mini de Manuel

Programmation fonctionnelle

Cours + exos corrigés

Éric Violard
Maître de conférences HDR à l'université de Strasbourg

DUNOD

Au terme de ce projet, je souhaite dédier ce livre à ma fille Julie et à son bonheur de lire et d'apprendre.

Je ne saurais assez dire merci à tous les membres de ma famille, aussi je pense particulièrement à ma compagne (Myriam), à son écoute et son aide précieuse, à mes parents (Lyne et Raymond) et leur implication dans mon parcours scolaire et universitaire, et à mon frère (Christophe) au sens artistique inné.

Je tiens également à remercier M. Jean-Luc Blanc des éditions DUNOD pour m'avoir fait confiance, ainsi que mon interlocutrice privilégiée Mme Carole Trochu et tous ceux et celles qui ont contribué à leur manière à la réalisation effective de ce manuel.

Enfin, je veux adresser un grand remerciement à tous mes étudiants pour la curiosité et l'enthousiasme qu'ils ont manifesté lors de mes enseignements et qui sont à l'origine de ma motivation pour écrire ce manuel.

© Dunod, Paris, 2014
ISBN 978-2-10-070385-2

Table des matières

Disponibles en ligne sur
www.dunod.com/contenus-complementaires/9782100703852

Avant-propos

La **programmation fonctionnelle** est un *paradigme* de programmation, c'est-à-dire un *mode de programmation* et aussi une démarche, indépendante de tout langage de programmation, permettant de construire un programme résolvant un problème donné. Ce mode de programmation est fondé sur une théorie mathématique éprouvée, celle des fonctions, du *lambda-calcul* et des *systèmes de type.* Les concepts présentés sont donc fondamentaux et peuvent resservir pour répondre à d'autres questions liées à la programmation (certification de programmes, sémantique des langages, compilation, parallélisme, etc.).

La programmation fonctionnelle est de plus en plus présente et facilitée dans les langages. Même les langages traditionnels sont étendus pour permettre le mode fonctionnel comme en témoigne l'introduction des *fonctions anonymes* (ou *lambda-expressions)* dans C++ (depuis la norme C++11) et Java 8. Les langages fonctionnels *purs* s'imposent de plus en plus comme une alternative sérieuse aux langages traditionnels non purs.

Les qualités de ce mode de programmation sont en effet indéniables. D'une part, ce mode offre un grand pouvoir expressif permettant au programmeur d'écrire des expressions concises, de se rapprocher des termes du problème, de définir et d'utiliser simplement des *types abstraits* comme les *listes* ou les *arbres.* D'autre part, ce mode autorise la réalisation d'outils qui facilitent l'activité de programmation : interpréteurs permettant l'exécution interactive, compilateurs qui gèrent automatiquement l'utilisation de la mémoire, contrôleurs de types qui trouvent et vérifient automatiquement le type des expressions. En comparaison du *mode impératif,* le prix à payer en termes de performances du code exécutable produit par le compilateur, s'il fut dans le passé un facteur limitant l'utilisation du mode fonctionnel, est aujourd'hui très faible pour la grande majorité des applications.

En résumé, ce mode est transcendant, et pour paraphraser le slogan du langage fonctionnel Haskell : « *Utiliser le mode fonctionnel vous fera le plus grand bien !* ».

CONSEILS DE LECTURE

Les concepts présentés dans ce manuel sont indépendants de tout langage et illustrés en utilisant les implémentations récentes de trois langages fonctionnels : *Scheme* (prononcer ['ski:m]), *OCaml* et

Haskell. Il n'est pas nécessaire d'apprendre complètement la syntaxe d'un ou de plusieurs de ces langages ! Un programmeur expérimenté ne fait jamais cela : il a acquis suffisamment de connaissances et de recul au sujet d'un mode de programmation pour pouvoir utiliser n'importe quel langage qui supporte ce mode, et trouver dans un manuel de référence (destiné à un programmeur expérimenté), la syntaxe correspondant au concept qu'il souhaite utiliser.

À ce propos, ce livre n'est en aucun cas un manuel de référence, et pour connaître les subtilités ou raccourcis syntaxiques, le lecteur devra consulter le manuel de référence du langage au fur et à mesure des connaissances acquises. Au contraire, ce livre a l'objectif premier de présenter les concepts de façon incrémentale. La syntaxe correspondante est donnée incidemment à titre indicatif et dans un ordre et un découpage différent de celui d'un manuel de référence, car les objectifs sont bien différents (par exemple, certaines constructions syntaxiques ont été omises volontairement, et inversement certains concepts ne se traduisent pas directement par de la syntaxe).

Au-delà de la syntaxe différente, chacun de ces langages possède des caractéristiques propres (notamment en termes d'exécution) et apporte un éclairage différent :

– *Scheme* est « très souple » pour le programmeur au sens où il n'interdit pas certains raccourcis qui peuvent cependant être dangereux pour la correction des programmes ;
– *OCaml* est un produit universitaire plus éducatif, mais beaucoup plus rigide ;
– *Haskell* est totalement *pur* avec un pouvoir d'expression accru.

Il est conseillé au lecteur de commencer par l'un des langages OCaml ou Haskell, pour tester et expérimenter les concepts, puis de revenir ensuite sur les raccourcis et la souplesse de Scheme, une fois le mode fonctionnel mieux assimilé.

RESSOURCES NUMÉRIQUES

Ce manuel est accompagné de compléments en ligne qui permettront au lecteur d'aller plus loin. Ces compléments sont accessibles depuis le site de Dunod sur la page web dédiée à ce livre ou sur *http://www.dunod.com/contenus-complementaires/9782100703852*. Ces ressources numériques contiennent :

– **deux annexes (B et C)** : l'annexe B montre comment un programme écrit dans un langage fonctionnel peut se traduire en une expression du *lambda-calcul*. L'annexe C montre comment les opérations

d'entrée/sortie (saisie au clavier et affichage à l'écran) peuvent être définies proprement dans un langage fonctionnel pur.

– **des programmes** écrits en *OCaml, Haskell* et *Scheme :* ces programmes sont des solutions aux exercices énoncés à la fin de chaque chapitre. Dans ce manuel, la solution de chacun des exercices est écrite en utilisant un seul des trois langages. Ce complément révèle l'écriture de la solution dans les deux autres langages.
– **une application web** (*http://tolambda.sourceforge.net/fr*) : cette petite application permet à l'utilisateur d'expérimenter le *lambda-calcul* et en particulier de vérifier les solutions aux exercices du dernier chapitre. À l'aide de cette application, l'utilisateur peut réduire automatiquement une lambda-expression en utilisant la stratégie de son choix (*AOR, NOR* ou leurs variantes) et aussi effectuer la traduction présentée dans l'annexe B. Elle a été conçue et réalisée par l'auteur de ce manuel dans le seul but de renforcer la compréhension du lambda-calcul (le programme source a été écrit en OCaml). Elle est libre de droits (licence MIT) : la seule condition de réutilisation est de citer l'auteur.

USAGES TYPOGRAPHIQUES

Un mot est mis en gras s'il est important et que sa définition est présente dans la page où il apparaît. Dans le contexte d'expressions écrites en utilisant un formalisme (par exemple un langage de programmation, le lambda-calcul ou le langage des mathématiques), la couleur rouge est parfois utilisée pour mettre en évidence une partie d'une expression (par exemple, une certaine occurrence de la variable a dans cette expression $((\lambda x.\lambda a.a)\ a)$ du lambda-calcul). Elle est utilisée systématiquement lorsqu'il s'agit d'expressions qui sont affichées à l'écran par un interpréteur ou un code exécutable (par exemple l'invite `Prelude>` de l'interpréteur Haskell). Une expression de type est colorée en rouge pour signaler qu'elle est écrite en utilisant la syntaxe d'un certain langage de programmation (OCaml ou Haskell) : si une expression de type est en noir, cela signifie qu'elle n'est pas écrite dans un langage en particulier et qu'elle représente toutes les expressions qu'il est possible d'écrire dans les différents langages de programmation pour exprimer le même type (par exemple, `List Float` exprime un type qui s'écrit `float list` en OCaml et `List Float` en Haskell). Ces conventions seront rappelées juste avant leur utilisation.

CHAPITRE 1

Introduction

PLAN

OBJECTIFS

- Savoir situer le mode fonctionnel par rapport aux autres modes de programmation.
- Découvrir sur quoi repose la programmation fonctionnelle.
- Connaître le modèle sous-jacent et les concepts de base.

1.1 QUELQUES TERMES COURANTS EN PROGRAMMATION

Les termes cités dans cette section sont considérés comme des prérequis. L'activité de programmation consiste à écrire un *programme* résolvant un *problème* donné. Généralement, la solution d'un problème est un ensemble de *résultats* qui dépendent d'un ensemble de *données.* Un programme est un texte qui décrit de manière formelle un *algorithme,* c'est-à-dire une suite d'étapes ou d'opérations permettant d'obtenir la solution du problème. Ces opérations sont réalisables (on dit plutôt exécutables) par le *processeur* d'un ordinateur. Ce texte peut être saisi, modifié et enregistré dans un *fichier* à l'aide d'un *éditeur* de texte. Un programme est écrit en respectant la *syntaxe* rigoureuse d'un *langage de programmation.* Il a une signification précise ou *sémantique,* qui est définie par le langage. Il n'est pas directement exécutable par l'ordinateur, car celui-ci ne sait interpréter qu'un *code binaire,* c'est-à-dire une suite de `0` ou `1`. Des outils logiciels (*compilateur* ou *interpréteur)* associés au langage permettent d'exécuter un programme : soit en réalisant sa traduction complète en un code exécutable par l'ordinateur (compilateur), soit, lorsque le langage le permet, en interprétant le texte au fur et à mesure de sa lecture (interpréteur).

Tous les termes informatiques utilisés dans ce manuel sont mis en italique. Pour la plupart, leur sens figure dans le contexte où ils apparaissent. Certains termes font référence à des notions qui ne sont pas plus développées dans ce manuel (car elles dépassent les objectifs fixés), mais sont intéressantes à connaître dans le cadre d'études en informatique (c'est le cas de certains termes utilisés dans la section suivante qui a pour but de permettre au lecteur de différencier ce qui est propre au mode fonctionnel de ce qui appartient aux autres modes).

1.2 MODÈLES

La programmation fonctionnelle est un **mode de programmation**, c'est-à-dire une façon de construire un programme résolvant un problème donné. Les principaux modes de programmation sont :

- le *mode logique*,
- le *mode fonctionnel*,
- le *mode impératif*,
- et le *mode orienté objet*.

Le **mode orienté** objet peut être placé à part dans la mesure où il adresse une problématique différente (et complémentaire). Ce mode a en effet l'objectif premier de faciliter le développement et la maintenance de grands *logiciels* complexes en permettant de gérer séparément des parties conceptuellement disjointes. Les autres modes ont eux l'objectif premier de définir et organiser les calculs à réaliser par l'ordinateur ou la plateforme d'exécution.

Une problématique différente

Une distinction similaire apparaît parfois lorsqu'on parle de programmation « in the large » ou « in the small » pour opposer la réalisation et la maintenance d'un programme par une équipe dont chaque membre ne connaît qu'une partie, et l'écriture d'un programme par un seul programmeur qui connaît la totalité du programme.

Lorsqu'on veut apprendre et utiliser un mode de programmation, il est important de bien séparer le mode de programmation, du langage de programmation qui n'est qu'une syntaxe pour faciliter l'utilisation d'un ou plusieurs modes de programmation. (Ce manuel a l'ambition d'apprendre au lecteur à utiliser le mode fonctionnel avec tout langage le permettant).

Un langage de programmation est donc qualifié respectivement de « logique », « fonctionnel », « impératif » ou « orienté objet », dès lors qu'il encourage le programmeur à utiliser respectivement le mode logique, fonctionnel, impératif ou orienté objet. En théorie, il est possible d'utiliser un mode de programmation avec un langage de programmation qui n'encourage pas ce mode. Par exemple, utiliser le mode orienté objet avec un langage qui ne l'est pas. Cependant, dans ce cas, le programmeur doit gérer « à la main » de nombreuses contraintes d'accès aux données qui sont gérées automatiquement par le compilateur d'un langage orienté objet.

La plupart des langages de programmation récents sont *multi-paradigmes,* c'est-à-dire qu'ils permettent d'utiliser plusieurs modes de programmation (et un grand nombre sont orientés objet puisque ce mode est complémentaire des autres). Par exemple, le langage OCaml est à la fois fonctionnel, impératif et orienté objet.

Nous nous intéressons ici particulièrement aux modes que nous qualifions de « fondamentaux », c'est-à-dire ceux axés sur les calculs. Chacun de ces modes peut être défini ou caractérisé par un ensemble de notions mathématiques qui en constitue le **modèle** et sur lequel le mode est basé. Un modèle est une façon de formaliser la relation entre les données et les résultats d'un problème en utilisant le langage des mathématiques. Une fois cette relation formalisée, on obtient un *énoncé* rigoureux et non ambigu. Lorsque l'énoncé a une certaine forme, il est associé à un certain *modus operandi* ou procédé de calcul permettant d'obtenir étape par étape les résultats à partir des données. Un tel énoncé peut être directement codé dans un langage de programmation qui supporte le mode de programmation.

Dans la suite, nous montrons pour chacun des modes de programmation quel est son modèle sous-jacent. Les notions sont illustrées par le problème du calcul du maximum de deux nombres. Nous en donnons un énoncé et une solution sous la forme d'un programme écrit dans un langage qui supporte le mode de programmation.

Mode logique

Le mode logique est naturel puisqu'il s'agit de définir la relation entre les données et les résultats et qu'en logique mathématique, un *prédicat* définit une telle relation. Son modèle est celui des formules logiques (plus précisément les formules du *calcul des prédicats du premier ordre).* Un énoncé est une formule logique qui définit un prédicat, et ce prédicat est la propriété qui relie les données aux résultats.

Exemple Un énoncé pour le maximum de deux nombres est par exemple :

$$(A \geq B \wedge M = \mathrm{A} \Rightarrow max(A,B,M)) \wedge$$
$$(B \geq A \wedge M = \mathrm{B} \Rightarrow max(A,B,M))$$

où *max*(*A, B, M*) est un prédicat à trois places signifiant que *M* est le plus grand parmi *A* et *B*. Une formule logique est ainsi construite en utilisant des *connecteurs logiques :* l'implication $\Rightarrow$, le « et » logique $\wedge$, le « ou » logique $\vee$ et le « non » logique $\neg$ (la formule $P \Rightarrow Q$ peut aussi s'écrire $Q \vee \neg P$).

Lorsque l'énoncé est une conjonction de *clauses de Horn,* c'est-à-dire de formules de la forme $P_1 \wedge P_2 \wedge \cdots \wedge P_n \Rightarrow$ Q, comme c'est le cas de l'exemple, alors il est associé à un procédé de calcul permettant d'obtenir le résultat à partir des données, et peut s'écrire en utilisant un langage logique comme Prolog.

Exemple Le texte qui suit est un programme écrit en Prolog, et correspond à l'énoncé précédent:

```
max(A,B,M) :- A >= B, M = A.
max(A,B,M) :- B >= A, M = B.
```

Ce programme comporte deux clauses de Horn terminées par un point. La virgule note le « et » logique et `:-` note l'implication (la notation `Q :- P` signifie $P \Rightarrow Q$, autrement dit « Q est vrai si P est vrai »).

L'exécution d'un programme consiste à faire la démonstration qu'un prédicat est vrai. Un prédicat dont la vérité est à démontrer est appelé *but,* et le procédé de calcul (appelé *chaînage arrière)* consiste à tenter de démontrer tous les prédicats permettant de déduire le but.

Exemple Un but est par exemple `max(2,7,M)` où `M` est une *variable*. Ce prédicat signifie que `M` est le plus grand parmi `2` et `7`, il n'est vrai que pour certaines valeurs de `M`. L'exécution correspondant à ce but consiste à démontrer que ce prédicat est vrai et permet d'obtenir toutes ces valeurs de `M`. Le procédé de calcul consiste à rechercher une clause de Horn dont la *conclusion* (le prédicat à gauche de `":-"`) correspond (on dit plutôt dans ce contexte s'*unifie*) avec le but à démontrer. Ici, la première clause convient, car la partie gauche est `max(A,B,M)` qui s'unifie avec `max(2,7,M)` à condition de substituer la variable `A` par `2` et la variable `B` par `7`. Cette unification, définie par ces substitutions, est propagée dans la partie droite de la clause : à la fin de cette première

étape, on obtient les prédicats `2 >= 7` et `M = 2` qui sont autant de buts à démontrer en procédant de la même façon. Lorsqu'un but est un prédicat élémentaire, par exemple `2 >= 7`, un test a lieu pour décider s'il est vrai ou faux. S'il est faux, alors il faut retourner en arrière et chercher une autre clause qui pourrait convenir. S'il est vrai, alors il faut démontrer les buts restants. L'exécution s'arrête lorsqu'il ne reste plus de but à démontrer (le but initial est vrai) ou qu'il n'y a plus de clause à essayer (le but initial est faux). Voici ci-dessous les étapes de l'exécution du programme précédent avec un interpréteur du langage Prolog. Les affichages de l'interpréteur sont en rouge et les étapes du calcul sont numérotées de (1) à (5) :

```
| ?- max(2,7,M).
2 >= 7, M = 2          (1)
2 >= 7 ? faux          (2)
7 >= 2, M = 7          (3)
7 >= 2 ? vrai          (4)
M = 7 ? vrai           (5)
M = 7
yes
```

À la fin de l'étape (2), il y a un retour arrière et la recherche d'une autre clause. À l'étape (5), le prédicat élémentaire `M = 7` est vérifié en unifiant la variable `M` à la valeur `7`. L'exécution se termine donc avec l'affichage de `M = 7` et la réponse `yes` pour signifier que le but a été démontré.

Mode fonctionnel

Dans ce mode, la relation entre les données et les résultats est restreinte au cas particulier d'une fonction. Le modèle est donc celui bien connu des fonctions en mathématiques. Un énoncé est la définition d'une fonction qui à un *n-uplet* de données associe un *n-uplet* de résultats.

Exemple Un énoncé pour le maximum de deux nombres est par exemple :

$$\begin{array}{rcl} max : E \times E & \to & E \\ (a,b) & \mapsto & max(a,b) = \begin{cases} a, \text{ si } a \geq b \\ b, \text{ sinon} \end{cases} \end{array}$$

où E est l'ensemble des nombres (ou n'importe quel ensemble totalement ordonné), $E \times E$ est le *domaine* de la fonction, et a et b sont des variables qui désignent n'importe quels éléments de E.

Un tel énoncé peut être associé à un procédé de calcul permettant d'obtenir les résultats à partir des données. Ce procédé termine seulement si la définition (éventuellement récursive) de la fonction est correcte au sens où l'image par la fonction est bien définie pour tous les éléments de son *domaine*. L'ensemble des fonctions que l'on peut énoncer de cette façon est l'ensemble des *fonctions calculables* au sens des théories de la *calculabilité*. On peut écrire un tel énoncé dans un langage fonctionnel par exemple OCaml.

Exemple Le texte ci-dessous est un programme OCaml issu de l'énoncé précédent :

```
let max(a,b) = if a >= b then a else b;;
```

Ce programme est la définition d'une fonction `max` dont l'argument est un couple. Le mot-clé `let` introduit une définition et toutes les phrases OCaml se terminent par `;;` . La fonction est définie au moyen d'une équation dont le membre gauche, c.-à-d. `max(a,b)`, note l'image par la fonction du couple `(a,b)` où `a` et `b` sont des variables, et le membre droit, c.-à-d. `if a >= b then a else b`, est une expression appelée *alternative,* égale à `a` si la valeur de `a` est plus grande que celle de `b`, et égale à `b` sinon (ce programme exprime implicitement que les valeurs de `a` et `b` appartiennent à un ensemble ordonné).

L'exécution d'un programme consiste essentiellement à calculer formellement le résultat de l'application d'une fonction. Le procédé de calcul, appelé **évaluation**, consiste à réécrire l'expression du résultat en utilisant des règles (dites de *réécriture*) qui substituent une partie de l'expression par une autre expression de même valeur, comme on le ferait en mathématiques pour calculer la valeur d'une expression arithmétique. Ces règles sont celles du *lambda-calcul,* théorie sur laquelle sont fondés tous les langages fonctionnels et qui fait l'objet du chapitre 7 de ce manuel. Le chapitre 7, entièrement dédié à cette théorie, décrit donc plus en détail le comportement opérationnel d'un programme.

Exemple Pour exécuter le programme précédent, on peut par exemple appliquer la fonction `max` au couple `(2,7)`, c.-à-d. demander à un interpréteur d'évaluer l'expression `max(2,7)` qui note l'image de ce couple par la fonction. L'évaluation de cette expression implique de substituer `max` par sa définition et de remplacer les variables `a` et `b` respectivement par `2` et `7`. Elle consiste en l'application, dans un certain ordre, des règles de réécriture du lambda-calcul. À l'issue de ces substitutions, on obtient l'expression `if 2 >= 7 then 2 else 7`

qui est une alternative. L'évaluation de cette alternative, consiste à évaluer d'abord la *condition* `2 >= 7`, c'est-à-dire remplacer cette expression par `false` selon une règle prédéfinie pour l'*opérateur* `>=`. On obtient l'expression `if false then 2 else 7` qui est ensuite remplacée par `7` selon une autre règle prédéfinie pour l'alternative. En résumé, les étapes de l'exécution sont les suivantes :

```
# max(2,7);;
= if 2 >= 7 then 2 else 7      (1)
= if false then 2 else 7       (2)
= 7                            (3)
- : int = 7
```

Chaque étape peut contenir plusieurs applications de règles du lambda-calcul. L'interpréteur OCaml affiche le résultat et il indique que la valeur obtenue représente un entier (de *type* `int`).

Mode impératif

Ce mode de programmation est atypique et son modèle, plus complexe. On modélise d'abord la mémoire de l'ordinateur ou plus exactement un *état mémoire :* un état mémoire est une fonction qui à tout nom de variable (**identificateur**) associe une valeur. Le modèle est celui des fonctions entre états mémoire (également appelées *fonctions de transition).* Un énoncé est la définition d'une fonction de transition qui à un état mémoire « initial » associe un état mémoire « final ». Une donnée du problème est la valeur associée à un identificateur dans l'état initial, et un résultat est la valeur associée à un identificateur dans l'état mémoire final. La relation entre données et résultats est ainsi établie par une fonction de transition.

Exemple Un énoncé pour le maximum de deux nombres est par exemple la fonction de transition suivante qui à un état mémoire s, associe un état mémoire s' :

$$max : S \to S$$
$$s \mapsto s' \text{ tel que } s'(\mathtt{m}) = \begin{cases} s(\mathtt{a}), & \text{si } s(\mathtt{a}) \geq s(\mathtt{b}) \\ s(\mathtt{b}), & \text{sinon.} \end{cases}$$

où S est l'ensemble des états mémoire, et `a` et `b` et `m` sont des identificateurs de variable, $s(\mathtt{a})$ (respectivement $s(\mathtt{b})$) note l'image de `a` (respectivement `b`) par la fonction s, c'est-à-dire la valeur associée à `a` (respectivement `b`) dans l'état mémoire s. Dans l'état initial s, les « variables » (au sens impératif du terme) `a` et `b` ont les valeurs des données et dans l'état final s', la variable `m` a la valeur du résultat.

Un tel énoncé peut s'écrire comme une composition (on dit *séquence* dans ce contexte) de fonctions de transition élémentaires appelées *affectations.* Une affectation modifie un état mémoire en associant une autre valeur à un identificateur. Lorsque l'énoncé prend cette forme, il est associé à un procédé de calcul, et peut s'écrire dans un langage impératif comme C.

Exemple Voici un programme C pour le maximum de deux nombres :

```
#include <stdio.h>

main() {
   int a,b,m;
   scanf("%d %d",&a,&b);
   m = a;
   if(b > m) m = b;
   printf("%d\n",m);
}
```

Les éléments en rouge concernent les calculs, les autres éléments concernent la saisie des données au clavier et l'affichage du résultat à l'écran. Le point-virgule note la séquence de plusieurs fonctions de transition. Ce programme note une fonction de transition qui est la séquence d'une affectation notée `m = a` qui affecte à `m`, la valeur associée à `a`, et d'une *conditionnelle* notée `if(b > m) m = b`, qui change un état mémoire (en affectant à `m`, la valeur associée à `b`) si dans cet état mémoire la valeur de `b` est plus grande que celle de `m`, et laisse l'état mémoire inchangé sinon.

L'exécution d'un programme consiste à calculer l'état mémoire final étant donné un état mémoire initial. Le procédé de calcul est d'appliquer successivement chaque fonction de transition élémentaire dans l'ordre de la séquence.

Exemple Le déroulement de l'exécution du programme précédent (plus exactement du code exécutable obtenu en utilisant un compilateur) est résumé ci-dessous. On note s_0, s_1 et s_2, les états mémoires successifs. L'état mémoire s_0 est l'état initial dans lequel les variables `a` et `b` ont respectivement les valeurs `2` et `7` qui sont les données du problème, et s_2 est l'état mémoire final dans lequel la variable `m` a la valeur du résultat :

`2 7` (saisie des données)
s_0 : `a` $\mapsto$ `2` `b` $\mapsto$ `7` `m` $\mapsto$ `0` (1)

s_1 : a ↦ 2 b ↦ 7 m ↦ 2 = s_0(a) (2)

s_2 : a ↦ 2 b ↦ 7 m ↦ 7 = (s_1(b) si 7>2, s_1(m) sinon) (3)

7 (affichage du résultat)

À l'étape (1) (dans l'état mémoire initial s_0), la valeur de `m` est quelconque et indéfinie (on suppose ici que cette valeur est `0`). La première transition (l'affectation `m = a`) mène à l'état intermédiaire s_1 dans lequel la variable `m` a la valeur qu'avait `a` dans l'état précédent (s_0) c.-à-d. la valeur `2`. À l'étape (3), un test est effectué pour déterminer la valeur de `m` dans l'état final et cette valeur est celle du résultat.

1.3 SIGNIFICATIONS DU MOT « VARIABLE »

Nous venons de voir que tous les modes de programmation fondamentaux utilisent des variables, mais il est important de souligner que le sens du mot « variable » diffère :

- En programmation logique et fonctionnelle, une variable a le même sens qu'en mathématiques : une variable est une inconnue dans une équation. Par exemple, la variable x dans l'équation $2x - 6 = 0$. Résoudre l'équation en donne une définition explicite : $2x - 6 = 0 \Leftrightarrow x = 3$.
- En programmation impérative, une variable est l'expression abstraite d'une *adresse* en mémoire. Une définition est appelée « affectation » et signifie le rangement d'une valeur à cette adresse. Par exemple, on peut écrire `a = 1 ; a = 2` ou `x = x + 1`, ce qui n'a pas de sens (ou pas de solution) en mathématiques.

Cette différence est fondamentale et a des conséquences très importantes lorsqu'il s'agit de concevoir ou transformer un programme. La notion de variable au sens impératif est en effet très dangereuse pour raisonner sur les programmes : elle interdit, par exemple, toute substitution d'une variable par sa définition.

Exemple Considérons la substitution de la variable `x` par sa définition dans le programme impératif ci-dessous à gauche (écrit en utilisant la syntaxe du langage C) :

```
t = x + 1;
x = t;
y = t;
```

```
t = x + 1;
x = x + 1;
y = x + 1;
```

Si l'on substitue dans ce programme, la variable `t` par sa définition c.-à-d. l'expression `x + 1`, on obtient le programme à droite, qui

n'est pas équivalent : par exemple, si dans l'état initial, la variable `x` a pour valeur `1`, alors dans l'état final, la valeur de `y` sera `2` avec le programme à gauche, et `3` avec le programme à droite.

1.4 ÉLÉMENTS PURS OU IMPURS

L'utilisation de variables au sens mathématique du terme facilite la transformation ou l'amélioration d'un programme. Le programmeur peut s'appuyer sur des principes mathématiques bien connus, par exemple la substitution d'une variable par sa définition (ce qui est la base même des mathématiques) ou une propriété plus générale de certains langages de programmation appelée *propriété de transparence référentielle.* Cette propriété s'énonce informellement comme suit :

Propriété de transparence référentielle

« On obtient un programme équivalent si l'on remplace une expression par une expression de même valeur. »

Cette propriété est vraie en programmation logique et fonctionnelle, et fausse en programmation impérative (l'exemple précédent est un contre-exemple).

Définition : (Élément impur) Nous dirons qu'un élément d'un langage (une construction syntaxique autorisée par le langage et associée à une certaine sémantique) est *impur* s'il contredit d'une manière ou d'une autre la propriété de transparence référentielle.

Par exemple, l'affectation qui modifie la valeur associée à une variable est un élément impur. Un élément impur fait référence à un état global ou externe (par exemple un état mémoire) et modifie cet état sans que cela soit explicite dans un programme (on parle d'*effet de bord).* De tels éléments nuisent à la lisibilité des programmes et leur absence augmente leur réutilisabilité.

Certains langages fonctionnels ont des éléments impurs. C'est le cas des langages OCaml et Scheme par exemple. D'autres sont purs, c'est-à-dire totalement dépourvus d'éléments impurs, par exemple Haskell.
En conséquence, « programmer en utilisant le mode de programmation fonctionnelle » ne signifie pas simplement utiliser un langage fonctionnel, car il est possible de programmer en OCaml en adoptant le mode impératif (en utilisant