Chapitre 8

Qu'est-ce qu'un programme informatique ?

Comme nous l'avons vu, les algorithmes, qu'ils soient administratifs ou non, sont, de nos jours, exécutés, non pas à la main, mais sur des ordinateurs, grâce à des programmes informatiques. Ceci n'est pas, comme nous l'avons vu également, une fatalité : l'exécuter « à la main » permet de réfléchir à toutes les conséquences d'une décision donnée, ce qui n'est pas nécessairement souhaitable. Par ailleurs, les programmes informatiques sont beaucoup plus tangibles que les algorithmes. Donnons une idée de ce qu'est un *programme informatique* dans ce chapitre.

8.1 Introduction aux langages de programmation

8.1.1 Qu'est-ce qu'un ordinateur ?

On appelle **calculateur** toute machine d'aide aux calculs, que celle-ci soit simple comme les abaques de l'Antiquité, la machine arithmétique de Blaise PASCAL (qui permet d'additionner deux nombres entiers de huit chiffres) ou nettement plus complexe tel que le *marémomètre* de Lord KELVIN permettant de calculer les coefficients de marée. Les exemples que nous venons de donner sont des calculateurs *spécialisés*.

Un **ordinateur** (ou **calculateur universel**) est un calculateur permettant d'effectuer tout calcul pouvant être calculé sur une machine, quelle qu'elle soit. Un ordinateur est donc, par essence, une machine très versatile, s'adaptant à de très nombreuses situations. Par un retournement des choses, on appelle maintenant **calcul** tout type d'instances de ce que peut effectuer un ordinateur : il s'agit d'une généralisation des types de calculs (addition, multiplication, soustraction, division) que l'on apprend à l'école élémentaire, puis des divers calculs que certains voient ensuite (depuis l'extraction des racines carrées au calcul des *fonctions élémentaires* [fonctions trigonométriques, logarithmiques et autres]) ; au-delà de ces *calculs numériques*, on trouve des opérations, dites *non numériques*, tel que le classement de données et une grande partie du calcul des dérivées, des primitives, des développements limités, une petite partie de la résolution des équations différentielles (constituant le *calcul formel*).

Formellement, en mathématisant la situation, un calcul consiste à évaluer une expression telle que $f(a_1, \cdots, a_n)$ pour une certaine fonction f de $A_1 \times \cdots \times A_n$ dans B et des valeurs $a_1, \cdots, a_n$ qui lui sont appliquées, ces dernières étant appelés **données** en informatique.

8.1.2 Le principe d'un langage de programmation

Aux deux extrémités de la réalisation d'un logiciel, ou plus simplement programme, il y a un ordinateur O (qui exécute le programme) et un être humain H (qui le construit) [1].

En schématisant, l'ordinateur O exécute une suite d'instructions, qui effectuent des opérations sur la mémoire de l'ordinateur (des *cases dans lesquels on peut* ranger des valeurs, des nombres). Cette suite d'instructions est écrite dans un langage appelé *langage machine*.

Les objets sur lesquels travaille l'ordinateur sont des suites finies de 0 et de 1, la suite [00010101] par exemple. Même s'il est possible de programmer la machine en lui donnant des instructions sous cette forme-là, très rapidement les informaticiens ont développé des outils de communication avec la machine plus conviviaux et/ou plus lisibles par un être humain. De la même façon que les êtres humains communiquent entre eux grâce à une langue, les informaticiens ont développé des *langages* dits *de programmation*. Comme pour une langue naturelle, un langage de programmation est composé d'un vocabulaire et d'une grammaire. Ce vocabulaire et cette grammaire sont accompagnés d'une sémantique (en fait plusieurs sémantiques mais dont une est principale, appelée *sémantique opérationnelle*). Dans la suite nous allons étudier une sous-partie du langage ADA, non pas qu'il soit très utilisé mais il est en partie utilisé dans le code du programme APB fourni par le ministère. Nous exposerons suffisamment d'éléments de ce langage pour que le code fourni par le ministère soit compréhensible, ou tout du moins dont l'analyse faite de ce programme soit convaincante au regard de sa sémantique opérationnelle.

1. Il est en fait possible qu'un autre ordinateur produise le programme mais, pour simplifier, nous prendrons un être humain

8.1.2.1 Syntaxe d'ADA

Un langage de programmation est constitué de mots clés (ou mots réservés) qui ne peuvent être utilisés dans un autre contexte que celui pour lequel ils ont été définis à l'origine. Par ailleurs, on peut fabriquer de nouveaux mots dans les langages de programmation, qui permettent d'enrichir le vocabulaire du langage. Les mots clés ont un rôle très précis. Nous allons insister sur deux de ces rôles : donner un nom à une case mémoire dans laquelle on peut ranger des informations et donner un nom à un groupe de mots.

8.1.2.1.1 Les mots du langage. On liste (partiellement) une suite de mots écrits en utilisant les vingt-six caractères de l'alphabet latin (sans éléments diacritiques tels que les accents) ainsi que les chiffres (`0`, ..., `9`), certains symboles de ponctuation (`;`, `:`) et des symboles venant souvent du vocabulaire des mathématiques (`=`, `'`, `(`, `)`, `+`, `*`, `-`, `<`, `>`). Les mots réservés principaux sont : `LOOP`, `END`, `BEGIN`, `FOR`, `FUNCTION`, `IF`, `THEN`, `ELSE`, `RETURN`, `IN`, `OUT`, `EXIT`, `WHEN` et `NUMBER`.

On autorise la création de nouveaux mots, appelés *identificateurs*, commençant par une lettre de l'alphabet, pouvant contenir des lettres, des chiffres et quelques symboles (ici on se limite au symbole _), qui ne sont pas des mots clés, par exemple :

`login`, `x`, `nip`, `dummy2`, `mess_err`.

8.1.2.1.2 La grammaire du langage. On peut constituer des phrases avec ces mots (appelées 'expressions' et 'instructions' dans le cadre des langages de programmation).

Les expressions. Une expression est construite à partir de mots ou symboles qui eux même manipulent ou expriment une expression peu clair. Les informaticiens utilisent allègrement des constructions dites *récursives* (c'est-à-dire qui sont définies à partir d'elles-mêmes, en précisant cependant qu'il y a des cas initiaux). Par exemple une suite de chiffres, comme 2381, est une expression de base ; d'autre part :

- une suite de chiffres est une expression ;
- une suite de lettres et/ou de chiffres et/ou de symboles placée entre une paire du symbole « ' » est une expression ;
- un identificateur est une expression.

Les expressions composées sont définies de la façon suivante :

- s'il y a des symboles ou des mots qui peuvent se composer avec des expressions alors se sont des expressions à leur tour. peu clair

Exemple. Un exemple simple d'expression composée est : 2381+26. En effet, les deux suites de chiffres sont des expressions et une règle dit le symbole + se compose avec deux expressions pour donner une expression. On peut aussi construire l'expression `nip` + `nip`, car `nip` est un identificateur et que le symbole + construit une expression à partir de deux expressions, comme nous venons de le dire. De même si `write` est un identifiant qui se compose avec une expression entourée d'un couple de parenthèses, respectivement ouvrante et fermante, alors `write('toto est présent')` est une expression. Nous verrons au cours du développement de ce chapitre de nouveaux constructeurs d'expressions.

Les instructions Nous nous limiterons aux instructions suivantes :

- l'instruction d'*affectation* s'écrit à l'aide du symbole composé : = entouré, à sa gauche, d'un identificateur et, à sa droite, d'une expression. Par exemple les trois expressions « `x := 17` », « `nip := '01340202'` » et « `x` := 2 + 17 » sont des affectations ;

```
FUNCTION Inf(M, N : IN INTEGER) RETURN INTEGER
IS
 X : INTEGER := M;
 Y : INTEGER := N;
 R : INTEGER := 0;
BEGIN
 FOR I in 1..X LOOP
   R := R + 1;
   Y := Y - 1;
   EXIT WHEN Y = 0;
 END LOOP;
 RETURN R;
END Inf;
```

TABLE 8.1 – Un programme pour calculer le plus petit de deux entiers M et N

- l'instruction de *bloc d'instructions* s'écrit `BEGIN` $\mathtt{I}_1$; $\mathtt{I}_2$; ... $\mathtt{I}_n$; `END`, où les I_i sont des instructions ;
- l'*instruction conditionnelle* s'écrit `IF C THEN` $\mathtt{I}_1$; `ELSE` $\mathtt{I}_2$; `END IF`;
- les instructions d'*itération* s'écrivent d'au moins deux manières différentes :
  - l'*itération bornée* `FOR id IN` $\mathtt{E}_1..\mathtt{E}_2$ `LOOP I; END LOOP` où `id` est un identificateur, $\mathtt{E}_1$ et $\mathtt{E}_2$ des expressions et `I` une instruction ;
  - l'itération non bornée `LOOP` $\mathtt{I}_1$; ... $\mathtt{I}_n$; `EXIT WHEN C;` $\mathtt{I}_{n+1}$; ... $\mathtt{I}_m$; `END LOOP` où les $\mathtt{I}_i$ sont des instructions et `C` est une expression booléenne.

Une fois définie la *syntaxe* d'un langage de programmation, il faut se mettre d'accord sur le sens de l'exécution des instructions, ce qui l'objet de la *sémantique*. Il existe de nombreuses sortes d'ordinateurs : les ordinateurs portables, les ordinateurs de bureaux, les gros calculateurs, les tablettes et les smartphones. Pour chacun de ces types de machines, on peut avoir du matériel (processeur, mémoire) différent. On a aussi des systèmes d'exploitation différents (Windows, Linux et MacOS sont les principaux).

Il faut garantir que l'exécution d'un programme sur chacune de ces machine donne le même résultat. Pour cela, les informaticiens ont développé les notions de *sémantique opérationnelle* et de *sémantique dénotationnelle*. Illustrons ces notions sur les deux exemples suivants.

8.1.3 Exemples

Le code de ces deux exemples est donné par la table 9.1 et la table 9.2.

8.1.4 La sémantique dénotationnelle

La *sémantique dénotationnelle* est la plus simple à comprendre mais pas forcément la plus simple à construire. L'utilisateur s'attend à ce qu'un programme calcule **quelque chose**. Ce **quelque chose** est ce qui est appelé la *dénotation* du programme, en utilisant un mot introduit par le philosophe Gottlob FREGE ; par exemple, on s'attend à ce que le logiciel APB attribue une place dans une formation de l'enseignement supérieur à tous les titulaires du baccalauréat qui le souhaitent. Mais ceci n'est pas suffisant. On peut tenir compte, par exemple, des envies des candidats et des formations. On peut alors préciser que les entrées du programme sont :

```
PROCEDURE Fib(N : INTEGER) IS
 a : INTEGER := 1;
 b : INTEGER := 0;
 r : INTEGER := 0;
 i : INTEGER := 0;
BEGIN
 WHILE (i < N) LOOP
   r := a + b;
   b := a;
   a := r;
   i := i + 1;
 END LOOP;
 WRITE(r);
END Fib;
```

TABLE 8.2 – Un programme pour calculer le N^{eme}-terme de la suite de Fibonacci.

— pour chaque candidat, une liste ordonnée finie bornée de choix de formations ;
— pour chaque formation, une liste ordonnée (ou pas), finie (non bornée pour certaines) de candidats ;
— des données concernant les candidats (notes, appréciations...).

C'est la première étape de la définition de la sémantique dénotationnelle d'un programme : spécifier le genre d'entrées à fournir (une structure avec des propriétés) et spécifier le genre de sortie qu'on attend.

Cette notion restreinte de sémantique dénotationnelle est parfois appelée *spécification*[2].

La sémantique dénotationnelle a pour rôle de définir la fonction calculée par le programme (fonction en terme d'entrées-sorties).

Le premier programme donné en exemple calcule le minimum de deux valeurs données en paramètre : on note $[[Inf]] = min$ la fonction calculée par le programme. On peut écrire de nombreux programmes qui respectent cette sémantique.

8.1.5 La sémantique opérationelle

Pour comprendre l'exécution d'un programme, il faut pouvoir comprendre ce que l'exécution de chaque instruction fait sur la machine sur laquelle elle s'exécute. Comme on ne va pas décrire réellement la machine (il y en a beaucoup trop !), on développe un modèle de machine sur lequel on montre l'exécution de chacune des instructions.

8.1.5.1 Un modèle de machine

Nous nous plaçons à un certain niveau au-dessus de la machine et on décrit le modèle en donnant un ensemble de fonctions représentant des composants d'une vraie machine :

8.1.5.1.1 La mémoire La mémoire d'une machine est représentée par une fonction Mem : $\mathbb{N}_{mem} \to \mathbb{Z}$ qui à un entier positif associe un entier relatif. Dans la notation $Mem(i)$, on identifie

2. La sémantique dénotationnelle n'est pas toujours synonyme de spécification ; elle permet aussi de définir mathématiquement des comportements plus fins que la relation entre les entrées d'un programme et les sorties d'un programme [Col-, Sco-]

i comme étant la i-ème case de la mémoire, $Mem(i)$ désignant la valeur contenue dans la i-ème case. On a, par exemple : $Mem(100) = 14, Mem(101) = 19$ et $Mem(102) = 0$.

8.1.5.1.2 Le dictionnaire des identificateurs Pour des raisons de lisibilité, on introduit un dictionnaire $DictId$ qui permet de lier un identificateur à une case de la mémoire : $DictId : Identifiant \to \mathbb{N}_{mem}$. Les identifiants sont représentés par leur nom (ici une chaîne de caractères). Par exemple $DictId("X") = 101, DictId("Y") = 101$ et $DictId("R") = 102$.

8.1.5.1.3 Opérations de base Une machine comporte des opérations dites *de base*. Ces opérations sont au moins de deux types :

- les opérations permettant d'accéder à la mémoire de la machines [3] :
  - on peut ajouter une variable dans le dictionnaire : $DictId \oplus (c, a)$;
  - on peut obtenir le numéro d'une case non encore utilisée : $new(Mem)$, qui renvoie le numéro d'une case de Mem qui n'a pas encore été utilisée ;
  - on peut indiquer à la mémoire que l'on n'utilise plus une certaine case : $free(i)$ (la case i de la mémoire est libre) ;
  - on peut modifier la valeur de la case de la mémoire : $Mem \oplus (i, v)$;
- les opérations permettant d'effectuer des calculs sur des valeurs. On appelle ces opérations des **fonctions statiques**. Plus cet ensemble de fonctions est volumineux, plus on peut écrire des programmes de manière concise. **Remarque** si ces opérations de bases sont calculables (*i.e. peuvent être décrites par un programme*) alors le gain n'est qu'un gain sur la concision du programme Trop elliptique. Pour illustrer cela, on peut imaginer que nous ayons dans nos fonctions de base la fonction qui calcule le plus petit de deux entiers. On peut alors se passer d'écrire le programme 8.1 et utiliser cette fonction à la place.

  Quand on définit le modèle sur lequel on va réaliser l'exécution du programme, on associe une sémantique dénotationnelle à chacune des opérations de base. Par exemple, notre modèle d'exécution peut contenir la fonction de base *plus* qui a la sémantique suivante : $plus(u, v)$ a comme valeur la valeur de l'addition des valeurs de u et de v.

Dans la suite, nous emploierons la notation suivante $A \to A'$ pour indiquer la transformation de A en A' en une étape. Cette transformation est *mécanique* dans le sens où l'on peut exécuter la transformation en appliquant la règle qui est unique.

8.1.6 Opérationnalité d'une expression

On appelle *expression* (au sens sémantique) tout morceau de programme qui calcule une valeur. Par exemple `3` est une valeur donc une expression. De même les variables sont des expressions quand elles ne sont pas dans la partie gauche d'une affectation : dans `X := X + 1;` l'occurrence de `X` à droite est une valeur alors que celle de gauche ne l'est pas [4]. La définition formelle de la *valeur d'une expression* E est la suivante :

Si E est une valeur alors c'est la valeur elle-même ;

3. On définit une opération de base $\oplus$ par les équations suivantes, si E est un ensemble de couples (x_i, v_i) qui permet d'écrire que la valeur de E en x_i est v_i (noté $E(x_i) = v_i$) :

$$[E \oplus (x, v)](y) = \begin{cases} a \text{ si } y = x \\ DictId(y) \text{ sinon } x \neq y \end{cases}$$

4. Certains langages de programmation différencient syntaxiquement le rôle que jouent les variables dans un programme. Par exemple, dans le langage de programmation `Caml`, on écrit l'instruction précédente de la façon suivante : `x := !x + 1;`

Si E est une variable `x` alors c'est la valeur de $Mem(DictId(\texttt{x}))$;

Si E est composée d'opérations de base alors c'est la valeur des opérations de bases appliquées aux valeurs des sous expressions. Par exemple, la valeur de $a + b$ est la valeur de $plus(Mem(DictId(a)), Mem(DictId(a)))$.

Choisir entre E et e

On note $\langle \texttt{e} \mid Mem, DictId \rangle$ l'évaluation de l'expression `e` dans l'environnement (avec l'aide de) Mem et $DistId$. On dit que $\langle \texttt{e} \mid Mem, DictId \rangle \rightarrow \langle v \mid Mem, DistId \rangle$ est la réduction en une étape de l'expression `e`.

8.1.7 Déclaration des variables

Considérons la déclaration : `x : INTEGER;`

8.1.7.0.1 Informellement : Dans cette déclaration, `INTEGER` est ce qui s'appelle un *type* : cette phrase indique que `x` ne peut contenir que des valeurs définies par le type `INTEGER`. Nous n'avons pas défini les types dans notre modèle d'exécution pour ne pas alourdir la compréhension de la sémantique de l'exécution [5].

8.1.7.0.2 Formellement : On ajoute dans le dictionnaire la variable `x` et on lui attribue une case de la mémoire non déjà utilisée, ainsi que les cases suivantes (dépendant de la taille du type) : $DictId \oplus (\texttt{x}, new(Mem))$.

Dans la suite, on applique la réduction d'un bloc d'instructions en regardant de quelle forme est la première instruction. Pourquoi ici ?

8.1.8 Opérationnalité de l'affectation : `x := e;`

8.1.8.0.1 Informellement L'expression `e` est évaluée et la valeur obtenue, disons v, est placée dans la case mémoire désignée par `x` : $Mem \oplus (DictId(\texttt{x}), v)$.

8.1.8.0.2 Formellement : si $\langle \texttt{e} \mid Mem, DistId \rangle \rightarrow v$ alors

$$\langle \texttt{BEGIN x := e; i}_2 \texttt{ ... i}_n \texttt{ END;} \mid Mem, DictId \rangle \rightarrow$$

$$\langle \texttt{BEGIN i}_2 \texttt{ ... i}_n \texttt{ END;} \mid Mem \oplus (DictId(x), v), DictId \rangle$$

8.1.8.0.3 Remarque : `x : INTEGER := e;` est équivalent à faire d'abord `x : INTEGER;` puis ensuite `x := e;`.

8.1.9 La conditionnelle : `IF e THEN b1 ELSE b2 END IF;`

8.1.9.0.1 Informellement : Dans cette instruction, le type de `e` est `BOOLEAN`, avec que deux valeurs possibles {`TRUE`, `FALSE`}. On commence par calculer la valeur de `e` ; si cette valeur est `TRUE` alors on exécute le bloc d'instructions $\texttt{b}_1$ sinon (si la valeur est `FALSE`) on exécute le bloc d'instructions `b2`.

5. Il ne faut pas se méprendre sur le fait que nous ne traitons pas les types ; la notion de type est fondamentale en informatique car elle permet de relever certaines erreurs possibles avant l'exécution. La recherche est riche dans ce domaine [Typ-].

8.1.9.0.2 Formellement : si $\langle \texttt{e} \mid Mem, DictId \rangle \rightarrow \texttt{TRUE}$ alors

$$\langle \texttt{BEGIN IF e THEN } \texttt{b}_1 \texttt{ ELSE } \texttt{b}_2 \texttt{ END IF; } \texttt{i}_2 \texttt{ ... } \texttt{i}_\texttt{n} \texttt{ END;} \mid Mem, DictId \rangle \rightarrow$$

$$\langle \texttt{BEGIN } \texttt{b}_1 \texttt{ i}_2 \texttt{ ... } \texttt{i}_\texttt{n} \texttt{ END;} \mid Mem, DictId \rangle$$

si $\langle \texttt{e} \mid Mem, DictId \rangle \rightarrow \texttt{FALSE}$ alors

$$\langle \texttt{BEGIN IF e THEN } \texttt{b}_1 \texttt{ ELSE } \texttt{b}_2 \texttt{ END IF; } \texttt{i}_2 \texttt{ ... } \texttt{i}_\texttt{n} \texttt{ END;} \mid Mem, DictId \rangle \rightarrow$$

$$\langle \texttt{BEGIN } \texttt{b}_2 \texttt{ i}_2 \texttt{ ... } \texttt{i}_\texttt{n} \texttt{ END;} \mid Mem, DictId \rangle$$

8.1.10 L'itération bornée : FOR id IN $\texttt{e}_1..\texttt{e}_2$ LOOP b END LOOP;

8.2 Bibliographie

[Col-]
[Sco-]
[Typ-]