held as part of CP2003.

[87] STEELE, JR. G. L., *Common Lisp : The language,* 2nd edition, Digital Press, Bedford, MA, 1990.

[88] STERLING L. & SHAPIRO E., *The art of Prolog : Advanced programming techniques*, Series in Logic Programming, MIT Press, Cambridge, MA, 1986.

[89] STROUSTRUP B., « A history of C++ », *ACM SIGPLAN Notices* **28** (1993), no. 3, p. 271-297, Originally appeared in History of Programming Languages Conference (HOPL-II), 1993.

[90] ———, *The C++ programming language,* 3rd edition, Addison-Wesley, 1997.

[91] SUCCI G. & MARCHESI M., *Extreme programming examined*, Addison-Wesley, 2001.

[92] SUN MICROSYSTEMS , *The Java series*, Sun Microsystems, Mountain View, CA, 1996, Available at `www.javasoft.com`.

[93] SZYPERSKI C., *Component software : Beyond object-oriented programming*, Addison-Wesley and ACM Press, 1999.

[94] VAN ROY P., « Can logic programming execute as fast as imperative programming ? », PhD Thesis, Computer Science Division, University of California at Berkeley, 1990, Technical Report UCB/CSD 90/600.

[95] VAN ROY P., BRAND P., DUCHIER D., HARIDI S., HENZ M. & SCHULTE C., « Logic programming in the context of multiparadigm programming : The Oz experience », *Theory and Practice of Logic Programming* **3** (2003), no. 6, p. 715-763.

[96] VAN ROY P. & DESPAIN A., « High-performance logic programming with the Aquarius Prolog compiler », *IEEE Computer* (1992), p. 54-68.

[97] VAN ROY P. & HARIDI S., *Concepts, techniques, and models of computer programming*, MIT Press, 2004.

[98] WILF H. S., *generatingfunctionology*, Academic Press, 1994.

[99] WINSKEL G., *The formal semantics of programming languages*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1993.

Index

Les numéros de page en gras renvoient à la définition du terme.

B

D

E

F

J

K

L

M

P

Q

R

S

T

U

V

W

X

051196 - (I) - (2) - OSB 80° - PUB - CDD

Achevé d'imprimer sur les presses de
SNEL Grafics sa
Z.I. des Hauts-Sarts - Zone 3
Rue Fond des Fourches 21 – B-4041 Vottem (Herstal)
Tél +32(0)4 344 65 60 - Fax +32(0)4 286 99 61
Juillet 2007 – 42477

Dépôt légal : septembre 2007

Imprimé en Belgique

SCIENCES SUP

Peter Van Roy
Seif Haridi

PROGRAMMATION

Concepts, techniques et modèles

Ce cours de programmation s'adresse à des étudiants qui ont déjà une première expérience de la programmation, qu'ils soient en licence d'informatique (niveaux L2 ou L3) ou en écoles d'ingénieurs.

Traduction partielle d'un **cours de référence** publié par MIT Press, cet ouvrage présente l'originalité d'expliquer les **concepts majeurs de la programmation** à l'aide d'une approche synthétique qui en fait ressortir l'unité sous-jacente. Il propose une sémantique simple et complète qui permet de comprendre tous ces concepts sans pour autant sacrifier la rigueur. Toutes les notions théoriques présentées sont illustrées par des **centaines d'extraits de code**. En outre, **51 énoncés d'exercices** fournissent au lecteur l'occasion de tester ses connaissances.

Un site gratuit avec des TP, des énoncés d'examens, des transparents, un lexique français/anglais et d'autres suppléments… complète utilement le livre.

Ce livre de cours est complété par un ***Labo interactif*** qui vous permettra de compiler et d'exécuter tous les exemples de code du livre et de nombreux autres, afin de comprendre par l'expérience comment fonctionnent ces programmes, les modifier, voire en écrire de nouveaux.

Ce ***Labo interactif*** est édité par ScienceActive. Il est vendu séparément et vous pouvez vous le procurer sur le site *www.scienceactive.com*.

PETER VAN ROY

est professeur à l'Université catholique de Louvain à Louvain-la-Neuve. Il a lui-même traduit et adapté son ouvrage de l'anglais au français.

SEIF HARIDI

est professeur au Royal Institute of Technology (Suède) et « chief scientist » au Swedish Institute of Computer Science.

MATHÉMATIQUES
PHYSIQUE
CHIMIE
SCIENCES DE L'INGÉNIEUR
INFORMATIQUE
SCIENCES DE LA VIE
SCIENCES DE LA TERRE

LICENCE MASTER DOCTORAT
1 2 3 4 5 6 7 8

9 782100 511969

6494124
ISBN 978-2-10-051196-9

www.dunod.com

DUNOD