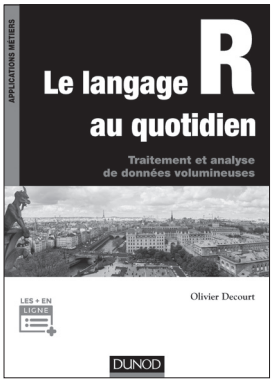
Algorithmique et programmation en Java

Informatique
Joëlle Delacroix et al.
480 pages
Dunod, 2017

Le langage R au quotidien - Traitement et analyse de données volumineuses
Olivier Decourt
288 pages
Dunod, 2018



Algorithmique et programmation en Java

Vincent Granet
Maître de conférences à Polytech'Sophia
de l'université Nice-Sophia-Antipolis

5e édition

DUNOD

Toutes les marques citées dans cet ouvrage
sont des marques déposées par leurs propriétaires respectifs.

11 rue Paul Bert, 92240 Malakoff
www.dunod.com

ISBN 978-2-10-078375-5

à Maud

Table des matières

Avant-propos

Pour cette cinquième édition d'*Algorithmique et Programmation en Java*, l'ensemble de l'ouvrage a été révisé. Il a été corrigé et actualisé pour tenir compte de l'évolution des technologies depuis quatre ans. De nouveaux algorithmes sont présentés, et de nouveaux exercices sont proposés. Des diagrammes de classes au format UML ont été introduits pour donner au lecteur une vision plus claire des relations entre les classes qui implémentent les types abstraits présentés dans la seconde partie de l'ouvrage.

Contrairement à la version 8 qui avait introduit le paradigme fonctionnel, la version 9 [1] de JAVA n'apporte pas de grands bouleversements au niveau du langage lui-même. La grande nouveauté est l'introduction de la notion de *module* et le découpage en modules de toute l'API. Cette notion définit un nouveau niveau d'encapsulation et sera évoquée au chapitre 12. Toutefois, dans le cadre de cet ouvrage, elle reste marginale et sera présentée succinctement.

L'informatique est une science mais aussi une technologie et un ensemble d'outils. Ces trois composantes ne doivent pas être confondues, et l'enseignement de l'informatique ne doit pas être réduit au seul apprentissage des logiciels. Ainsi, l'activité de programmation ne doit pas se confondre avec l'étude d'un langage de programmation particulier. Même si l'importance de ce dernier ne doit pas être sous-estimée, il demeure un simple outil de mise en œuvre de concepts algorithmiques et de programmation généraux et fondamentaux. L'objectif de cet ouvrage est d'enseigner au lecteur des méthodes et des outils de construction de programmes informatiques valides et fiables.

L'étude de l'algorithmique et de la programmation est un des piliers fondamentaux sur lesquels repose l'enseignement de l'informatique. Ce livre s'adresse principalement aux étudiants des cycles informatiques et élèves ingénieurs informaticiens, mais aussi à tous ceux qui ne se destinent pas à la carrière informatique mais qui seront certainement confrontés au développement de programmes informatiques au cours de leur scolarité ou dans leur vie professionnelle.

---

1. À noter qu'en attendant la version stable numéro 11 du langage, une version 10 intermédiaire propose l'inférence de type des variables locales.

Les seize premiers chapitres présentent les concepts de base de la programmation *impérative* en s'appuyant sur une *méthodologie objet*. Le chapitre 13 est quant à lui dédié à la *programmation fonctionnelle*. Tous mettent l'accent sur la notion de preuve des programmes grâce à la notion d'*affirmations* (antécédent, conséquent, invariant) dont la vérification formelle garantit la validité de programmes. Ils introduisent aussi la notion de *complexité* des algorithmes pour évaluer leur performance.

Les onze derniers chapitres étudient en détail les structures de données abstraites classiques (liste, graphe, arbre...) et de nombreux algorithmes fondamentaux (recherche, tri, jeux et stratégie...) que tout étudiant en informatique doit connaître et maîtriser. D'autre part, un chapitre est consacré aux interfaces graphiques et à leur programmation avec SWING.

La présentation des concepts de programmation cherche à être indépendante, autant que faire se peut, d'un langage de programmation particulier. Les algorithmes seront décrits dans une notation algorithmique épurée. Pour des raisons pédagogiques, il a toutefois bien fallu faire le choix d'un langage pour programmer les structures de données et les algorithmes présentés dans cet ouvrage. Ce choix s'est porté sur le langage à objets JAVA, non pas par effet de mode, mais plutôt pour les qualités de ce langage, malgré quelques défauts. Ses qualités sont en particulier sa relative simplicité pour la mise en œuvre des algorithmes, un large champ d'application et sa grande disponibilité sur des environnements variés. Ce dernier point est en effet important ; le lecteur doit pouvoir disposer facilement d'un compilateur et d'un interprète afin de résoudre les exercices proposés à la fin des chapitres. Enfin, JAVA est de plus en plus utilisé comme langage d'apprentissage de la programmation dans les universités. Pour les défauts, on peut par exemple regretter l'absence de l'héritage multiple, et la présence de constructions archaïques héritées du langage C. Ce livre n'est toutefois pas un ouvrage d'apprentissage du langage JAVA. Même si les éléments du langage nécessaires à la mise en œuvre des notions d'algorithmique et de programmation ont été introduits, ce livre n'enseignera pas au lecteur les finesses et les arcanes de JAVA, pas plus qu'il ne décrira les nombreuses classes de l'API. Le lecteur intéressé pourra se reporter aux très nombreux ouvrages qui décrivent le langage en détail, comme par exemple [GR15, Del17].

Les corrigés de la plupart des exercices, ainsi que des applets qui proposent une vision graphique de certains programmes présentés dans l'ouvrage sont accessibles sur le site web de l'auteur à l'adresse :

`http://www.polytech.unice.fr/~vg/index-algoprog.html`

Cet ouvrage doit beaucoup à de nombreuses personnes. Tout d'abord, aux auteurs des algorithmes et des techniques de programmation qu'il présente. Il n'est pas possible de les citer tous ici, mais les références à leurs principaux textes sont dans la bibliographie. À Olivier Lecarme et Jean-Claude Boussard, mes professeurs à l'université de Nice qui m'ont enseigné cette discipline au début des années 1980. Je tiens tout particulièrement à remercier ce dernier qui fut le tout premier lecteur attentif de cet ouvrage alors qu'il n'était encore qu'une ébauche, et qui m'a encouragé à poursuivre sa rédaction. À mes collègues et mes étudiants qui m'ont aidé et soutenu dans cette tâche ardue qu'est la rédaction d'un livre.

Enfin, je remercie toute l'équipe Dunod, et particulièrement Jean-Luc Blanc, pour leur aide précieuse et leurs conseils avisés qu'ils m'ont apportés pour la publication des cinq éditions de cet ouvrage.

Sophia Antipolis, avril 2018.

Chapitre **1**

Introduction

Les informaticiens, ou les simples usagers de l'outil informatique, utilisent des systèmes informatiques pour concevoir ou exécuter des programmes d'application. Nous considérerons qu'un environnement informatique est formé d'une part d'un ordinateur [1] et de ses équipements externes, que nous appellerons *environnement matériel*, et d'autre part d'un système d'exploitation avec ses programmes d'application, que nous appellerons *environnement logiciel*. Les programmes qui forment le logiciel réclament des méthodes pour les construire, des langages pour les rédiger et des outils pour les exécuter sur un ordinateur.

Dans ce chapitre, nous introduirons la terminologie et les notions de base des ordinateurs et de la programmation. Nous présenterons les notions d'environnement de développement et d'exécution d'un programme, nous expliquerons ce qu'est un langage de programmation et nous introduirons les méthodes de construction des programmes.

1.1 ENVIRONNEMENT MATÉRIEL

Un *automate* est un ensemble fini de composants physiques pouvant prendre des états identifiables et reproductibles en nombre fini, auquel est associé un ensemble de changements d'états non instantanés qu'il est possible de commander et d'enchaîner sans intervention humaine.

Un *ordinateur* est un automate *déterministe* à composants électroniques. Tous les ordinateurs, tout au moins les ordinateurs monoprocesseurs, sont construits, peu ou prou, sur le mo-

1. Même si, aujourd'hui, les équipements électroniques qui permettent l'exécution de logiciels deviennent très variés (tablettes numériques, smartphones, montres connectées, processeurs embarqués, etc), dans cet ouvrage nous conserverons le terme *ordinateur* pour désigner l'équipement qui sert à développer les programmes et applications présentés dans cet ouvrage.

dèle proposé en 1944 par le mathématicien américain d'origine hongroise VON NEUMANN. Un ordinateur est muni :

- D'une *mémoire*, dite *centrale* ou *principale*, qui contient deux sortes d'informations : d'une part l'information traitante, les *instructions*, et d'autre part l'information traitée, les *données* (les ordinateurs actuels ont tendance à posséder deux mémoires séparés, une pour les données et l'autre pour les instructions). Cette mémoire est formée d'un ensemble de cellules, ou *mots*, ayant chacune une *adresse unique*, et contenant des instructions ou des données. La représentation de l'information est faite grâce à une codification *binaire*, 0 ou 1. On appelle *longueur de mot*, caractéristique d'un ordinateur, le nombre d'éléments binaires, appelés *bits*, groupés dans une simple cellule. Les longueurs de mots usuelles des ordinateurs actuels (ou passés) sont, par exemple, 8, 16, 24, 32, 48 ou 64 bits. Cette mémoire possède une capacité *finie*, exprimée en gigaoctets (Go) ; un *octet* est un ensemble de 8 bits, un kilo-octet (Ko) est égal à 1 024 octets, un mégaoctet est égal à 1 024 Ko, un gigaoctet (Go) est égal à 1 024 Mo, et enfin un téraoctet (To) est égal à 1 024 Go. Actuellement, les tailles courantes des mémoires centrales des ordinateurs individuels varient entre 4 Go à 8 Go [2].
- D'une *unité centrale de traitement*, formée d'une *unité de commande* (UC) et d'une *unité arithmétique et logique* (UAL). L'unité de commande extrait de la mémoire centrale les instructions et les données sur lesquelles portent les instructions ; elle déclenche le traitement de ces données dans l'unité arithmétique et logique, et éventuellement range le résultat en mémoire centrale. L'unité arithmétique et logique effectue sur les données qu'elle reçoit les traitements commandés par l'unité de commande.
- De *registres*. Les registres sont des unités de mémorisation, en petit nombre (certains ordinateurs n'en ont pas), qui permettent à l'unité centrale de traitement de ranger de façon temporaire des données pendant les calculs. L'accès à ces registres est très rapide, beaucoup plus rapide que l'accès à une cellule de la mémoire centrale. Le rapport entre les temps d'accès à un registre et à la mémoire centrale est de l'ordre de 100.
- D'*unités d'échanges* reliées à des périphériques pour échanger de l'information avec le monde extérieur. L'unité de commande dirige les unités d'échange lorsqu'elle rencontre des instructions d'entrée-sortie.

Jusqu'au milieu des années 2000, les constructeurs étaient engagés dans une course à la vitesse avec des microprocesseurs toujours plus rapides. Toutefois, les limites de la physique actuelle ont été atteintes et depuis 2006 la tendance nouvelle est de placer plusieurs (le plus possible) microprocesseurs sur une même puce (le circuit-intégré). Ce sont par exemple les processeurs 64 bits Core i7 d'INTEL ou AMD Rysen d'AMD, qui possèdent de 4 à 16 cœurs permetttant l'exécution en parallèle de plusieurs programmes.

Les ordinateurs actuels possèdent aussi plusieurs niveaux de mémoire. Ils introduisent, entre le processeur et la mémoire centrale, des mémoires dites *caches* qui accélèrent l'accès aux données. Les mémoires caches peuvent être *primaires*, c'est-à-dire situées directement sur le processeur, ou *secondaires*, c'est-à-dire situées sur la carte mère. Certains ordinateurs introduisent même un troisième niveau de cache. En général, le rapport entre le temps d'accès entre les deux premiers niveaux de mémoire cache est d'environ 10 (le cache de niveau 1 est

2. Notez qu'avec 32 bits, l'espace adressage est de 4 Go mais qu'en général les systèmes d'exploitation ne permettent d'utiliser qu'un espace mémoire de taille inférieure. Les machines 64 bits actuelles, avec un système d'exploitation adapté, permettent des tailles de mémoire centrale supérieures à 4 Go.

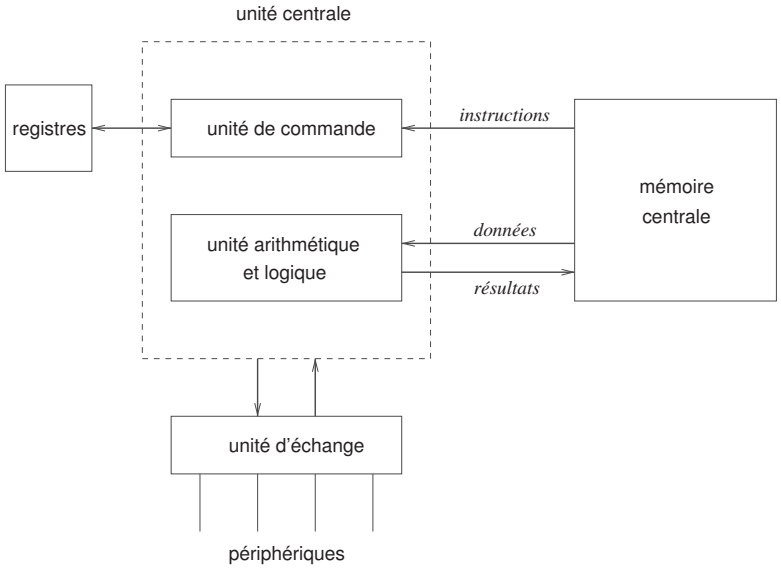


FIGURE 1.1 Structure générale d'un ordinateur.

le plus rapide et le plus petit). Le temps d'accès entre la mémoire cache secondaire et la mémoire centrale est lui aussi d'un rapport d'environ 10.

Les *équipements externes*, ou *périphériques*, sont un ensemble de composants permettant de relier l'ordinateur au monde extérieur, et notamment à ses utilisateurs humains. On peut distinguer :

- Les dispositifs qui servent pour la communication avec l'homme (clavier, écran (tactile), imprimantes, micros, haut-parleurs, scanners, etc.) et qui demandent une transcodification appropriée de l'information, par exemple sous forme de caractères alphanumériques.
- Les mémoires *secondaires*, qui permettent de conserver de l'information, impossible à garder dans la mémoire centrale de l'ordinateur faute de place, ou que l'on désire conserver pour une période plus ou moins longue après l'arrêt de l'ordinateur.
  Les mémoires secondaires sont de plus en plus les disques SSD (*Solid-State drive*, disque à semi-conducteurs), et, de moins en moins, les disques durs magnétiques les clés USB, les CD-ROM, les DVD, ou les Blu-ray. Par le passé, les bandes magnétiques ou les disquettes étaient des supports très utilisés. Actuellement les bandes magnétiques ne le sont quasiment plus, la fabrication des disquettes (supports de très petite capacité et peu fiables) a été arrêtée depuis de nombreuses années, et les ordinateurs vendus aujourd'hui ne sont plus pourvus de lecteur/graveur de DVD ou Blu-ray, comme c'était la cas il y encore quelques années.
  Aujourd'hui, la capacité des mémoires secondaires atteint des valeurs toujours plus importantes. Alors que certains DVD ont une capacité de 17 Go, qu'un disque Blu-ray atteint 50 Go, et que des clés USB de 64 ou 128 Go sont courantes, un seul disque SSD peut mémoriser jusqu'à 2 téraoctets, et un disque magnétique jusqu'à 10 téraoctets. Des systèmes permettent de regrouper plusieurs disques, vus comme un disque

unique, offrant une capacité de plusieurs centaines de téraoctets. Toutefois, l'accès aux informations sur les supports secondaires reste bien plus lent que celui aux informations placées en mémoire centrale. Pour les disques durs, le rapport est d'environ 10. De nous jours, avec le développement du « nuage » (*cloud computing*), la tendance est plutôt à la limitation des mémoires locales, en faveur de celles, délocalisées et bien plus vastes, proposées par les centres de données (*datacenters*) accessibles par le réseau Internet.

- Les dispositifs qui permettent l'échange d'informations sur un réseau. Pour relier l'ordinateur au réseau, il existe par exemple des connexions filaires comme celles de type *ethernet*, ou des connexions sans fil comme celles de type *WiFi* ou *Bluetooth*.

Le lecteur intéressé par l'architecture et l'organisation des ordinateurs pourra lire avec profit le livre d'A. TANENBAUM [Tan12] sur le sujet.

1.2 ENVIRONNEMENT LOGICIEL

L'ordinateur que fabrique le constructeur est une machine incomplète à laquelle il faut ajouter, pour la rendre utilisable, une quantité importante de programmes variés, qui constituent le *logiciel*.

En général, un ordinateur est livré avec un *système d'exploitation*. Un système d'exploitation est un programme, ou plutôt un ensemble de programmes, qui assurent la gestion des ressources, matérielles et logicielles, employées par le ou les utilisateurs. Un système d'exploitation a pour tâche la gestion et la conservation de l'information (gestion des processus et de la mémoire centrale, système de gestion de fichiers) ; il a pour rôle de créer l'environnement nécessaire à l'exécution d'un travail, et est chargé de répartir les ressources entre les usagers. Il propose aussi de nombreux protocoles de connexion pour relier l'ordinateur à un réseau. Entre l'utilisateur et l'ordinateur, le système d'exploitation propose une interface *textuelle* au moyen d'un *interprète de commandes* et une interface *graphique* au moyen d'un *gestionnaire de fenêtres*.

Les systèmes d'exploitation des premiers ordinateurs ne permettaient l'exécution que d'une seule tâche à la fois, selon un mode de fonctionnement appelé *traitement par lots* qui assurait l'enchaînement de l'exécution des programmes. À partir des années 1960, les systèmes d'exploitation ont cherché à exploiter au mieux les ressources des ordinateurs en permettant le *temps partagé*, pour offrir un accès simultané à plusieurs utilisateurs.

Jusqu'au début des années 1980, les systèmes d'exploitation étaient dits *propriétaires*. Les constructeurs fournissaient avec leurs machines un système d'exploitation spécifique, et le nombre de systèmes d'exploitation différents était important. Aujourd'hui, ce nombre a considérablement réduit, et seuls quelques-uns sont réellement utilisés dans le monde. Citons, par exemple, WINDOWS, MACOS ou LINUX pour les ordinateurs individuels, et UNIX pour les ordinateurs multi-utilisateurs. Avec l'apparition des tablettes et smartphones, de nouveaux systèmes d'exploitation spécifiquement dédiés à ces appareils sont apparus. Citons par exemple ANDROID ou IOS.

Avec l'augmentation de la puissance des ordinateurs personnels et l'avènement des réseaux mondiaux, les systèmes d'exploitation offrent, en plus des fonctions déjà citées, une quantité extraordinaire de services et d'outils aux utilisateurs. Ces systèmes d'exploitation

modernes mettent à la disposition des utilisateurs tout un ensemble d'applications (traitement de texte, tableurs, outils multimédias, navigateurs pour le web, outils de cummunication, jeux...) qui leur offrent un environnement de travail pré-construit, confortable et facile d'utilisation.

Le traitement de l'information est l'exécution par l'ordinateur d'une série finie de commandes préparées à l'avance, le *programme*, qui vise à calculer et rendre des résultats, généralement, en fonction de données entrées au début ou en cours d'exécution par l'intermédiaire d'interfaces textuelles ou graphiques. Les commandes qui forment le programme sont décrites au moyen d'un *langage*. Si ces commandes se suivent strictement dans le temps, et ne s'exécutent jamais simultanément, l'exécution est dite *séquentielle*, sinon elle est dite *parallèle*.

Chaque ordinateur possède un langage qui lui est propre, appelé *langage machine*. Le langage machine est un ensemble de commandes élémentaires représentées en code binaire qu'il est possible de faire exécuter par l'unité centrale de traitement d'un ordinateur donné. Le seul langage que comprend l'ordinateur est son langage machine.

Tout logiciel est écrit à l'aide d'un ou plusieurs *langages de programmation*. Un langage de programmation est un ensemble d'énoncés déterministes, qu'il est possible, pour un être humain, de rédiger selon les règles d'une grammaire donnée, et destinés à représenter les objets et les commandes pouvant entrer dans la constitution d'un programme. Ni le langage machine, trop éloigné des modes d'expressions humains, ni les langues naturelles écrites ou parlées, trop ambiguës, ne sont des langages de programmation.

La production de logiciel est une activité difficile et complexe, et les éditeurs font payer, parfois très cher, leur logiciel dont le code source n'est, en général, pas distribué. Toutefois, tous les logiciels ne sont pas payants. La communauté internationale des informaticiens produit depuis longtemps du logiciel gratuit (ce qui ne veut pas dire qu'il est de mauvaise qualité, bien au contraire) mis à la disposition de tous. Il existe aux États-Unis [3], une fondation, la FSF (*Free Software Foundation*), à l'initiative de R. STALLMAN, qui a pour but la promotion de la construction du logiciel *libre*, ainsi que celle de sa distribution. Libre ne veut pas dire nécessairement gratuit [4], bien que cela soit souvent le cas, mais indique que le texte source du logiciel est disponible. D'ailleurs, la FSF propose une licence, GNU GPL, afin de garantir que les logiciels sous cette licence soient *libres* d'être redistribués et modifiés par tous leurs utilisateurs.

Le système d'exploitation LINUX est disponible librement depuis de nombreuses années, et aujourd'hui certains éditeurs suivent ce courant en distribuant, comme Apple par exemple, gratuitement la dernière version de leur système d'exploitation *High Sierra* (toutefois sans le code source).

3. FSF France a pour but la promotion du logiciel libre en France (`http://fsffrance.org`).

4. La confusion provient du fait qu'en anglais le mot « free » possède les deux sens.

1.3 LES LANGAGES DE PROGRAMMATION

Nous venons de voir que chaque ordinateur possède un langage qui lui est propre : le langage machine, qui est en général totalement incompatible avec celui d'un ordinateur d'un autre modèle. Ainsi, un programme écrit dans le langage d'un ordinateur donné ne pourra être réutilisé sur un autre ordinateur.

Le langage *d'assemblage* est un codage alphanumérique du langage machine. Il est plus lisible que ce dernier et surtout permet un adressage relatif de la mémoire. Toutefois, comme le langage machine, le langage d'assemblage est lui aussi dépendant d'un ordinateur donné (voire d'une famille d'ordinateurs) et ne facilite pas le transport des programmes vers des machines dont l'architecture est différente. L'exécution d'un programme écrit en langage d'assemblage nécessite sa traduction préalable en langage machine par un programme spécial, l'*assembleur*.

Le texte qui suit [5], écrit en langage d'assemblage d'un Core i7 d'Intel, correspond à l'appel, de la fonction C `printf("Bonjour\n")` qui écrit *Bonjour* sur la sortie standard (*e.g.* l'écran) depuis la fonction `main`.

```
.LC0:
        .string "Bonjour"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
```

Le langage d'assemblage, comme le langage machine, est d'un niveau très élémentaire (une suite linéaire de commandes et sans structure) et, comme le montre l'exemple précédent, guère lisible et compréhensible. Son utilisation par un être humain est alors difficile, fastidieuse et sujette à erreurs.

Ces défauts, entre autres, ont conduit à la conception des langages de programmation dits de *haut niveau*. Un langage de programmation de haut niveau offrira au programmeur des

5. Produit par le compilateur gcc.

moyens d'expression structurés proches des problèmes à résoudre et qui amélioreront la fiabilité des programmes. Pendant de nombreuses années, les ardents défenseurs de la programmation en langage d'assemblage avançaient le critère de son efficacité. Les optimiseurs de code ont balayé cet argument depuis longtemps, et les défauts de ces langages sont tels que leurs thuriféraires sont de plus en plus rares.

Si on ajoute à un ordinateur un langage de programmation, tout se passe comme si l'on disposait d'un nouvel ordinateur (une machine abstraite), dont le langage est maintenant adapté à l'être humain, aux problèmes qu'il veut résoudre et à la façon qu'il a de comprendre et de raisonner. De plus, cet ordinateur fictif pourra recouvrir des ordinateurs différents, si le langage de programmation peut être installé sur chacun d'eux. Ce dernier point est très important, puisqu'il signifie qu'un programme écrit dans un langage de haut niveau pourra être exploité (théoriquement) sans modification sur des ordinateurs différents.

La définition d'un langage de programmation recouvre trois aspects fondamentaux. Le premier, appelé *lexical*, définit les symboles (ou caractères) qui servent à la rédaction des programmes et les règles de formation des mots du langage. Par exemple, un entier décimal sera défini comme une suite de chiffres compris entre 0 et 9. Le second, appelé *syntaxique*, est l'ensemble des règles grammaticales qui organisent les mots en phrases. Par exemple, la phrase « $234 / 54$ », formée de deux entiers et d'un opérateur de division, suit la règle grammaticale qui décrit une expression. Le dernier aspect, appelé *sémantique*, étudie la signification des phrases. Il définit les règles qui donnent un sens aux phrases. Notez qu'une phrase peut être syntaxiquement valide, mais incorrecte du point de vue de sa sémantique (*e.g.* $234/0$, une division par zéro est invalide). L'ensemble des règles lexicales, syntaxiques et sémantiques définit un langage de programmation, et on dira qu'un programme appartient à un langage de programmation donné s'il vérifie cet ensemble de règles. La vérification de la conformité lexicale, syntaxique et sémantique d'un programme est assurée automatiquement par des analyseurs qui s'appuient, en général, sur des notations formelles qui décrivent sans ambiguïté l'ensemble des règles.

Comment exécuter un programme rédigé dans un langage de programmation de haut niveau sur un ordinateur qui, nous le savons, ne sait traiter que des programmes écrits dans son langage machine ? Voici deux grandes familles de méthodes qui permettent de résoudre ce problème :

- La première méthode consiste à *traduire* le programme, appelé *source*, écrit dans le langage de haut niveau, en un programme sémantiquement *équivalent* écrit dans le langage machine de l'ordinateur (voir la figure 1.2). Cette traduction est faite au moyen d'un logiciel spécialisé appelé *compilateur*. Un compilateur possède au moins quatre phases : trois phases d'analyse (lexicale, syntaxique et sémantique), et une phase de production de code machine. Bien sûr, le compilateur ne produit le code machine que si le programme source respecte les règles du langage, sinon il devra signaler les erreurs au moyen de messages précis. En général, le compilateur produit du code pour un seul type de machine, celui sur lequel il est installé. Notez que certains compilateurs, dits *multicibles*, produisent du code pour différentes familles d'ordinateurs.

- Nous avons vu qu'un langage de programmation définit un ordinateur fictif. La seconde méthode consiste à *simuler* le fonctionnement de cet ordinateur fictif sur l'ordinateur réel par *interprétation* des instructions du langage de programmation de haut niveau.

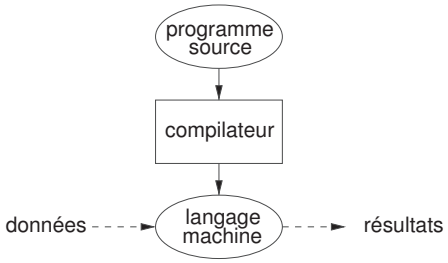


FIGURE 1.2 Traduction en langage machine.

Le logiciel qui effectue cette interprétation s'appelle un *interprète*. L'interprétation directe des instructions du langage est en général difficilement réalisable. Une première phase de traduction du langage de haut niveau vers un langage *intermédiaire* de plus bas niveau est d'abord effectuée. Remarquez que cette phase de traduction comporte les mêmes phases d'analyse qu'un compilateur. L'interprétation est alors faite sur le langage intermédiaire. C'est la technique d'implantation du langage JAVA (voir la figure 1.3), mais aussi de beaucoup d'autres langages. Un programme source JAVA est d'abord traduit en un programme objet écrit dans un langage intermédiaire, appelé JAVA pseudo-code (ou *byte-code*). Le programme objet est ensuite exécuté par la machine virtuelle JAVA, JVM (*Java Virtual Machine*).

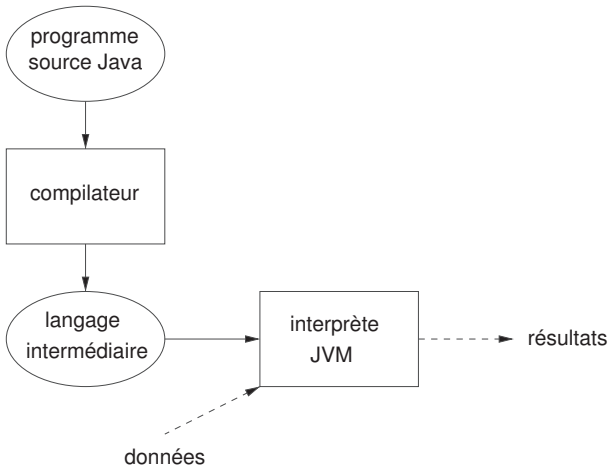


FIGURE 1.3 Traduction et interprétation d'un programme JAVA.

Ces deux méthodes, compilation et interprétation, ne sont pas incompatibles, et bien souvent pour un même langage les deux techniques sont mises en œuvre. L'intérêt de l'interprétation est d'assurer au langage, ainsi qu'aux programmes, une grande *portabilité*. Ils dépendent faiblement de leur environnement d'implantation et peu ou pas de modifications sont nécessaires à leur exécution dans un environnement différent. Son inconvénient majeur est que le temps d'exécution des programmes interprétés est notablement plus important que celui des programmes compilés.

➤ Bref historique

La conception des langages de programmation a souvent été influencée par un domaine d'application particulier, un type d'ordinateur disponible, ou les deux à la fois. Depuis près de soixante ans, plusieurs milliers de langages de programmation ont été conçus. Certains n'existent plus, d'autres ont eu un usage limité, et seule une minorité sont vraiment très utilisés [6]. Le but de ce paragraphe est de donner quelques repères importants dans l'histoire des langages de programmation « classiques ». Il n'est pas question de dresser ici un historique exhaustif. Le lecteur pourra se reporter avec intérêt aux ouvrages [Sam69, Wex81, Hor83] qui retracent les vingt-cinq premières années de cette histoire, à [ACM93] pour les quinze années qui suivirent, et à [M$^+$89] qui présente un panorama complet des langages à objets.

FORTRAN (*Formula Translator*) [Int57, ANS78] fut le premier traducteur en langage machine d'une notation algébrique pour écrire des formules mathématiques. Il fut conçu à IBM à partir de 1954 par J. BACKUS en collaboration avec d'autres chercheurs. Jusqu'à cette date, les programmes étaient écrits en langage machine ou d'assemblage, et l'importance de FORTRAN a été de faire la démonstration, face au scepticisme de certains, de l'efficacité de la traduction automatique d'une notation évoluée pour la rédaction de programmes de calcul numérique scientifique. À l'origine, FORTRAN n'est pas un langage et ses auteurs n'en imaginaient pas la conception. En revanche, ils ont inventé des techniques d'optimisation de code particulièrement efficaces.

LISP (*List Processor*) a été développé à partir de la fin de l'année 1958 par J. MCCARTHY au MIT (*Massachusetts Institute of Technology*) pour le traitement de données symboliques (*i.e.* non numériques) dans le domaine de l'intelligence artificielle. Il fut utilisé pour résoudre des problèmes d'intégration et de différenciation symboliques, de preuve de théorèmes, ou encore de géométrie et a servi au développement de modèles théoriques de l'informatique. La notation utilisée, appelée *S-expression*, est plus proche d'un langage d'assemblage d'une machine abstraite spécialisée dans la manipulation de liste (le type de donnée fondamental ; un programme LISP est lui-même une liste) que d'un véritable langage de programmation. Cette notation préfixée de type fonctionnel utilise les expressions conditionnelles et les fonctions récursives basées sur la notation λ (*lambda*) de A. CHURCH. Une notation, appelée *M-expression*, s'inspirant de FORTRAN et à traduire en *S-expression*, avait été conçue à la fin des années 1950 par J. MCCARTHY, mais n'a jamais été implémentée. Hormis LISP 2, un langage inspiré de ALGOL 60 (voir paragraphes suivants) développé et implémenté au milieu des années 1960, les nombreuses versions et variantes ultérieures de LISP seront basées sur la notation *S-expression*. LISP et ses successeurs font partie des langages dits *fonctionnels* (*cf.* chapitre 13). Il est à noter aussi que LISP est le premier à mettre en œuvre un système de récupération automatique de mémoire (*garbage-collector*).

La gestion est un autre domaine important de l'informatique. Au cours des années 1950 furent développés plusieurs langages de programmation spécialisés dans ce domaine. À partir de 1959, un groupe de travail comprenant des universitaires, mais surtout des industriels américains, sous l'égide du Département de la Défense des États-Unis (DOD), réfléchit à la conception d'un langage commun pour les applications de gestion. Le langage COBOL (*Common Business Oriented Language*) [Cob60] est le fruit de cette réflexion. Il a posé les premières bases de la structuration des données.

6. Voir les classements donnés par les sites `www.tiobe.com` ou `redmonk.com`.

On peut dire que les années 1950 correspondent à l'approche expérimentale de l'étude des concepts des langages de programmation. Il est notable que FORTRAN, LISP et COBOL, sous des formes qui ont bien évolué, sont encore largement utilisés aujourd'hui. Les années 1960 correspondent à l'approche mathématique de ces concepts, et le développement de ce qu'on appelle la théorie des langages. En particulier, beaucoup de notations formelles sont apparues pour décrire la sémantique des langages de programmation.

De tous les langages, ALGOL 60 (*Algorithmic Language*) [Nau60] est celui qui a eu le plus d'influence sur les autres. C'est le premier langage défini par un comité international (présidé par J. BACKUS et presque uniquement composé d'universitaires), le premier à séparer les aspects lexicaux et syntaxiques, à donner une définition syntaxique formelle, la Forme de BACKUS-NAUR [7], et le premier à soumettre la définition à l'ensemble de la communauté pour en permettre la révision avant de figer quoi que ce soit. De nombreux concepts, que l'on retrouvera dans la plupart des langages de programmation qui suivront, ont été définis pour la première fois dans ALGOL 60 (la structure de bloc, le concept de déclaration, le passage des paramètres, les procédures récursives, les tableaux dynamiques, les énoncés conditionnels et itératifs, le modèle de pile d'exécution, etc.). Pour toutes ces raisons, et malgré quelques lacunes mises en évidence par D. KNUTH [Knu67], ALGOL 60 est le langage qui fit le plus progresser l'informatique.

Dans ces années 1960, des tentatives de définition de langages *universels*, c'est-à-dire pouvant s'appliquer à tous les domaines, ont vu le jour. Les langages PL/I (*Programming Language One*) [ANS76] et ALGOL 68 reprennent toutes les « bonnes » caractéristiques de leurs aînés conçus dans les années 1950. PL/I cherche à combiner en un seul langage COBOL, LISP, FORTRAN (entre autres langages), alors qu'ALGOL 68 est le successeur *officiel* d'ALGOL 60. Ces langages, de par la trop grande complexité de leur définition, et par conséquence de leur utilisation, n'ont pas connu le succès attendu.

Lui aussi fortement inspiré par ALGOL 60, PASCAL [NAJN75, AFN82] est conçu par N. WIRTH en 1969. D'une grande simplicité conceptuelle, ce langage algorithmique a servi (et peut-être encore aujourd'hui) pendant de nombreuses années à l'enseignement de la programmation dans les universités.

Le langage C [KR88, ANS89] a été développé en 1972 par D. RITCHIE pour la récriture du système d'exploitation UNIX. Conçu à l'origine comme langage d'écriture de système, ce langage est utilisé pour la programmation de toutes sortes d'applications. Malgré de nombreux défauts, C est encore très utilisé aujourd'hui, sans doute pour des raisons d'efficacité du code produit et une certaine portabilité des programmes. Ce langage a été normalisé en 1989 par l'ANSI [8], puis en 1999 et 2011 par l'ISO [9] (normes C99 et C11).

Les années 1970 correspondent à l'approche « génie logiciel ». Devant le coût et la complexité toujours croissants des logiciels, il devient essentiel de développer de nouveaux langages puissants, ainsi qu'une méthodologie pour guider la construction, maîtriser la complexité, et assurer la fiabilité des programmes. ALPHARD [W+76] et CLU [L+77], deux langages expérimentaux, MODULA-2 [Wir85], ou encore ADA [ANS83] sont des exemples

7. À l'origine appelée Forme Normale de BACKUS.

8. *American National Standards Institute*, l'institut de normalisation des États-Unis.

9. *International Organization for Standardization*, organisme de normalisation représentant 164 pays dans le monde.

parmi d'autres de langages imposant une méthodologie dans la conception des programmes. Une des originalités du langage ADA est certainement son mode de définition. Il est le produit d'un appel d'offres international lancé en 1974 par le DOD pour unifier la programmation de ses systèmes embarqués. Suivirent de nombreuses années d'étude de conception pour déboucher sur une norme (ANSI, 1983), posée comme préalable à l'exploitation du langage.

Les langages des années 1980-1990, dans le domaine du génie logiciel, mettent en avant le concept de la *programmation objet*. Cette notion n'est pas nouvelle puisqu'elle date de la fin des années 1960 avec SIMULA [DN66], certainement le premier langage à objets. SMALLTALK [GR89], C++ [Str86] (issu de C), EIFFEL [Mey92], ou JAVA [GJS96, GR15], ou encore plus récemment C# [SG08] sont, parmi les très nombreux langages à objets, les plus connus.

JAVA connaît aujourd'hui un grand engouement, en particulier grâce au web et Internet. Ces quinze dernières années bien d'autres langages ont été conçus autour de cette technologie. Citons, par exemple, les langages de script (langages de commandes conçus pour être interprétés) JAVASCRIPT [Fla10] destiné à la programmation côté client, et PHP [Mac10] défini pour la programmation côté serveur HTTP.

Dans le domaine de l'intelligence artificielle, nous avons déjà cité LISP. Un autre langage, le langage *déclaratif* PROLOG (Programmation en Logique) [CKvC83], conçu dès 1972 par l'équipe marseillaise de A. COLMERAUER, a connu une grande notoriété dans les années 1980. PROLOG est issu de travaux sur le dialogue homme-machine en langage naturel et sur les démonstrateurs automatiques de théorèmes. Un programme PROLOG ne s'appuie plus sur un algorithme, mais sur la déclaration d'un ensemble de règles à partir desquelles les résultats pourront être déduits par unification et rétro-parcours (*backtracking*) à l'aide d'un évaluateur spécialisé.

Poursuivons cet historique par le langage ICON [GHK79]. Il est le dernier d'une famille de langages de manipulation de chaînes de caractères (SNOBOL 1 à 4 et SL5) conçus par R. GRISWOLD dès 1960 pour le traitement de données symboliques. Ces langages intègrent le mécanisme de confrontation de modèles (*pattern matching*), la notion de succès et d'échec de l'évaluation d'une expression, mais l'idée la plus originale introduite par ICON est celle du mécanisme de générateur et d'évaluation dirigée par le but. Un générateur est une expression qui peut fournir zéro ou plusieurs résultats, et l'évaluation dirigée par le but permet d'exploiter les séquences de résultats produites par les générateurs. Ces langages ont connu un vif succès et il existe aujourd'hui encore une grande activité autour du langage ICON.

Si l'idée de langages universels des années 1960 a été aujourd'hui abandonnée, plusieurs langages dits *multiparadigmes* ont vu le jour ces dernières années, alors que d'autres se sont étendus, comme par exemple les langages C++ et JAVA qui ont ajouté récemment le paradigme fonctionnel à celui d'objet.

Parmi les langages multiparadigmes conçus récemment, citons les langages PYTHON [Lut09], RUBY [FM08], SCALA [OSV08] et D [10]. Les deux premiers langages sont à typage dynamique (vérification de la cohérence des types de données à l'exécution) et incluent les paradigmes fonctionnel et objet. De plus, PYTHON intègre la notion de générateur similaire à celle d'ICON, et RUBY permet la manipulation des processus légers (threads) pour la programmation concurrente. SCALA, quant à lui, intègre les paradigmes objet et fonctionnel avec un typage statique fort. La mise en œuvre du langage permet la production de bytecode

10. https ://dlang.org

pour la machine virtuelle JVM, ce qui lui offre une grande compatibilité avec le langage JAVA. D a été conçu comme le successeur du langage C. Il propose les paradigmes objet, fonctionnel, la programmation par contrat, et la programmation concurrente.

Enfin, terminons avec le langage GO [11] développé par GOOGLE et apparu en 2009. Les langages multiparadigmes précédents, dans la mesure où ils renferment de nombreux concepts de nature souvent différente sont difficiles à maîtriser. Le langage GO prend le contre-pied. Inspiré des langages PASCAL et C, GO est un langage impératif à typage fort permettant la programmation concurrente. Il a été conçu pour être « *facile à comprendre et facile à adopter* », selon Rob Pike, l'un des auteurs du langage. L'objectif de simplicité du langage va de pair avec celle d'efficacité du code produit. La programmation concurrente exploite au mieux les processeurs multi-cœurs des ordinateurs actuels.

1.4 CONSTRUCTION DES PROGRAMMES

L'activité de programmation est difficile et complexe. Le but de tout programme est de calculer et fournir des résultats *valides* et *fiables*. Quelle que soit la taille des programmes, de quelques dizaines de lignes à plusieurs centaines de milliers, la conception des programmes exige des méthodes rigoureuses, si les objectifs de justesse et de fiabilité veulent être atteints.

D'une façon très générale, on peut dire qu'un programme effectue des *actions* sur des *objets*. Jusque dans les années 1960, la structuration des programmes n'était pas un souci majeur. C'est à partir des années 1970, face à des coûts de développement des logiciels croissants, que l'intérêt pour la structuration des programmes s'est accru. À cette époque, les méthodes de construction des programmes commençaient par structurer les actions. La structuration des objets venait ultérieurement. Depuis la fin des années 1980, le processus est inversé. Essentiellement pour des raisons de pérennité (relative) des objets par rapport à celle des actions : les programmes sont structurés d'*abord* autour des objets. Les choix de structuration des actions sont fixés par la suite.

Lorsque le choix des actions précède celui des objets, le problème à résoudre est décomposé, en termes d'actions, en sous-problèmes plus simples, eux-mêmes décomposés en d'autres sous-problèmes encore plus simples, jusqu'à obtenir des éléments directement programmables. Avec cette méthode de construction, souvent appelée *programmation descendante par raffinements successifs*, la représentation particulière des objets, sur lesquels portent les actions, est retardée le plus possible. L'analyse du problème à traiter se fait dans le sens descendant d'une arborescence, dont chaque nœud correspond à un sous-problème bien déterminé du programme à construire. Au niveau de la racine de l'arbre, on trouve le problème posé dans sa forme initiale. Au niveau des feuilles, correspondent des actions pouvant s'énoncer directement et sans ambiguïté dans le langage de programmation choisi. Sur une même branche, le passage du nœud père à ses fils correspond à un accroissement du niveau de détail avec lequel est décrite la partie correspondante. Notez que sur le plan horizontal, les différents sous-problèmes doivent avoir chacun une cohérence propre et donc minimiser leur nombre de relations.

11. https ://golang.org.

En revanche, lorsque le choix des objets précède celui des actions, la structure du programme est fondée sur les objets et sur leurs interactions. Le problème à résoudre est vu comme une modélisation (opérationnelle) d'un aspect du monde réel constitué d'objets. Cette vision est particulièrement évidente avec les logiciels graphiques et plus encore, de simulation. Les objets sont des composants qui contiennent des *attributs* (données) et des *méthodes* (actions) qui décrivent le comportement de l'objet. La communication entre objets se fait par *envoi de messages*, qui donne l'accès à un attribut ou qui lance une méthode.

Les critères de fiabilité et de validité ne sont pas les seuls à caractériser la qualité d'un programme. Il est fréquent qu'un programme soit modifié pour apporter de nouvelles fonctionnalités ou pour évoluer dans des environnements différents, ou soit dépecé pour fournir « des pièces détachées » à d'autres programmes. Ainsi de nouveaux critères de qualité, tels que la *robustesse*, l'*extensibilité*, la *compatibilité*, la *portabilité* ou la *réutilisabilité*, viennent s'ajouter aux précédents. Nous verrons que l'approche objet, bien plus que la méthode traditionnelle de décomposition fonctionnelle, permet de mieux respecter ces critères de qualité.

Les actions mises en jeu dans les deux méthodologies précédentes reposent sur la notion d'*algorithme* [12]. L'algorithme décrit, de façon non ambiguë, l'ordonnancement des actions à effectuer dans le temps pour spécifier une fonctionnalité à traiter de façon automatique. Il est dénoté à l'aide d'une notation *formelle*, qui peut être indépendante du langage utilisé pour le programmer.

La conception d'algorithme est une tâche difficile qui nécessite une grande réflexion. Notez que le travail requis pour l'exprimer dans une notation particulière, c'est-à-dire la programmation de l'algorithme dans un langage particulier, est réduit par comparaison à celui de sa conception. *La réflexion sur papier, stylo en main, sera le préalable à toute programmation sur ordinateur*.

Pour un même problème, il existe bien souvent plusieurs algorithmes qui conduisent à sa solution. Le choix du « meilleur » algorithme est alors généralement guidé par des critères d'*efficacité*. La *complexité* d'un algorithme est une mesure théorique de ses performances en fonction d'éléments caractéristiques de l'algorithme. Le mot *théorique* signifie en particulier que la mesure est indépendante de l'environnement matériel et logiciel. Nous verrons à la section 10.5 comment établir cette mesure.

Le travail principal dans la conception d'un programme résidera dans le choix des objets qui le structureront, la validation de leurs interactions et le choix et la vérification des algorithmes sous-jacents.

1.5 DÉMONSTRATION DE VALIDITÉ

Notre but est de construire des programmes valides, conformes à ce que l'on attend d'eux, c'est-à-dire qui respectent les *spécifications* qui décrivent le problème à résoudre.

12. Le mot *algorithme* ne vient pas, comme certains le pensent, du mot logarithme, mais doit son origine à un mathématicien persan du IX^{e} siècle, dont le nom abrégé était AL-KHOWÂRIZMÎ (de la ville de Khowârizm). Cette ville située dans l'Üzbekistān, s'appelle aujourd'hui Khiva. Notez toutefois que cette notion est bien plus ancienne. Les Babyloniens de l'Antiquité, les Égyptiens ou les Grecs avaient déjà formulé des règles pour résoudre des équations. Euclide (vers 300 av. J.-C.) conçut un algorithme permettant de trouver le *pgcd* de deux nombres.

Comment vérifier la validité d'un programme ? Une fois le programme écrit, on peut, par exemple, tester son exécution. Si la phase de test, c'est-à-dire la vérification expérimentale par l'exécution du programme sur des données particulières, est nécessaire, elle ne permet en aucun cas de démontrer la justesse à 100% du programme. En effet, il faudrait faire un test *exhaustif* sur l'ensemble des valeurs possibles des données. Ainsi, pour une simple addition de deux entiers codés sur 32 bits, soit 2^{32} valeurs possibles par entier, il faudrait tester $2^{32\times 2}$ opérations. Pour une microseconde par opération, il faudrait 9×10^9 années ! N. WIRTH résume cette idée dans [Wir75] par la formule suivante :

> « *L'expérimentation des programmes peut servir à montrer la présence d'erreurs, mais jamais à prouver leur absence.* »

La preuve [13] de la validité d'un programme ne pourra donc se faire que *formellement* de façon *analytique*, tout le long de la construction du programme et, évidemment, pas une fois que celui-ci est terminé.

La technique que nous utiliserons est basée sur des *assertions* qui décriront les propriétés des éléments (objets, actions) du programme. Par exemple, une assertion indiquera qu'en tel point du programme telle valeur entière doit être strictement positive, et donc qu'une valeur négative ou nulle provoquera une erreur.

Nous parlerons plus tard des assertions portant sur les objets. Celles pour décrire les propriétés des actions, c'est-à-dire leur sémantique, suivront l'*axiomatique* de C.A.R. HOARE [Hoa69]. L'assertion qui précède une action s'appelle l'*antécédent* ou *pré-condition* et celle qui la suit le *conséquent* ou *post-condition*.

Pour chaque action du programme, il sera possible, grâce à des *règles de déduction*, de déduire de façon *systématique* le conséquent à partir de l'antécédent. Notez qu'il est également possible de déduire l'antécédent à partir du conséquent. Ainsi pour une tâche particulière, formée par un enchaînement d'actions, nous pourrons démontrer son exactitude, c'est-à-dire le passage de l'antécédent initial jusqu'au conséquent final, par application des règles de déduction sur toutes les actions qui le composent.

Il est important de comprendre que les affirmations ne doivent pas être définies *a posteriori*, c'est-à-dire une fois le programme écrit, mais bien au contraire *a priori* puisqu'il s'agit de construire l'action en fonction de l'effet prévu.

Une action `A` avec son *antécédent* et son *conséquent* sera dénotée :

```
{antécédent}
A
{conséquent}
```

Les assertions doivent être le plus formelles possible, si l'on désire *prouver* la validité du programme. Elles s'apparentent d'ailleurs à la notion mathématique de *prédicat*. Toutefois, il sera nécessaire de trouver un compromis entre leur complexité et celle du programme. En d'autres termes, s'il est plus difficile de construire ces assertions que le programme lui-même, on peut se demander quel est leur intérêt ?

13. La preuve de programme est un domaine de recherche théorique ancien, mais toujours ouvert et très actif.

Certains langages de programmation, en fait un nombre réduit [14], intègrent des mécanismes de vérification de la validité des assertions spécifiées par les programmeurs. Dans ces langages, les assertions font donc parties intégrantes du programme. Elles sont contrôlées au fur et à mesure de l'exécution du programme, ce qui permet de détecter une situation d'erreur. En JAVA, une assertion est représentée par une expression *booléenne* introduite par l'énoncé `assert`. Le caractère booléen de l'assertion est toutefois assez réducteur car bien souvent les programmes doivent utiliser des assertions avec les quantificateurs de la logique du premier ordre que cet énoncé ne pourra exprimer. Des extensions au langage à l'aide d'annotations spéciales basées sur l'axiomatique de C.A.R. HOARE, comme [LC06], ont été proposées pour obtenir une spécification formelle des programmes JAVA.

Dans les autres langages, les assertions, même si elles ne sont pas traitées automatiquement par le système, devront être exprimées sous forme de *commentaires*. Ces commentaires serviront à l'auteur du programme, ou aux lecteurs, à se convaincre de la validité du programme.

14. Citons certains langages expérimentaux conçus dans les années 1970, tels que ALPHARD [M. 81], ou plus récemment EIFFEL.