# Chapitre 1

# Exécution d'un programme

Comme nous nous emploierons à diverses reprises dans cet ouvrage à traduire des programmes écrits dans un langage source vers un langage cible, ceci afin de démonter les mécanismes sous-jacents aux exécutions des programmes par les machines, nous allons dans ce chapitre et en préliminaire traduire des structures de contrôle classiques afin de permettre au lecteur de s'initier à ladite traduction.

Les langages auxquels nous allons nous intéresser sont les langages impératifs séquentiels. C'est-à-dire ceux qui décrivent les calculs sous la forme de suites d'instructions structurées qui chacune modifie de façon finie et bornée l'état de la mémoire. On les oppose communément aux langages fonctionnels qui eux, se focalisent sur l'aspect mathématique de l'application de fonctions et de leur évaluation, et dans lesquels les données ne sont que de pures valeurs. Dans les langages les plus utilisés et qui font partie de la famille qui nous intéresse on distingue généralement quatre types de contrôle de l'exécution :

**la séquence** qui ordonne les instructions à exécuter de sorte qu'elles soient exécutées une à une, les unes après les autres et dans l'ordre indiqué. La séquence est en général décrite simplement par une succession naturelle d'instructions considérées comme des phrases et leur ensemble formant un texte. C'est pourquoi on parle parfois du texte du code source.

**le branchement ou saut** qui se décline en **branchement inconditionnel** et **branchement conditionnel** est une rupture de la séquence permettant de sauter (sous réserve ou non qu'une condition soit établie) par-dessus (en avant ou en arrière) un paquet d'instructions de la séquence en cours. Le branchement inconditionnel (le fameux `goto`) est habituellement considéré comme dangereux (lire [10]) car il facilite la construction de **code dit spaghetti**. Un tel code possède une structure rappelant l'inextricable entremêlement d'un plat de spaghetti. Il existe pourtant des usages très raisonnés (comme le `continue` ou le `break`). Le branchement condition-

nel, plus connu sous le nom de `if`, est lui très utilisé. Il permet d'exécuter une séquence d'instructions sous réserve qu'une condition soit réalisée. Sans ce branchement l'expression de calculs un peu compliqués serait particulièrement difficile. Il en existe d'ailleurs une forme étendue dénommée **branchement multiple** ou **sélection** connu sous le nom de `switch-case` permettant à la manière d'un aiguillage de se brancher, en fonction d'une valeur calculée, en différents points d'une séquence.

**la boucle** se décline aussi en plusieurs versions. On trouve pêle-mêle : la boucle `for` (considérée comme la boucle de base), les boucles conditionnelles `while-do` et `repeat-until`. Toutes ces boucles ont pour usage de permettre la répétition contrôlée de l'exécution d'une séquence donnée. Le nombre de répétitions peut être connu à l'avance pour la boucle `for` ou indéterminé pour les boucles `while-do` et `repeat-until`. Quand bien même il est possible de déterminer à l'avance le nombre de répétitions, il est parfois plus facile d'écrire une boucle sous cette forme plutôt que sous celle d'une boucle `for`.

**l'appel (et le retour) de fonction (ou de procédure)** qui autorise à dérouter temporairement le flux d'exécution vers une séquence déterminée représentant un sous-calcul ; l'exécution revenant alors à l'instruction suivant le déroutement initial lorsque ledit sous-calcul prend fin. Ce type de contrôle particulier sera détaillé plus tard au chapitre 4.

Les ordinateurs sont équipés de nombreux dispositifs dont le principal est le **processeur**, **micro-processeur** ou encore **cœur** qui est chargé d'exécuter les instructions d'un programme-machine qui lui est fourni en s'appuyant sur la mémoire vive contenant les données nécessaires pour conduire le calcul.

Le programme-machine est généralement obtenu par la traduction d'un programme écrit en langage source (C, Java, etc.). Cette traduction appelée **compilation** est utile et pratique car les langages-machines ne sont pas adaptés aux êtres humains. Leur objet principal est de contrôler/piloter les organes de la machine de sorte que :

- les effets produits seront exactement ceux désirés par le concepteur du programme sans que celui n'ait besoin de s'attacher aux détails internes de la machine ;
- garantir qu'une certaine efficacité sera obtenue à l'exécution.

Ainsi on préfère donc écrire des programmes facilement lisibles pour les êtres humains et dont la traduction automatique garantira que la machine réalisera au bout du compte ce qui était voulu.

La caractéristique des programmes-machines est d'utiliser un langage très minimaliste et on y trouve en général quelques instructions très élémentaires : instructions arithmétiques de base (additions, multiplications), logiques (comparai-

sons), lecture ou écriture de la mémoire, etc. ainsi que quelques instructions de branchements conditionnels et inconditionnels. Pour s'en rendre compte, la lecture de n'importe quel ouvrage d'architecture des ordinateurs suffira (on trouvera des exemples dans la bibliographie à la fin de cet ouvrage).

La compilation consiste donc à traduire, entre autres, les structures de contrôle offertes comme facilités dans le langage source de haut-niveau en suites structurées d'instructions de bas-niveau (instructions de la machine) dont l'exécution produira exactement l'effet voulu et décrit par le manuel de référence du langage source.

C'est ce à quoi nous allons nous attacher dans ce chapitre. C'est-à-dire traduire des programmes exprimés dans un langage évolué en un programme dont l'exécution s'effectuera, dans les grandes lignes et par sa structure, comme celle d'un programme-machine. De façon générale, l'exécution d'un programme par une machine est ordinairement conduit comme dans le schéma suivant :

```
1  n° d'instruction courante = première à exécuter;
2  tant que ce n'est pas terminé {
3    récupérer l'instruction de n° courant
4    si l'instruction courante n'est pas un saut {
5      modifier l'état de la machine en conséquence
6      incrémenter le n° d'instruction courante
7    }
8    sinon
9      modifier le n° d'instruction courante
10 }
```

Ce schéma est celui de toute exécution séquentielle d'un programme par une machine de type von Neumann. C'est aussi le schéma que respecteront toutes nos traductions, car nous nous intéressons uniquement aux machines de ce type (ce que sont tous les ordinateurs modernes).

Comme on le verra au fur et à mesure, la ligne 5 sera représentée dans notre langage cible comme une suite finie de traductions d'instructions élémentaires. Ainsi la traduction prendra la forme générale d'un unique branchement multiple dans lequel chaque cas contiendra la traduction en instructions élémentaires de tout ou partie d'une instruction du langage source.

Ceci peut être représenté par le schéma (logigramme) suivant :

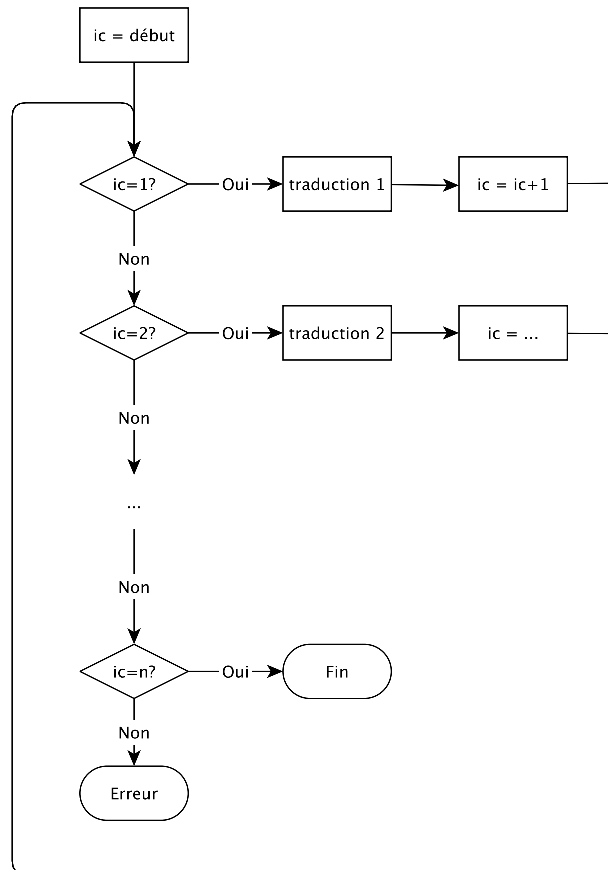


FIGURE 1.1 – Le schéma général de la traduction d'un programme

Dans ce schéma, on notera que les parties `traduction` correspondent chacune à la traduction d'une instruction de notre langage source en séquences d'instructions du langage cible en respectant l'ordre du programme source. Dans les `traduction`s, il est interdit de modifier en quoi que ce soit la valeur du **compteur ordinal** `ic`. Le compteur ordinal est une partie de la machinerie contenant le numéro de l'instruction en cours d'exécution.

Notre schéma comprend deux traductions types :

- l'une terminant par l'instruction `ic=ic+1` qui représente donc la mécanique permettant après chaque instruction de passer à la suivante immédiate dans l'ordre d'écriture du programme source ;

- l'autre terminant par une instruction modifiant le contenu du compteur ordinal de façon arbitraire `ic=...`. Une telle modification sert à briser la séquence ordonnée du programme comme on le verra.

Bien entendu dans le second cas le risque est d'essayer de passer à une instruction inexistante, le comportement de la machine est alors et à partir de là habituellement indéterminé mais en règle générale cela conduit assez rapidement à une erreur grave. Selon la gravité on dit que **le programme plante** ou que **la machine plante** et l'on a donc à faire avec **un plantage**. Les amateurs d'anglicismes utilisent de préférence le terme de *crash* : « Aaaah ! Mon programme a crashé ».

Pour des raisons pratiques, il nous arrivera de traduire les programmes en utilisant le schéma suivant :

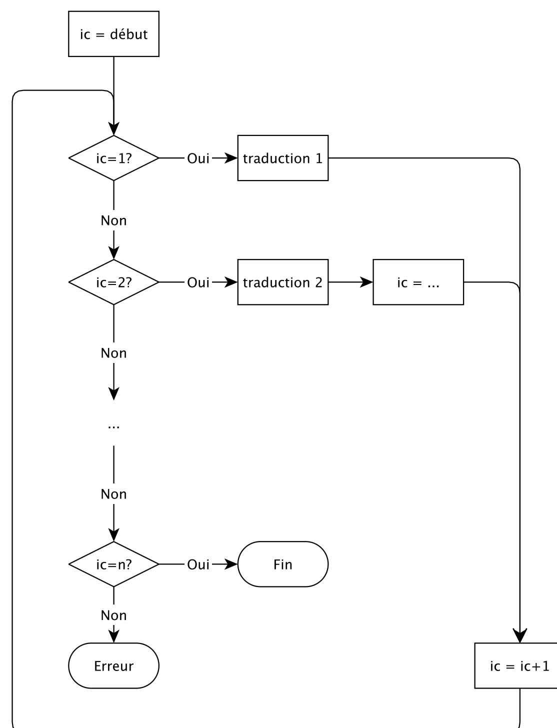


FIGURE 1.2 – Un autre schéma de traduction

Dans ce schéma, l'incrémentation du compteur ordinal est factorisée. C'est pourquoi si il est toujours interdit de modifier `ic` dans les blocs traduits, on peut toujours terminer un bloc par une modification arbitraire du compteur ordinal (attention cette modification doit prendre en compte qu'immédiatement après le compteur est incrémenté !).

## 1.1 Logigrammes

Il existe une notation graphique permettant d'exprimer des algorithmes ou des programmes tels que nous souhaitons les étudier. Cette notation permet d'exprimer différents types de contrôles qui vont nous intéresser.

Ce sont les **logigrammes**, **organigrammes** ou encore **algorigrammes**. La notation est définie par la norme ISO 5807 (voir [16]) qui date de 1985, mais il faut savoir que ces notations sont bien plus anciennes et qu'elles ont été utilisées très tôt, on en retrouve la trace dès les premières écritures de programmes et presque partout lorsqu'on tente de décrire un processus contenant des étapes un peu nombreuses et structurées par des choix. Les curieux peuvent se reporter à la lecture du *manuel technique d'utilisation des organigrammes* édité par IBM en 1969 (voir [9]).

Leur avantage est de faire apparaître visuellement la structure du contrôle d'un programme qui est par nature multidimensionnelle (au moins planaire). En effet dans l'écriture textualisée d'un programme c'est la linéarité qui prime puisqu'un un texte n'est qu'une succession linéaire de signes. C'est ce qui rend en partie compliquée la tâche de relecture des programmes, car l'essentiel des programmes est contenu dans la structure de son contrôle, c'est-à-dire ses possibles flux d'exécution.

Cette notation est parfois considérée comme désuète, à tort nous semble t-il, alors que justement elle exprime simplement et efficacement la structure du programme et qu'elle permet donc de s'affranchir (même faiblement) de la question du langage de programmation.

De telles notations sont par ailleurs courantes pour représenter des automates finis, des diagrammes de structure, des données, des collaborations, etc. comme dans la notation UML ou les diagrammes entité-association. Nous utiliserons ces diverses notations pour représenter lorsque nécessaire certaines structures de contrôle.

## 1.2 Traduction d'une séquence

La traduction que nous opérerons est simpliste, nous n'irons pas jusqu'à décomposer les instructions arithmétiques en suite d'instructions arithmétiques élémentaires afin d'éviter d'obtenir de trop longues traductions qui n'apporteraient rien quant à l'illustration du mécanisme général.

Une séquence est représentée, dans les logigrammes, par un diagramme comme celui-ci :

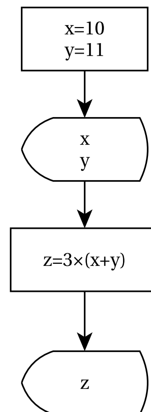


FIGURE 1.3 – Une séquence

La lecture de ce logigramme est relativement aisée et nous n'insisterons pas plus que nécessaire. On notera qu'il est possible d'inclure plusieurs opérations dans une boîte et que la forme des boîtes est signifiante.

La première boîte de la séquence spécifie un ensemble d'opérations à réaliser (dont l'ordre n'est pas spécifié). La seconde est une boîte spécifiant une sortie à l'écran (ici on affiche `x` et `y`). Nous rencontrerons par la suite d'autres boîtes élémentaires dont nous détaillerons le sens au moment voulu.

Si l'ordre des opérations à l'intérieur d'une boîte n'est pas spécifié, c'est qu'il s'agit en général d'opérations qui n'ont pas de relation de causalité, l'ordre importe peu. Si ce n'est pas le cas, *i.e.* si l'ordre importe, c'est l'enchaînement des boîtes qui l'exprime comme une **séquence**. Ce sont donc les flèches entrantes et sortantes qui imposent une causalité temporelle, le contenu d'une boîte doit être réalisé avant le contenu d'une boîte vers laquelle elle pointe

Une écriture en langage Java d'un programme exprimant le logigramme précédent peut être :

Prog. 1.1 – Une séquence d'instruction (`Sequence.java`)

```
1 public class Sequence {
2   public static int x, y, z;
3   public static void main(String []args) {
4     x = 10;
5     y = 11;
6     System.out.println("x="+x);
```

```
7       System.out.println("y="+y);
8       z = 3*(x+y);
9       System.out.println("z="+z);
10  } }
```

Et qui produit, après compilation et à l'exécution, le résultat suivant :

```
[Ciboulette:~] % java Sequence
x=10
y=11
z=63
[Ciboulette:~] %
```

Sa traduction vers notre langage cible consiste à attribuer un numéro à chaque « instruction » Java puis à intégrer la suite numérotée sous la forme du schéma précédent. Ce qui donne :

Prog. 1.2 – La traduction d'une séquence (`SequenceTraduite.java`)

```
1   public class SequenceTraduite {
2     // Mise en place des variables de contrôle
3     public static int instructionCourante = 4;
4     public static boolean fin = false;
5     public static void main(String []args) {
6       // Mémoire du programme à exécuter
7       int x=0, y=0, z=0;
8       // Exécution du programme
9       while (!fin) {
10        switch (instructionCourante) {
11        case 4:
12          x = 10;
13          instructionCourante++; break;
14        case 5:
15          y = 11;
16          instructionCourante++; break;
17        case 6:
18          System.out.println("x="+x);
19          instructionCourante++; break;
20        case 7:
21          System.out.println("y="+y);
22          instructionCourante++; break;
23        case 8:
24          z = 3*(x+y);
25          instructionCourante++; break;
26        case 9:
27          System.out.println("z="+z);
```