

Table des matières

I	ALGORITHMIQUE ET PROGRAMMATION	5
1	Algorithmes et programmes	7
1	La notion de programme	8
1.1	Avant le programme... l'algorithme	8
1.2	Paradigmes de programmation	9
1.3	Du fichier texte au programme	13
2	Terminaison et correction d'un algorithme	15
2.1	Terminaison	15
2.2	Correction	17
3	Complexité d'un algorithme	19
3.1	Introduction	19
3.2	Complexité en temps	20
3.3	Différents types de complexité	22
3.4	Complexité spatiale	23
3.5	Notion de coût amorti	23
3.6	Exemples	25
4	Exercices	29
2	Bonnes pratiques de programmation	33
1	Discipline de programmation	34
1.1	Spécification	34
1.2	Pour un code "bien" écrit	34
1.3	Notion de programmation défensive	36
2	Validation et test de code	38
2.1	Notion de test	38
2.2	Niveaux de test	39
2.3	Graphe de contrôle	40
2.4	Chemin d'exécution	41
2.5	Critères de couverture	42
3	Exercices	45
3	Récursion et induction	47
1	Récurrance et récursivité	48
1.1	Récurrance faible, récurrance forte	48

1.2	Algorithme récursif	48
1.3	Différents types de récursivité	49
1.4	Évaluation d'algorithmes récursifs	52
2	Induction	54
2.1	Définitions	54
2.2	Preuve par induction structurelle	56
2.3	Raisonnement par induction bien fondée	57
3	Retour sur les fonctions récursives	58
3.1	Terminaison et correction	58
3.2	Gestion mémoire d'une fonction récursive	60
4	Exercices	61

4 Décidabilité et classes de complexité 63

1	Notions générales	64
1.1	Introduction	64
1.2	Définitions	66
2	Les classes de problèmes	67
2.1	La classe P	67
2.2	La classe EXP	68
2.3	La classe NP	68
3	Réduction polynomiale	69
4	Les problèmes <i>NP</i> - complets	70
4.1	Problèmes <i>NP</i> -complets	70
4.2	Théorème de Cook	71
4.3	Réduction de problèmes <i>NP</i> -complets	72
4.4	Hiérarchie des classes	74
5	Problème d'optimisation	74
6	Exercices	75

II ALGORITHMIQUE 77

5 Structures de données 79

1	Types et abstraction	80
1.1	Types simples	80
1.2	Types composés	81
1.3	Pointeurs	83
2	Structures de données séquentielles	84
2.1	Liste	84
2.2	Pile	85
2.3	File	88
2.4	Tableau associatif	93
2.5	Sérialisation	98
3	Structures de données hiérarchiques	101

3.1	Définition	101
3.2	Quelques arbres particuliers	102
3.3	Parcours d'un arbre	107
3.4	File de priorité	109
3.5	Structure Unir & trouver	111
3.6	Implémentation	114
4	Structures de données relationnelles	118
4.1	Définitions	118
4.2	Propriétés	123
4.3	Relation aux arbres	124
4.4	Représentation d'un graphe	125
4.5	Implémentation	126
5	Exercices	129
6	Algorithmes de décomposition	131
1	Algorithmes gloutons	132
1.1	Définition	132
1.2	Exemples	132
2	Diviser pour régner	136
2.1	Présentation	136
2.2	Exemples	137
3	Programmation dynamique	142
3.1	Définition	142
3.2	Exemples	144
4	Exercices	151
7	Algorithmes d'approximation	155
1	Algorithmes probabilistes	156
1.1	Algorithmes déterministes/probabilistes	156
1.2	Algorithmes probabilistes	156
1.3	Algorithmes de type Monte Carlo	157
1.4	Algorithmes de type Las Vegas	158
1.5	Exemples	160
2	Algorithmes d'approximation	166
2.1	Définitions	166
2.2	Exemples	168
3	Exercices	170
8	Algorithmes d'exploration	175
1	Recherche par force brute	176
1.1	Principe	176
1.2	Exemple	176
2	Algorithme de retour sur trace	176
2.1	Principe	176

2.2	Exemple : le problème des n reines	178
2.3	Exemple : le problème SUBSET	179
3	Algorithme par séparation et évaluation	180
3.1	Principe	180
3.2	Fonction d'évaluation	182
3.3	Sélection	182
3.4	Séparation	183
3.5	Exemple : problème du sac à dos	183
3.6	Exemple : assignation de tâches	185
4	Exercices	188
9	Algorithmique des graphes	191
1	Parcours de graphes	192
1.1	Parcours en largeur	193
1.2	Parcours en profondeur	195
2	Applications des parcours de graphes	198
2.1	Caractérisation de graphes bipartis	198
2.2	Tri topologique	199
2.3	Connexité et composantes connexes d'un graphe non orienté	200
2.4	Composantes fortement connexes d'un graphe orienté	200
3	Plus courts chemins	201
3.1	Recherche des plus courts chemins à partir d'un sommet donné	202
3.2	Recherche des plus courts chemins entre toutes les paires de sommets	205
4	Recherche d'un arbre couvrant de poids minimum	207
4.1	Position du problème	207
4.2	Algorithme de Kruskal	208
5	Couplage maximum dans un graphe biparti	210
5.1	Définitions	210
5.2	Algorithme	212
6	Exercices	213
10	Algorithmique des textes	217
1	Recherche de motif dans un texte	218
1.1	Notations	218
1.2	Algorithme naïf	218
1.3	Algorithme de Boyer-Moore	219
1.4	Algorithme de Rabin-Karp	222
2	Compression de données	224
2.1	Algorithme de Huffman	225
2.2	Compression LZW	229
3	Exercices	231
11	Algorithmique pour l'IA et les jeux	235
1	Apprentissage supervisé	236

1.1	<i>k</i> -plus proches voisins	236
1.2	Arbres de décision	239
1.3	Évaluation des résultats	242
2	Apprentissage non supervisé	245
2.1	Algorithme des <i>k</i> -moyennes	245
2.2	Classification hiérarchique ascendante	248
3	Jeux d'accessibilité à deux joueurs	251
3.1	Définitions	252
3.2	Jeux d'accessibilité	253
4	Algorithme Minimax et élagage $\alpha\beta$	255
4.1	Algorithme Minimax	256
4.2	Élagage alpha-beta	257
4.3	Exemples d'heuristiques	259
5	Algorithme A^*	260
5.1	Graphes d'états	260
5.2	Algorithme A^*	261
6	Exercices	264

III INTRODUCTION AUX BASES DE DONNÉES 267

12 Manipulation de bases de données relationnelles 269

1	Le modèle relationnel	270
1.1	Définitions	270
1.2	Modélisation d'une base de données relationnelle	272
2	Manipulation des données	278
2.1	Le langage SQL	278
2.2	Obtention des données	279
2.3	Expression des projections	279
2.4	Expression des restrictions	279
2.5	Tri et présentation des résultats	280
2.6	Expression des jointures	281
2.7	Expression de manipulation des données	282
2.8	Fonctions statistiques	283
2.9	Regroupements	283
3	Exercices	284

IV GESTION DES RESSOURCES DE LA MACHINE 287

13 Elements de codage 289

1	Exemple introductif	290
2	Représentation des nombres	291
2.1	Représentation des entiers naturels	291

2.2	Représentation des entiers relatifs	292
2.3	Représentation des nombres flottants	292
2.4	Norme IEEE 754	293
3	Calcul sur les nombres flottants	294
3.1	Arrondis	295
3.2	Prise en compte des petits nombres	296
3.3	Absorption	296
3.4	Élimination	296
3.5	Monotonie des fonctions	297
3.6	Associativité	298
3.7	Quelques conseils	298
4	Retour sur l'exemple	299
14	Gestion de la mémoire d'un programme	301
1	La mémoire	302
1.1	La mémoire physique	302
1.2	Espace mémoire des programmes	302
2	Variables	303
2.1	Portée	303
2.2	Durée de vie	304
2.3	Nature des variables	304
3	Stockage des données en mémoire	305
3.1	La pile	306
3.2	Le tas	310
4	Quelques questions relatives à la gestion de la mémoire	311
15	Gestion des fichiers	313
1	Notion de fichier	314
1.1	Qu'est-ce qu'un fichier?	314
1.2	Types de fichiers	314
1.3	Accès, droits et attributs	315
2	Implémentation interne	316
2.1	La notion d'inode	316
2.2	Manipulation des inodes	318
3	Accès aux fichiers en C et Caml	319
3.1	Langage C	319
3.2	Langage OCaml	323
16	Synchronisation et concurrence	327
1	Introduction	328
1.1	Exemple introductif	328
1.2	Définitions	328
2	Algorithmes	330
2.1	Position du problème	330

2.2	Algorithme de Peterson	331
2.3	Algorithme de la boulangerie de Lamport	333
2.4	Analyse des algorithmes	335
3	Sémaphore et mutex	335
3.1	Sémaphores	335
3.2	Mutex	336
4	Quelques problèmes classiques	337
4.1	Le problème du rendez-vous	337
4.2	Le problème producteur-consommateur	338
4.3	Le dîner des philosophes	339
5	Implémentations	341
5.1	Langage C	341
5.2	Langage OCaml	343

V ÉLÉMENTS DE LOGIQUE

345

17 Logique

347

1	Syntaxe des formules logiques	348
1.1	Logique propositionnelle	348
1.2	Logique du premier ordre	352
2	Sémantique de vérité du calcul propositionnel	356
2.1	Sens des connecteurs	357
2.2	Valeur d'une formule	357
2.3	Équivalence sur les formules	360
2.4	Formes normales	361
3	Le problème SAT	366
3.1	Définition	366
3.2	Illustration	366
3.3	Algorithme de Quine	367
4	Implémentation	369
5	Quelques applications	372
5.1	Étude d'expressions booléennes	372
5.2	Preuve de programmes	372
5.3	Bases de données	373
6	Exercices	373

18 Dédution naturelle

375

1	Séquent	376
2	Preuve en déduction naturelle	377
2.1	Définitions	377
2.2	Règles d'introduction et d'élimination	378
2.3	Retour sur la dérivation	383
3	Exercices	384

19 Langages et expressions régulières	389
1 Définitions	390
1.1 Alphabet, mot, langage	390
1.2 Opérations sur les mots	391
1.3 Opérations sur les langages	391
1.4 Langages locaux	392
2 Expressions régulières	394
2.1 Notion de motif	394
2.2 Expressions régulières, langages réguliers	395
2.3 Expressions régulières linéaires	396
2.4 Implémentation	397
2.5 Expressions régulières étendues	399
3 Exercices	400
20 Automates finis	403
1 Automates finis déterministes	404
1.1 Définition	404
1.2 Représentation graphique	404
1.3 Automate émondé	406
1.4 Retour sur le lemme de l'étoile	407
1.5 Complexité d'un AFD	407
2 Automates finis non déterministes	408
2.1 Définition	408
2.2 Déterminisation d'un AFND	409
2.3 Complexité d'un AFND	411
3 Automates finis et langages réguliers	411
3.1 Automate associé à un langage local	412
3.2 Algorithme de Berry-Sethi	413
3.3 Des automates aux expressions régulières	415
3.4 Stabilité des langages reconnaissables	419
4 Implémentation	420
5 Exercices	421
21 Grammaires non contextuelles	425
1 Définitions	426
2 Arbres d'analyse	428
3 Ambiguïté et équivalence	432
3.1 Ambiguïté	432
3.2 Équivalence	433
4 Analyse syntaxique	433
5 Exercices	435

Éléments de Correction des exercices	437
Aide-mémoire	479
Liste des algorithmes	485
Index	489

Première partie

Algorithmique et programmation

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	8
1.1	Avant le programme... l'algorithme	8
1.2	Paradigmes de programmation	9
1.3	Du fichier texte au programme	13
2	Terminaison et correction d'un algorithme	15
2.1	Terminaison	15
2.2	Correction	17
3	Complexité d'un algorithme	19
3.1	Introduction	19
3.2	Complexité en temps	20
3.3	Différents types de complexité	22
3.4	Complexité spatiale	23
3.5	Notion de coût amorti	23
3.6	Exemples	25
4	Exercices	29

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 Dijkstra¹ 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'*immuabilité* : 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.

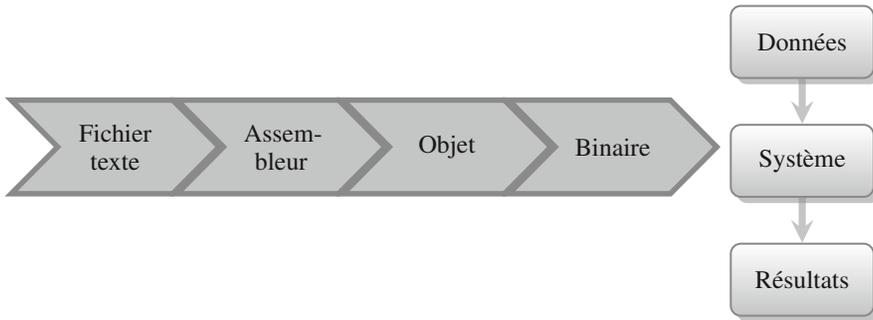
1.3.1. Langage compilé

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...

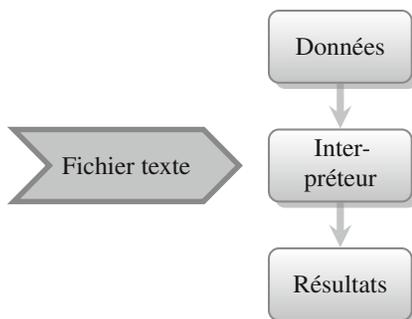
Un programme compilé dépend donc du système d'exploitation et du matériel utilisé (architecture des processeurs). Le programme résultant est généralement optimisé (optimisation pendant la phase de compilation, le compilateur pouvant être paramétré pour favoriser la vitesse d'exécution du programme, l'empreinte 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 *tolevel*).

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.



Python, HTML sont des exemples de langages interprétés.

Le tableau suivant donne quelques critères de comparaison entre ces deux types de langages.

	Compilés	Interprétés
Traduction du code	Avant l'exécution	Pendant l'exécution
Traduction	Ensemble du code	Ligne par ligne
Affichage des erreurs	À la fin de la compilation	Après chaque ligne
Efficacité de la traduction	Forte	Faible
Vitesse de traduction	Faible	Forte
Effort de développement	Forte	Faible
Portabilité	Faible	Forte
Vitesse d'exécution	Plus forte	Plus faible

R Un langage de programmation peut avoir une implémentation compilée et une autre interprétée. De même, il existe des langages *semi-compilés* (ou semi-interprétés) pour lesquels le code source est soumis à une phase de compilation intermédiaire vers un fichier non exécutable (il y a donc nécessité d'un interpréteur) avant de générer du code objet ou du langage machine pour la machine sur laquelle sera exécutée le programme. Le plus connu de ces langages est Java.

2

Terminaison et correction d'un algorithme

On s'intéresse dans la suite aux algorithmes non récursifs. Le cas des algorithmes récursifs est traité dans le chapitre 3 section 1.4.

2.1. Terminaison

Turing démontre en 1936 que le problème de l'arrêt est indécidable, c'est-à-dire qu'il n'existe pas de programme informatique qui prend comme entrée une description d'un programme informatique et un paramètre et qui, grâce à la seule analyse du code, répond Vrai si le programme s'arrête sur son paramètre et Faux sinon. Bien que vrai dans le cas général, ce résultat ne doit pas empêcher l'étude de l'arrêt des algorithmes dans des cas simples.