Chapitre

1 ALGORITHMES ET PROGRAMMES

AU PROGRAMME DE MPI

Alonzo Church (1903-1995) est un logicien américain, auteur de nombreux travaux en informatique théorique. Professeur à l'université de Princeton jusqu'en 1967, il y rencontre Von Neumann, Kleene ou encore Turing. Parmi ses nombreuses contributions, il complète les travaux de Gödel relatifs à l'indécidabilité, en développant les fondements du langage mathématique formel. Cela l'amène à décrire une logique basée sur le seul concept de fonction : le λ-calcul, dont on sait aujourd'hui qu'il permet de construire tout énoncé mathématique. Il est notamment connu pour un résultat, appelé *thèse de Church* (1936), qui affirme que les fonctions numériques formelles effectivement calculables sont les fonctions récursives générales.

Sommaire

1 La notion de programme

1.1. Avant le programme... l'algorithme

Un *algorithme* est un procédé automatique qui transforme une information symbolique (données ou entrées) en une autre information symbolique (résultats ou sorties). Seuls les problèmes qui sont susceptibles d'être résolus par un algorithme sont accessibles aux ordinateurs. Ce qui caractérise l'exécution d'un algorithme, c'est la réalisation d'un nombre fini d'opérations élémentaires (instructions), chacune d'elles étant réalisable en un temps fini. La quantité de données manipulées au cours du traitement est donc finie. La notion d'opération élémentaire dépend du degré de raffinement adopté pour la description du procédé. Ainsi, chaque algorithme peut être considéré comme une opération élémentaire dans un procédé plus important.

Pour passer de l'idée de l'algorithme à l'écriture effective de ces séquences d'opérations, on a souvent recours à un *pseudo langage* qui retranscrit les idées qui définissent les règles. Dans l'écriture de l'algorithme, le programme a un nom et des spécifications qui précèdent l'écriture de l'algorithme proprement dit. Nous reviendrons plus longuement sur ces aspects dans le chapitre 2. Il n'existe pas réellement de normes concernant ce pseudo langage. On demande néanmoins que ce dernier soit clair et explicite, fasse état des entrées et du résultat attendu, soit aéré et lisible (les différentes structures de contrôle sont indentées), permette de mettre l'accent sur la logique de construction de l'algorithme et soit commenté pour être lisible et aisément repris. Nous adoptons dans cet ouvrage un style de pseudo langage répondant à ces spécifications.

Exemple

Pour traduire le problème

« Factoriser dans $\mathbb{R}$ le trinôme du second degré $ax^2 + bx + c,\ a \neq 0$ »

en un algorithme, on repère les données (a, b et c), les éventuelles conditions sur ces entrées, le résultat désiré (les racines x_1, x_2, si elles existent, du trinôme) et on décrit la séquence d'opérations permettant de passer des entrées à la sortie.

Algorithme 1.1 : Calcul des racines d'un trinôme du second degré

Programme Factoriser(a, b, c)
▷ `Calcul, si elles existent, des racines du trinôme` $ax^2 + bx + c$
Entrées : $a,\ b,\ c$.
Sorties : $x_1,\ x_2$.
Précondition : $a \neq 0$
début
  $\Delta := b^2 - 4ac$
  si $\Delta < 0$ **alors**
    retourner *"Pas de solution"*
  sinon
    $x_1 := \frac{-b-\sqrt{\Delta}}{2a}$, $x_2 := \frac{-b+\sqrt{\Delta}}{2a}$
    retourner x_1, x_2

Pour effectivement mettre en œuvre un algorithme sur une machine, il est nécessaire de le traduire (l'implémenter) dans un *langage* permettant de réaliser les différentes opérations de l'algorithme. Le passage de l'algorithme au *code* va dépendre de la structure de l'algorithme, mais aussi du langage utilisé, de son type et de ses fonctionnalités.

1.2. Paradigmes de programmation

Les *paradigmes de programmation* sont une tentative de classer les langages dans des familles aux contours relativement bien définis. Un paradigme donné s'intéresse donc à l'ensemble de règles et concepts qui définissent une structure de langage et à la façon d'y formuler les problèmes et de les résoudre.
Les principaux paradigmes de programmation que nous détaillons ci-après découlent notamment de l'architecture des machines, de la notion théorique de calcul et de l'ensemble des besoins spécifiques au programme à produire.

1.2.1. Programmation impérative

La *programmation impérative* est le paradigme le plus rencontré. Il est caractérisé par une exécution des opérations en séquence et un état du programme modifié par ces opérations. En ce sens, ce paradigme met en avant les changements d'états et est alors proche de l'architecture de la machine.

Les programmations structurée et procédurale sont deux approches empruntant à la philosophie de la programmation impérative. La *programmation structurée* est issue d'une critique de Dijkstra [1] qui dénonce dans un article resté célèbre les méfaits de l'instruction de saut conditionnel `GOTO`, et des travaux de Böhm et Jacopini qui montrent que toute fonction calculable peut être exprimée en une suite de blocs d'instructions, d'instructions conditionnelles et de boucles. La *programmation procédurale* consiste quant à elle à découper un programme en procédures (ou fonctions), chacune représentant un ensemble d'étapes. L'objectif est de rendre le programme modulaire. Les procédures sont substituables par une autre implémentation (par exemple plus efficace en temps et/ou en espace), réutilisables, permettant ainsi la *factorisation* du code (éviter la répétition d'un même fragment de code) et du développement (réutilisation dans plusieurs programmes).

Le langage `C` est un langage impératif. Nous illustrerons les différents concepts à l'aide d'exemples dans ce langage. Le `Fortran`, le `Java` ou encore `Python` sont d'autres exemples de langages impératifs.

En programmation impérative, on distingue principalement cinq types d'instructions :

1. les *instructions élémentaires* qui effectuent une opération en mémoire et enregistrent le résultat pour qu'il soit réutilisé. Les opérations classiquement considérées comme élémentaires sont les opérations arithmétiques et les affectations.

1. https ://homepages.cwi.nl/ storm/teaching/reader/Dijkstra68.pdf

début
- $a := 2$
- $b := 3$
- $c := a + b$

```
int a,b,c;
a = 2;
b = 3;
c  a+b;
```

2. les *blocs d'instructions*, qui sont des suites d'instructions que la machine exécute de manière séquentielle. Les instructions peuvent être soit élémentaires, soit un ensemble d'instructions élémentaires regroupées en *procédures* ou *fonctions* (programmation procédurale).

début
- Ouvrir le fichier fic.txt
- Lire une valeur
- Fermer le fichier

```
int i;
FILE *f = fopen("fic.txt");
fscanf(f,"%d",&i);
fclose(f);
```

3. les *instructions conditionnelles* qui permettent à un bloc d'instructions de s'exécuter si une condition est remplie.

début
- **si** *condition= vrai* **alors**
  - Ouvrir le fichier fic.txt
  - Lire une valeur
  - Fermer le fichier

```
int i;
FILE *f = fopen("fic.txt");
if (condition)
{
 fscanf(f,"%d",&i);
 fclose(f);
}
```

4. les *instructions de boucle* qui permettent de répéter un bloc d'instructions, soit tant qu'une certaine condition est satisfaite, soit un nombre prédéterminé de fois.

début
- Ouvrir Le fichier fic.txt
- **tant que** *condition = vrai* **faire**
  - Lire une valeur
  - Mettre à jour la condition
- Fermer le fichier

```
int i = 0;
FILE *f = fopen("fic.txt");
while (i !=2)
{
 fscanf(f,"%d",&i);
}
fclose(f);
```

début
- Ouvrir Le fichier fic.txt
- **pour** $j := 1$ *à 10* **faire**
  - Lire une valeur
- Fermer le fichier

```
int i,j;
FILE *f = fopen("fic.txt");
for(j = 0 ; j<10 ; j++)
{
 fscanf(f,"%d",&i);
}
fclose(f);
```

5. les *sauts inconditionnels*, qui permettent de sauter à un bloc d'instructions plus loin dans le programme (nous ignorons ces instructions, puisque le programme officiel se place dans le cadre de la programmation impérative structurée).

Un des principaux défauts des langages de programmation impératifs est l'absence de *transparence référentielle*, propriété d'un langage ou d'un programme qui fait que toute variable, ainsi

que tout appel à une fonction, peut être remplacé par sa valeur sans changer le comportement du programme. Dans un paradigme impératif, le résultat d'une fonction dépend de l'état du programme à l'instant de son appel, et pas de l'argument auquel elle est appliquée. Ainsi, si

```
int i = 0;
int g(int n)
{
        i += n;
        return i;
}
```

alors $(g(1)+g(1)) \neq 2*g(1)$. En effet, au premier appel de g avec l'entier 1 passé en paramètre, $g(1)$ retourne 1. Au second appel, $g(1)$ retourne 2. L'évaluation de la relation fonctionnelle $g(1)$ dépend donc de l'état du contexte d'évaluation, par l'intermédiaire de la variable globale i qui change de valeur à chaque appel de g.

1.2.2. Programmation déclarative

La *programmation déclarative* est une famille de paradigmes de programmation dans lesquels on décrit ce qui doit être calculé, et pas comment. Les calculs ne dépendent pas de l'état du système et ne modifient pas cet état.

La *programmation fonctionnelle* est l'un des paradigmes déclaratifs, au même titre que la programmation descriptive ou la programmation par contraintes. Il est inspiré du λ-calcul, développé par A Church dans les années 1930, et considère les calculs en tant qu'évaluations de fonctions, objets au centre de ce paradigme. Un programme est décrit par un emboîtement de fonctions, chacune possédant plusieurs paramètres en entrée mais ne retournant qu'une seule valeur possible pour chaque jeu de données d'entrée. Puisque le changement d'état et la mutation des données ne peuvent être représentés par des évaluations de fonctions, aucun effet de bord n'est introduit dans ce paradigme.

`OCaml` étant un langage fonctionnel (textttLisp, `Scheme`, `Haskell` ou `Scala` étant d'autres exemples), il est utilisé pour illustrer les principaux concepts de ce paradigme.

Tous les langages possèdent la notion de fonction. Cependant, la programmation fonctionnelle exploite des fonctions avec des propriétés particulières, qui leur donnent des capacités de bonne composition et de décomposition. On peut ainsi caractériser la programmation fonctionnelle par quelques concepts clé :

— la notion de *fonction pure* : fonction qui, appelée avec les mêmes arguments, donne le même résultat, quel que soit l'état du système. La fonction est donc indépendante du contexte de son application, avec comme conséquence importante que si une expression est constituée de fonctions pures, alors il y a indépendance à l'ordre d'application de ces fonctions. Chaque sous-expression est évaluable à n'importe quel moment, et remplaçable par son résultat à n'importe quel moment (transparence référentielle),

— la notion de *première classe* : la fonction doit avoir le même statut qu'une valeur, en particulier : (i) pouvoir être nommée, affectée et typée ; (ii) pouvoir être définie à la demande ; (iii) pouvoir être passée en argument à une fonction ; (iv) pouvoir être le résultat d'une fonction et ; (v) pouvoir être stockée dans une structure de données. Les propriétés (ii) et (iv) induisent en particulier le concept de fonctions *d'ordre supérieur,* qui prennent

une ou plusieurs fonctions en paramètre et retournent une fonction. Dans l'exemple suivant

```
let rec cumsum f = function
        | [] -> 0
        | x :: l -> f(x) + cumsum f l
;;

let carre x = x * x ;;
cumsum (carre) [1; 2; 3 ; 4 ; 5] ;;
```

la fonction `cumsum` est une fonction d'ordre supérieur, calculant la somme cumulée des résultats d'une fonction appliquée aux éléments d'une liste,

— la *composition de fonctions* : combinaison de plusieurs fonctions, chacune ayant un et un seul rôle, et permettant de définir une logique de calcul à travers un ensemble de fonctions pures. Dans cette combinaison, il n'y a pas d'affectation, les résultats des calculs intermédiaires correspondent à de nouvelles valeurs (notion de première classe). Par exemple

```
let f x = x * x ;;
let rec g n =
    if n <= 1 then 1 else n * g (n - 1);;

let compose f g = fun n -> f (g n);;

let res = compose f g;;
```

compose (au vrai sens mathématique du terme) les fonctions f (carré) et g (factorielle). L'appel de `res n;;` calcule donc $f \circ g(n) = (n!)^2$.

— l'*immutabilité* : la programmation fonctionnelle n'introduit pas à proprement parler la notion de variable, comme on l'entend en programmation impérative. Lorsqu'une variable prend une valeur, elle ne doit plus changer,

— la *curryfication* : considérons la fonction

```
let mult x y =
 x*y ;;
```

`mult`, de signature `val mult : int -> int -> int = <fun>` est une fonction qui prend deux arguments entiers et retourne leur produit. L'appel de `mult 5 4` retourne donc la valeur 20. L'appel de `mult 5` retourne quant à lui un objet de signature `int -> int = <fun>`, qui est donc une fonction prenant un entier en paramètre et qui retourne un entier. L'opération de cette fonction consiste donc à multiplier par 5 l'entier passé en paramètre. La transformation de `mult` en une fonction à un seul argument est appelée la *curryfication*.[2]

Un code source écrit dans un langage de programmation fonctionnelle est plus concis, plus prédictible et plus facile à tester qu'un code impératif. En revanche, il peut paraître plus dense et plus ardu à comprendre au premier abord.

2. Du mathématicien Haskell Curry, dont les travaux ont posé les bases de la programmation fonctionnelle.

1.2.3. Programmation logique

La programmation logique est un paradigme déclaratif dans lequel un programme est un ensemble de *faits* et de *règles* exprimés dans une logique donnée. Un calcul est une requête sur cet ensemble de connaissances, qui est résolu grâce à un *moteur d'inférence*. Un programme logique vise donc à prouver qu'un énoncé peut être déduit à partir des faits et règles. L'objet de base est le *prédicat*, qui décrit une relation entre un certain nombre d'individus. Un programme logique correct est une description d'un problème suffisamment complète pour que la solution puisse en être déduite. Un des langages les plus connus suivant ce paradigme est le langage `Prolog`, inspiré de la logique du premier ordre.
La programmation logique peut être utilisée pour l'interrogation de bases de données, où les requêtes sont des formules logiques (calcul relationnel, chapitre 12).

1.2.4. Quel choix ?

Turing prouve en 1937 que les deux principaux modèles de calcul (λ-calcul, inspirant la programmation fonctionnelle et machines de Turing, proches de la programmation impérative) sont équivalents. La thèse de Church énonce à la même époque l'hypothèse que les règles formelles de calcul (machines de Turing, λ-calcul, fonctions récursives) formalisent correctement la notion de calculabilité. Il n'y a donc pas intrinsèquement de paradigme de programmation meilleur qu'un autre, mais selon le problème considéré un paradigme peut être plus « efficace » qu'un autre, où l'efficacité peut être mesurée de bien des manières (solution plus simple ou plus naturelle à implémenter, problème se prêtant plus particulièrement à un paradigme, ...).

1.3. Du fichier texte au programme

Pour pouvoir exécuter un algorithme écrit dans un langage de programmation, deux stratégies sont envisagées suivant le type de langage utilisé.

1.3.1. Langage compilé

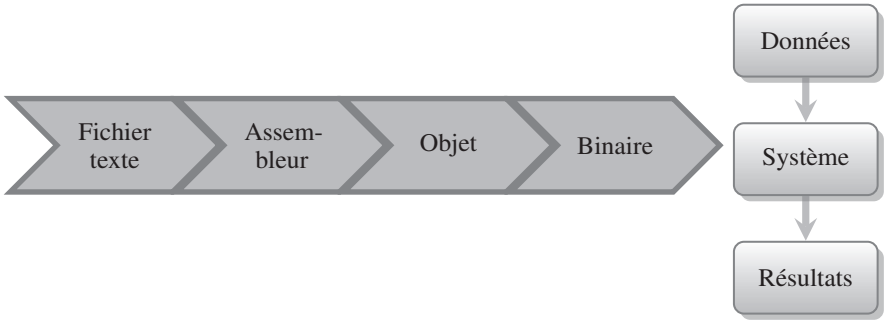
Dans le cas où le langage est un *langage compilé*, un *compilateur* traduit le code du programme écrit dans le langage de programmation dans le fichier texte en code machine avant son exécution. C'est uniquement après cette traduction que le programme est exécuté par le processeur qui dispose de toutes les instructions sous forme de code machine. Dans de nombreux cas, la traduction passe par les étapes suivantes :

— la *compilation* : réalisée par le compilateur, cette étape consiste à obtenir à partir du code source (en langage de haut niveau) l'équivalent en langage d'assemblage (langage assembleur). Le programme en langage d'assemblage est une suite d'instructions mnémotechniques décrivant les opérations que doit exécuter le processeur. Chaque processeur possède un jeu d'instructions spécifique, donc un langage d'assemblage spécifique,

— l'*assemblage* : réalisée par l'assembleur, cette étape consiste à obtenir à partir du programme en langage d'assemblage le programme binaire équivalent (appelé aussi module objet),

— l'*édition des liens* : réalisée par l'éditeur de liens, cette étape consiste à obtenir un programme exécutable à partir d'un ou plusieurs modules objets (ou bibliothèques de modules objets) compilés et assemblés séparément. Ces modules peuvent être des bibliothèques extérieures, d'autres programmes écrits par l'utilisateur...

Un programme compilé dépend donc du système d'exploitation et du matériel utilisé (architecture des processeurs). Le programme résultant est généralement optimisé (optimisation pendant la phase de compilation, le compilateur pouvant être paramétré pour favoriser la vitesse d'exécution du programme, l'empreinte mémoire du fichier binaire...) et de ce fait est plus rapide en exécution qu'une même version interprétée.



Parmi les langages compilés, on peut citer le `C`, le `C++` ou encore `OCaml` (il est cependant possible d'interpréter le code d'un programme `OCaml`, de manière interactive ou non, grâce au *toplevel*).

1.3.2. Langage interprété

Si le langage est *interprété*, le code du programme est traduit par un *interpréteur* pendant son exécution, qui joue le rôle d'interface entre le programme et le processeur. L'interpréteur traite le fichier texte du programme ligne par ligne, de manière à ce que les différentes instructions soient lues, analysées et préparées pour le processeur dans l'ordre. Si des instructions récurrentes sont rencontrées, elles sont ré-exécutées lorsque leur tour est arrivé. Pour traiter les lignes du programme, l'interpréteur utilise ses propres bibliothèques internes : lorsqu'une ligne de code source est convertie dans les commandes lisibles par la machine, elle est transmise au processeur. Ce processus s'achève soit lorsque l'ensemble du code a été interprété, soit lorsqu'une erreur est rencontrée.