Modèles de machines

Jean Berstel et Luc Boasson

Mots-clés : machines de Turing, problèmes indécidables, problèmes NP-complets, automates à pile, grammaires, hiérarchie de Chomsky.

Résumé : le modèle des automates finis est trop limité pour rendre compte d'importantes classes d'algorithmes qui font intervenir des structures arborescentes. Les automates à pile, décrits dans ce chapitre, sont particulièrement bien adaptés au traitement itératif de procédures récursives, et interviennent, dans ce cadre, notamment en compilation et plus généralement dans la manipulation d'arbres et de graphes. Mais les automates à pile ne sont qu'une version restreinte du modèle de machines introduites par Alan Turing. Les machines de Turing constituent un modèle mathématique dont il est communément admis qu'elles représentent précisément ce qui est calculable par un algorithme, et donc aussi par les ordinateurs. A contrario, tous les problèmes dont on sait prouver qu'ils ne peuvent être résolus par une machine de Turing ne peuvent pas être traités par ordinateur.

1 Introduction

La notion d'effectivité dans les opérations mathématiques a préoccupé les scientifiques, et notamment les mathématiciens, bien avant l'introduction des premiers ordinateurs. Dans le célèbre programme de recherches tracé par Hilbert au congrès international des mathématiciens à Paris en 1900, bon nombre de problèmes concernaient l'effectivité des calculs. D'ailleurs, le plus souvent il était admis que l'on trouverait, grâce au progrès de la science, des algorithmes pour résoudre les problèmes posés.

L'impossibilité de résoudre certains problèmes de manière effective signifie qu'il n'existe pas d'algorithme pour les résoudre. Or, autant on peut plus ou moins facilement se convaincre qu'un algorithme donné résout un problème posé, autant il est difficile d'admettre, sans preuve, qu'un problème ne possède pas de solution effective. Pour s'en convaincre, une démonstration est nécessaire, et pour cela, il faut formuler de manière précise et donc mathématique, ce que l'on entend par le mot « algorithme ».

Plusieurs formulations mathématiques de la notion d'algorithme ont été proposées. Elles sont de nature assez différente, et sans entrer trop dans les détails, on peut les classer en deux grandes familles : celles qui décrivent, par des opérations de construction, les fonctions que l'on peut calculer (fonctions dites récursives), et celles qui formalisent une machine faisant des calculs élémentaires, en grand nombre mais en nombre fini (schémas de Markov et machines de Turing). La deuxième famille — celle que nous décrivons de manière plus détaillée — comporte aussi des modélisations plus réalistes des ordinateurs (machines à accès direct). Elles montrent que les machines de Turing savent calculer ce qu'un superordinateur sait faire, même si cela prend du temps.

Il a été démontré que tous les modèles d'algorithmes sont équivalents dans leur puissance d'expression, et ceci est un bon argument pour admettre que ce modèle reflète bien ce qu'intuitivement on entend par calculable (cette adéquation entre le concept intuitif et les modèles mathématiques est appelée la « thèse de Church »).

2 Modèles de machines

Une *machine* est vue comme un objet mathématique. Elle comporte un composant fini appelé *programme* ou *contrôle fini*, et une structure sur laquelle elle opère qui est sa *mémoire*, et qui peut être infinie. À tout moment, le contrôle fini est dans un *état* particulier, et le nombre d'états distincts qui sont possibles est fini. La mémoire sert à déposer des données sous la forme de symboles pris dans des ensembles finis appelés *alphabets*. Dans d'autres machines, la mémoire contient au contraire des nombres, en général des entiers.

La mémoire est organisée en une structure régulière de *cellules*, et l'information est rangée dans les cellules. La structure de la mémoire est discrète et linéaire, mais dans certains modèles on rencontre une organisation de la mémoire en grille ou en arbre. Même si la mémoire est en principe infinie, seule une partie finie est utilisée dans un calcul. De manière formelle, ceci est réalisé en convenant qu'un symbole spécial (le symbole blanc □) figure dans les cellules de mémoire qui n'ont pas encore été utilisées.

Les programmes sont écrits dans un formalisme plus ou moins proche des langages de programmation. Cela peut aller d'une sorte d'assembleur (pour ce que l'on appelle les RAM) jusqu'à une simple liste d'instructions de réécriture. Dans les machines de Turing par exemple, les instructions sont d'un seul type qui combine la lecture en mémoire, le test, l'écriture en mémoire, et le calcul de l'adresse suivante.

Pour les entrées-sorties, deux parties de la mémoire sont distinguées : la zone de données où les données sont accessibles, symbole par symbole, et la zone des résultats, où le résultat est communiqué au monde extérieur, à nouveau symbole par symbole. Les calculs peuvent être classés en deux catégories :

- les calculs *en ligne*, où les données ne sont disponibles pour le traitement qu'une fois, comme des signaux venant d'une source extérieure. Si l'on veut disposer d'un symbole une deuxième fois, il doit être rangé quelque part dans la mémoire. Les automates finis sont des machines de ce type ;
- les calculs *hors ligne*, où les données sont disponibles en permanence, dans la zone de données, et peuvent être parcourues à loisir, et dans tous les sens.

Dans les modèles de machine de Turing les plus rudimentaires, à une seule bande, il n'y a pas de séparation entre les zones de données, de travail, et de résultat. Les données figurent alors dans la mémoire au début du calcul, et le résultat y figure à la fin du calcul. Nous allons montrer comment on peut perfectionner ce modèle si le besoin s'en fait sentir.

Pour formaliser la notion de calcul, nous avons besoin de la notion de *configuration*. Une configuration est la description complète de l'état d'une machine, c'est-à-dire l'état complet de sa mémoire, y compris de l'unité de contrôle. Pour une machine de Turing à plusieurs bandes et plusieurs têtes, il s'agit du contenu de chaque bande, de la position de chaque tête, et de l'état de l'unité de contrôle. Quand le programme d'une machine est plus élaboré, il peut être utile d'ajouter au modèle un compteur ordinal qui contient le numéro de l'instruction à exécuter.

La *relation de transition* entre configurations est la relation qui décrit le passage d'une configuration à la suivante par l'application d'une étape de calcul. Ici, une étape consiste en l'application d'une règle, ou en l'exécution d'une instruction de programme. Un *calcul* est une suite d'étapes de calcul. Un calcul est décrit par la séquence des configurations par lesquelles passe la machine.

Ici, on peut faire la distinction entre machines *déterministes* et *non déterministes*. Dans les machines déterministes, il n'y a jamais plus d'une transition possible à partir d'une configuration, alors que dans une machine non déterministe, plusieurs transitions sont possibles. Il y a donc plusieurs calculs possibles, à partir d'une configuration donnée, et ils sont tous envisageables au même titre.

3 Les machines de Turing

Nous présentons ici le modèle des machines de Turing, dénommées ainsi d'après le mathématicien Alan Turing qui les a introduites en 1936. Ces machines modélisent de façon élémentaire ce qui est calculable, et il est de prime abord surprenant qu'elles soient capables d'égaler les ordinateurs (en pouvoir de calcul, pas en vitesse). La vérification formelle que tout programme, rédigé dans un langage de programmation évolué, peut être exécuté par une machine de Turing, est en fait un compilateur. Elle procède par enchaînement et empilement de machines de plus en plus complexes.

Le modèle des machines de Turing, par sa simplicité et par sa puissance, a connu un engouement très important de la part des mathématiciens et des informaticiens, et continue à faire partie du bagage de base de tout informaticien scientifique.

3.1 Description et fonctionnement d'une machine de Turing

Une machine de Turing est composée de quatre parties :

– une *mémoire* ; c'est une bande unidimensionnelle infinie des deux côtés, et partagée en cellules ou *cases*. Chaque case est susceptible de contenir une information, à choisir parmi un jeu fixé et fini de possibilités. Chaque information est représentée par un symbole et l'ensemble de ces symboles constitue l'alphabet de la machine de Turing ;

– une *tête de lecture-écriture* ; ce dispositif permet de connaître le contenu d'une case de la bande, et éventuellement de la modifier. Cette tête peut se déplacer, sur la bande, d'une case par étape de calcul, soit vers la gauche, soit vers la droite. On ne peut connaître le contenu que d'une seule case à la fois ;

– une *unité de contrôle* ; elle est susceptible de se trouver dans un nombre fini de situations prédéfinies appelées des *états*. L'état courant est toujours connu ;

– un *programme*. C'est un ensemble fini d'instructions. Chaque instruction indique les modifications que l'on peut apporter à la bande, à la position de la tête de lecture-écriture, et l'état de l'unité de contrôle.

Le contenu représenté sur la bande est toujours fini, même si la bande est infinie. En d'autres termes, toutes les cases de la bande sont vides sauf

un nombre fini, que l'on suppose de plus contiguës. De manière équivalente, les cases non vides forment un mot fini sur l'alphabet de la machine. De manière formelle, on convient qu'une case vide contient le symbole blanc □.

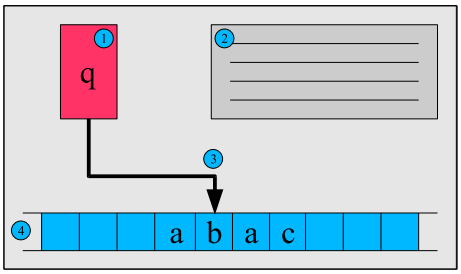

Figure 1.— Structure d'une machine de Turing. (1) l'unité de contrôle, (2) le programme, (3) la tête de lecture-écriture et (4) la bande de mémoire. Sa configuration est le triplet (a, q, bac).

Une *configuration* d'une machine de Turing se compose de l'état de son unité de contrôle, du contenu de la bande et de la position de la tête sur la bande. Cette position partage la bande en deux parties. Ainsi, une configuration est un triplet (g, q, d) formé des contenus à gauche g et à droite d de la tête et de l'état q de la machine. Par convention, la tête de lecture-écriture est placée au-dessus de la première lettre de d.

Une *instruction* est formée de cinq données : un état q, une lettre a qui peut être le symbole blanc, une instruction de mouvement (G pour gauche, D pour droit), un symbole b, et un état p. Chaque instruction s'écrit donc sous la forme $(q, a) \rightarrow (M, b, p)$, où M est l'un des symboles G ou D.

Une instruction $(q, a) \rightarrow (M, b, p)$ s'applique à une configuration (g, q, d) pourvu que a soit la première lettre de d. Dans ce cas, la lettre a est remplacée par b, la tête se déplace d'une case dans la direction M, et l'état passe en p. Le passage de la configuration (g, q, d) à une nouvelle configuration (g', p, d') par application d'une instruction est une *transition* ou un *mouvement*, et une séquence de mouvements est un calcul.

Une machine de Turing effectue des calculs sur des données représentées par des mots écrits sur la bande au début du calcul. Si x est un tel mot, la *configuration initiale* pour x est la configuration où la tête se trouve sur la lettre la plus à gauche de x, et où l'état est un état *initial* i particulier. La configuration initiale pour x est donc (ε, i, x), où ε désigne le mot vide.

Un calcul se poursuit aussi longtemps que possible, c'est-à-dire tant qu'il y a une instruction applicable. Un calcul peut donc se poursuivre indéfiniment, ou s'arrêter parce qu'il se trouve dans une configuration où il n'y a aucune instruction applicable.

Si la machine de Turing est déterministe, un seul de ces deux cas est possible pour une donnée x. Si la machine n'est pas déterministe, il peut exister, pour une donnée particulière, des calculs infinis et des calculs finis.

Selon qu'une machine de Turing est vue comme une machine qui réalise une fonction ou comme une machine qui reconnaît un langage, l'interprétation des calculs est différente.

Quand une machine de Turing est construite pour réaliser une fonction f, on la suppose déterministe. L'ensemble des mots x pour lequel le calcul s'arrête est le *domaine* de la fonction f, et le mot qui se trouve sur la bande au moment de l'arrêt est la valeur de la fonction $f(x)$ réalisée pour la donnée x.

Si en revanche une machine de Turing est vue comme un reconnaisseur, on convient fréquemment de distinguer, en plus, un ensemble d'états particuliers appelés états *terminaux* ou d'*acceptation*. Un mot x est alors reconnu par la machine s'il existe un calcul, pour le mot x, qui se termine dans un état terminal. Le *langage reconnu* par la machine est l'ensemble des mots reconnus. Notons que pour une donnée x, il peut exister, dans une machine de Turing non déterministe, à la fois des calculs infinis, des calculs qui se terminent dans des états qui ne sont pas terminaux et des calculs qui se terminent dans des états terminaux.

3.2 Exemples

Le premier exemple est une machine qui ajoute 1 à un entier écrit en binaire. Ainsi, pour la donnée 1011, le résultat est 1100.

Notre machine de Turing opère en deux phases. Elle commence par déplacer la tête vers la dernière lettre de la donnée x, puis elle examine ses bits de la droite vers la gauche. Tant qu'elle rencontre un 1, elle le remplace par un 0. Dans le cas contraire, elle écrit 1 et s'arrête. Notre machine a trois états : i (initial), j et k, et les instructions

$$\begin{array}{lcl} (i, 0) & \rightarrow & (D, 0, i) \\ (i, 1) & \rightarrow & (D, 1, i) \\ (i, \square) & \rightarrow & (G, \square, j) \\ (j, 1) & \rightarrow & (G, 0, j) \\ (j, 0) & \rightarrow & (G, 1, k) \\ (j, \square) & \rightarrow & (G, 1, k) \end{array}$$

On peut imaginer une machine un peu plus sophistiquée qui remet la tête sur la cellule le plus à gauche du résultat. Dans ce cas on peut enchaîner plusieurs copies de cette machine et réaliser la fonction qui à un entier n écrit en binaire associe l'écriture binaire de $n + h$ pour un entier fixe h.

Notre deuxième exemple réalise la multiplication par 3 d'un entier écrit en binaire. Par exemple, pour la donnée $x = 1011$, le résultat est 100001. À nouveau, la machine se déplace sur la cellule la plus à droite de la donnée, puis elle examine les bits, de la droite vers la gauche. L'état de la machine peut prendre l'une des valeurs r_0, r_1, r_2, selon que la retenue du calcul en cours est 0, 1, ou 2. Ainsi, si la retenue courante est 2, l'état est r_2. On applique alors les instructions de multiplication par 3 : par exemple, si le bit courant est 0, la machine écrit 0 et passe de l'état r_2 en r_1 ; si le bit courant est 1, la machine écrit 1 et reste dans l'état r_2.

Les instructions sont les suivantes :

$(i, 0)$	$\rightarrow$	$(D, 0, i)$
$(i, 1)$	$\rightarrow$	$(D, 1, i)$
$(i, \square)$	$\rightarrow$	$(G, \square, r_0)$
$(r_0, 0)$	$\rightarrow$	$(G, 0, r_0)$
$(r_0, 1)$	$\rightarrow$	$(G, 1, r_1)$
$(r_1, 0)$	$\rightarrow$	$(G, 1, r_0)$
$(r_1, 1)$	$\rightarrow$	$(G, 0, r_2)$
$(r_2, 0)$	$\rightarrow$	$(G, 0, r_1)$
$(r_2, 1)$	$\rightarrow$	$(G, 1, r_2)$
$(r_2, \square)$	$\rightarrow$	$(G, 0, r_1)$
$(r_1, \square)$	$\rightarrow$	$(G, 1, r_0)$

Notre troisième exemple est la transformation de l'écriture ternaire d'un entier en écriture binaire. Si la donnée est par exemple 201, qui est l'écriture en base 3 du nombre 19, le résultat est 10011, son écriture en base 2. La machine opère de la gauche vers la droite en autant de phases que la donnée contient de chiffres ternaires. À chaque phase, la machine a déjà transformé les chiffres de gauche en binaire. La phase courante consiste en deux étapes : d'abord une multiplication par 3 du nombre déjà écrit en binaire, puis l'addition du nombre représenté par le chiffre courant, ce qui est réalisé par 0, 1 ou 2 appels à la machine qui incrémente de 1 l'entier sur la bande.

Le dernier exemple illustre comment on peut, en enchaînant des machines, composer des fonctions de plus en plus complexes à partir de fonctions élémentaires. Un autre mécanisme, tout aussi fondamental, est l'usage d'une machine de Turing comme auxiliaire à l'intérieur d'une autre. De cette manière, on simule l'emploi de sous-programmes, technique bien connue des programmeurs.

3.3 Variantes des machines de Turing

De nombreuses variantes ont été définies et étudiées, en vue de mettre en évidence certaines propriétés ou de faciliter certaines constructions. Une machine de Turing à *plusieurs bandes* possède un certain nombre de bandes et, pour chacune de ces bandes, d'une tête de lecture-écriture. Une transition d'une telle machine opère simultanément sur toutes les bandes : sur chacune, la tête peut écrire une lettre et se déplacer d'une case, indépendamment des autres. Seul l'état de contrôle est commun. Une machine à plusieurs bandes a la même puissance qu'une machine à une bande.

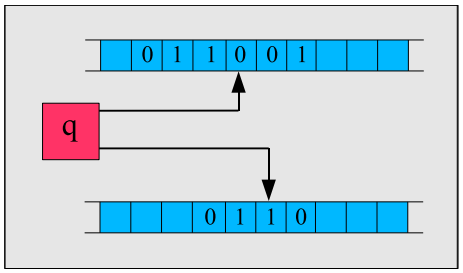

Figure 2.— Une machine de Turing à deux bandes. Elle possède deux têtes de lecture-écriture qui se déplacent de façon indépendante.

Pour simuler par exemple une machine à deux bandes, on imagine une machine qui possède une seule bande, mais où chaque cellule est composée de quatre cases empilées. Si on observe la bande horizontalement, on voit apparaître quatre niveaux, et les cases d'un même niveau forment un « couloir ». Deux couloirs servent à contenir les données des deux bandes, et deux autres couloirs indiquent, par un symbole particulier, l'emplacement des têtes de lecture-écriture. La machine simulante parcourt sa bande à la recherche des emplacement des têtes de lecture-écriture, et modifie en conséquence les contenus des pistes des cellules.

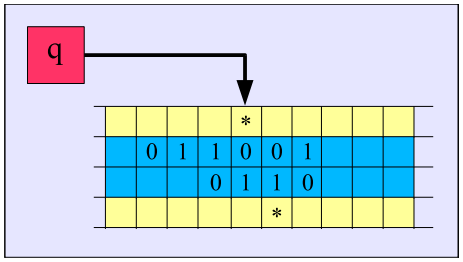

Figure 3.— Une machine de Turing à une bande qui simule une machine à deux bandes. Les deux couloirs centraux contiennent les contenus des bandes de la machine à simuler, et les deux couloirs extrêmes repèrent les positions des têtes.

Une application utile est une machine à trois bandes, l'une contenant la donnée qui n'est pas modifiable, une deuxième, destinée aux calculs intermédiaires, et une troisième servant à la présentation des résultats et qui n'est pas effaçable (en d'autres termes, sur la bande d'entrée, la tête

n'écrit jamais, et sur la bande de sortie, la tête ne lit jamais). Dans un tel modèle, on peut séparer le coût, en temps et en place, pour le calcul proprement dit, du coût et de la place nécessaire pour représenter la donnée et le résultat. Ainsi, quand on dit qu'un calcul requiert une place logarithmique en fonction de la taille de l'entrée, on fait référence à la place additionnelle, sur la bande de calcul.

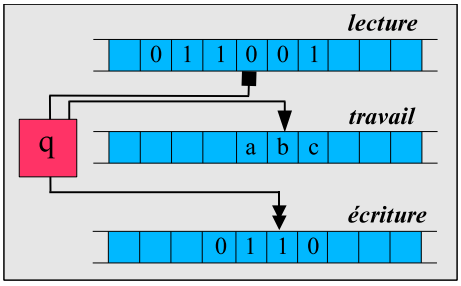

Figure 4.— Une machine de Turing à trois bandes : une bande de lecture, qui ne peut pas écrire, une bande d'écriture qui ne sait pas lire, et une bande de travail. Ce modèle sert à décrire la complexité des calculs.

Une distinction doit être faite à propos des machines déterministes et non déterministes. Il est tout à fait remarquable qu'une machine non déterministe $\mathcal{M}$ puisse être simulée par une machine déterministe $\mathcal{T}$ en ce sens que les mots reconnus par ces deux machines sont les mêmes. Pour ce faire, la machine $\mathcal{T}$ parcourt l'ensemble des calculs de la machine $\mathcal{M}$, par longueur croissante, et s'arrête dès qu'un calcul de $\mathcal{M}$ s'arrête dans un état d'acceptation.

La comparaison entre les machines déterministes et les machines non déterministes est un problème de base pour tout modèle de machines. Pour les automates finis (voir le chapitre ££[Pin]) et pour les machines de Turing, ces deux variantes ont le même pouvoir d'expression. Pour les automates à pile en revanche, les langages reconnus ne sont pas les mêmes (voir ci-dessous).

Même si les langages reconnus par les deux types de machines de Turing sont les mêmes, les efficacités en temps de calcul semblent très différents. Ceci est le fameux problème $P = NP$ dont nous parlerons un peu plus loin.

3.4 Les machines à accès direct

Quand on cherche un type de machine mathématique qui représente de manière satisfaisante les ordinateurs, les machines de Turing nous laissent un peu sur notre faim, et pourtant elles possèdent, en germe, une organisation déjà proche des ordinateurs. Nous décrivons ici succinctement les *machines à accès direct* (RAM, *random access machine*). Ces machines reflètent de manière plus réaliste les composants essentiels d'un ordinateur, tout en étant d'un maniement simple qui permet notamment des mesures de complexité.

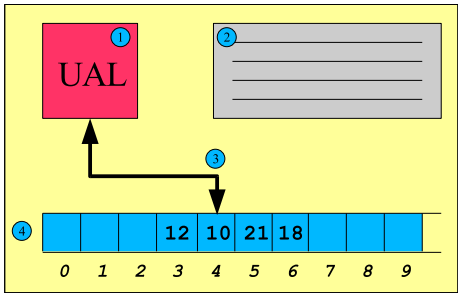

Figure 5.— Structure d'une machine à accès direct. (1) l'unité arithmétique, (2) le programme, (3) le dispositif d'échange entre unité centrale et mémoire et (4) la mémoire à cellules adressables.

Une machine à accès direct diffère d'une machine de Turing essentiellement par le fait que les données élémentaires (contenu des cellules de mémoire, état de l'unité de contrôle) sont des nombres, et non pas des symboles pris dans un ensemble fini. Une machine à accès direct possède une *unité arithmétique et logique*, une *mémoire* organisée linéairement dans laquelle chaque cellule est directement adressable, un *mécanisme d'échange* entre l'unité de calcul et la mémoire, et enfin elle est dirigée par un *programme*. Les instructions du programme sont plus élaborées et se rapprochent d'un assembleur.

4 Problèmes indécidables

Un *problème* est composé de deux parties :

- une description d'une représentation finie des éléments d'un ensemble dénombrable (les données du problème) ;
- un énoncé portant sur les éléments de cet ensemble, énoncé qui peut être vrai ou faux selon l'élément de cet ensemble.

Une *instance* d'un problème est un exemplaire des données du problème. Un problème est *décidable* s'il existe un algorithme pour le résoudre. Dans notre formalisme, un algorithme est réalisé par une machine de Turing. Un problème est donc *décidable* s'il existe une machine de Turing (déterministe) qui s'arrête toujours, et qui s'arrête dans une configuration d'acceptation si et seulement si l'énoncé est vrai sur la donnée (l'instance) fournie. Le problème est *indécidable* sinon.

De la même manière, une fonction partielle f est *calculable* s'il existe une machine de Turing (déterministe) qui s'arrête toujours, et qui pour la donnée x du domaine de f, s'arrête dans une configuration d'acceptation avec $f(x)$ sur la bande.

Commençons par un exemple concret et simple. On l'appelle le *problème du zéro dans les matrices carrées d'ordre* 3. Une donnée pour ce problème est un ensemble fini $\mathcal{M} = \{M_1, \ldots, M_r\}$ de matrices carrées d'ordre 3 à coefficients entiers relatifs. La question posée est : « existe-t-il une matrice $M = M_{i_1} M_{i_2} \cdots M_{i_n}$, produit de matrices de l'ensemble $\mathcal{M}$ dont l'élément d'indice $3, 2$ est nul » ?

La description formelle des données est composée de l'ensemble fini de matrices $\mathcal{M} = \{M_1, \ldots, M_r\}$. Une représentation sur une bande de machine de Turing peut prendre la forme d'une suite de $9r$ nombres représentant les éléments des r matrices. Trouver un algorithme qui résout le problème revient à décrire une machine de Turing déterministe qui s'arrête sur chacune de ces données, soit dans un état d'acceptation soit dans un autre état, selon qu'il existe une matrice M qui est un produit de matrices de $\mathcal{M}$ dont l'élément d'indice $3, 2$ de la matrice est nul ou non. Le problème des zéros de matrices d'ordre 3 est, aussi surprenant que cela puisse paraître, un exemple de problème indécidable.

Comment procéder pour démontrer qu'un problème est indécidable ? On procède en fait par *réduction effective*. Soient P et Q deux problèmes. On dit que P se réduit effectivement en Q s'il existe un algorithme (une machine de Turing) f qui transforme toute instance p de P en une instance $q = f(p)$ de Q telle que p est vrai si et seulement si $q = f(p)$ est vrai. Si P est indécidable et si P se réduit algorithmiquement à Q, alors Q est indécidable. En effet, si Q était décidable, on aurait l'algorithme suivant pour $p \in P$: transformer p effectivement en $q = f(p)$ et décider si q est vrai.

Si Q est un problème dont on veut prouver qu'il est indécidable, il suffit de trouver un problème P dont on sait qu'il est indécidable, et de construire une réduction effective de P en Q. Pour pouvoir amorcer ce procédé, il faut au moins un problème dont on montre directement qu'il est indécidable. C'est le *problème de l'arrêt des machines de Turing*. Les données de ce problème sont les couples formés d'une machine de Turing $\mathcal{T}$ et d'une donnée x. La question posée est : « est-ce que $\mathcal{T}$ s'arrête sur la donnée x » ? On prouve (voir l'encadré plus bas) que ce problème est indécidable.

À titre d'exemples, voici trois autres problèmes indécidables :

Le *problème de correspondance de Post* est le suivant. Les données sont les paires composées de deux suites $(x_1, \ldots, x_r)$ et $(y_1, \ldots, y_r)$ de mots. La question posée pour une paire est la suivante : existe-t-il une suite $i_1, \ldots, i_n$ d'indices tels que $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$. Le problème de correspondance de Post est indécidable. Il est très largement utilisé en théorie des langages formels, en combinatoire et ailleurs pour démontrer que des problèmes spécifiques de ces disciplines sont indécidables.

Un autre problème, encore plus simple à énoncer, est le *problème de l'associativité* d'une opération. Soit E un ensemble muni d'une loi de composition binaire. La question posée est : est-ce que cette loi est associative ? Ce problème est indécidable. Bien entendu, dans certains cas particuliers, on sait décider de cette propriété. Par exemple si l'ensemble E est fini, un test exhaustif permet de répondre à la question. Par exemple encore, si E est l'ensemble des entiers et que l'opération considérée est l'addition (ou la multiplication), on sait aussi répondre à la question. Ce qui est affirmé ici, c'est qu'il n'existe pas de méthode générale uniforme valable dans tous les cas. Il faut noter que ce résultat d'indécidabilité ne provient pas d'éventuelles caractéristiques « exotiques » de l'ensemble E ou de l'opération binaire : il peut même être établi quand E est l'ensemble des entiers naturels et pour des opérations binaires effectives (c'est-à-dire telles qu'on sait calculer effectivement le résultat de l'opération binaire sur deux entiers).

Parmi les problèmes indécidables figure le célèbre *dixième problème de Hilbert*, formulé par Hilbert dans son programme de recherche pour le XX^e siècle : il est indécidable de savoir si un polynôme à coefficients entiers en plusieurs variables possède une racine dont les composantes sont des entiers.

Voici une preuve, un peu technique, du fait que le problème de l'arrêt des machines de Turing est indécidable.
Toute machine de Turing $\mathcal{T}$ possède une description finie, donnée par la suite de ses instructions. Cette description peut être codée dans un mot $w(\mathcal{T})$ sur un alphabet fini. D'autre part, on peut construire une machine de Turing dite machine de Turing *universelle* qui « comprend » les codes des machines de Turing, au sens suivant : elle prend en argument le code $w(\mathcal{T})$ et une donnée x, et elle réalise le calcul de $\mathcal{T}$ sur la donnée x.
Ainsi, la machine universelle s'arrête pour $w(\mathcal{T})$ et x si et seulement si $\mathcal{T}$ s'arrête sur x, et dans ce cas, le résultat est le résultat de $\mathcal{T}$ sur x. En revanche, il n'existe pas de machine de Turing qui s'arrête toujours sur $w(\mathcal{T})$ et la donnée x, et qui accepte le couple $(w(\mathcal{T}), x)$ si et seulement si $\mathcal{T}$ s'arrête sur x. En d'autres termes, le problème de l'arrêt des machines de Turing est indécidable.

4.1 Langages récursifs et récursivement énumérables

Rappelons qu'une machine de Turing $\mathcal{T}$ avec états d'acceptation peut avoir le comportement suivant pour une donnée x :

– aucun calcul pour x ne s'arrête ;

- il existe un calcul qui s'arrête pour x, mais aucun des calculs qui s'arrêtent ne s'arrête dans un état d'acceptation ;
- il existe un calcul qui s'arrête pour x dans un état d'acceptation.

Le mot x est reconnu par la machine $\mathcal{T}$ dans le troisième cas seulement. Le *langage reconnu* est l'ensemble des mots reconnus. Dans le cas d'une machine déterministe, les trois cas s'énoncent plus simplement : le calcul pour x

- ne s'arrête pas ;
- s'arrête dans état qui n'est pas un état d'acceptation ;
- s'arrête dans un état d'acceptation.

On peut prouver qu'un langage reconnu par une machine de Turing peut aussi être reconnu par une machine de Turing déterministe.

Un langage est *récursivement énumérable* s'il est reconnu par une machine de Turing. La terminologie est due à la propriété suivante. Considérons une machine de Turing qui possède une bande de sortie, et qui se contente d'écrire, sur cette bande, des mots, séparés par un marqueur. Une telle machine *énumère* dans un certain ordre, le langage constitué des mots qu'elle écrit. On peut prouver qu'un langage est récursivement énumérable précisément quand il peut être énuméré par une machine de Turing.

Un langage est *récursif* s'il est reconnu par une machine de Turing qui s'arrête toujours. Il est facile de voir que le complémentaire d'un langage récursif l'est aussi. Il est aussi facile de vérifier que si un langage et son complémentaire sont récursivement énumérables, ils sont tous deux récursifs. En revanche, le complémentaire d'un langage récursivement énumérable ne l'est pas toujours.

5 Mesures de complexité et problèmes infaisables en pratique

La *mesure en temps* la plus simple d'un calcul qui se termine est le nombre de ses transitions. Ceci revient à assimiler chaque transition à une opération élémentaire, et à les considérer comme de même difficulté. On parle de *coût uniforme*. Cette approche n'est pas toujours réaliste dans des modèles comme les RAM où le contenu d'une cellule de mémoire peut être arbitrairement grand. Quand la taille des données est prise en compte pour une opération élémentaire, le coût peut ne plus être constant. De même, la *place* prise par un calcul est le nombre de cellules qui ont été modifiées par le calcul. On ne compte pas, dans cette évaluation, la place prise par les données ou les résultats pourvu que cette zone soit clairement séparée du reste.

On dit qu'une machine termine en temps (ou en place) $f(n)$ si, pour toute donnée x de longueur n pour laquelle il existe un calcul qui se termine, il en existe un dont le temps (la place) est majoré par $f(n)$. Un langage L est dit de complexité $f(n)$ s'il existe une machine de Turing de complexité $f(n)$ qui l'accepte en temps ou en place $f(n)$. Les langages de même complexité forment une classe de complexité. Les classes sont différentes pour les machines de Turing ou non déterministes. Ces dernières sont précédées de la lettre « N » dans la hiérarchie suivante. Les plus fréquentes sont mesurées par les logarithmes (LOG), les polynômes (P) ou les exponentielles (EXP), avec TIME pour le temps et SPACE pour l'espace.

$$\begin{aligned}&\mathrm{LOGSPACE} \subseteq \mathrm{NLOGSPACE}\\ &\subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE}\\ &\subseteq \mathrm{EXPTIME} \subseteq \mathrm{NEXPTIME}\\ &\subseteq \mathrm{EXPSPACE} = \mathrm{NEXPSPSPACE}\end{aligned}$$

Par exemple, la classe P (resp. NP) est formée des langages qui sont reconnaissables en temps polynomial par des machines de Turing déterministes (resp. non déterministes). On ne sait pas si les inclusions sont strictes ; en particulier, le fameux problème $P = NP$ est toujours ouvert.

Pour classer les problèmes dans une classe de complexité, on utilise le concept de réduction déjà évoqué, sauf que la réduction doit être rapide. Intuitivement, un problème P se réduit à Q si P est plus simple que Q parce qu'un algorithme pour Q se traduit en un algorithme pour P avec un surcoût négligeable.

Plus précisément, soient P et Q deux problèmes. On dit que P *se réduit en temps polynomial* en Q s'il existe une machine de Turing $\mathcal{M}$ qui transforme toute instance p de P en une instance $q = \mathcal{M}(p)$ de Q en temps polynomial en fonction de la taille de p et telle que p est vrai si et seulement si $q = \mathcal{M}(p)$ est vrai. Si P est dans l'une des classes de complexité ci-dessus et si P se réduit en temps polynomial à Q, alors Q est la même classe.

Les classes ci-dessus sont fermées par réductions polynomiales (sauf pour LOG). En effet, pour savoir si p dans P est vrai, on transforme p en temps polynomial en $q = \mathcal{M}(p)$ et on décide si q est vrai dans la classe de Q.

Un problème P est *difficile* pour une classe de complexité si tout problème de cette classe se réduit polynomialement à P. Un problème P est *complet* pour une classe de complexité s'il est membre de cette classe et s'il est difficile pour cette classe.

Les problèmes NP-complets sont très répandus dans toute l'algorithmique. Un problème NP-difficile est souvent réputé être *infaisable*. De manière intuitive, un problème est dans la classe NP s'il existe un algorithme (au sens usuel) déterministe en temps polynomial qui permet de *tester* si une solution x proposée pour une instance p du

problème en est effectivement une. Rien n'est dit sur la façon de trouver cette solution. En revanche, le problème est dans la classe P si on peut *trouver* une solution en temps polynomial.

5.1 Exemples de problèmes NP-complets

Pour prouver qu'un problème Q est NP-complet, il suffit de prouver qu'il est dans la classe NP, et qu'il existe un problème P dont on sait déjà qu'il est NP-complet et qui se réduit polynomialement à Q. Comme pour les problèmes indécidables, il convient d'amorcer le processus par un problème originel. Il s'agit traditionnellement du *problème de satisfaisabilité des formules booléennes* : une instance de ce problème est décrite par n variables booléennes $x_1, \ldots, x_n$, et une formule booléenne ϕ en ces variables. La question posée est : existe-t-il une affectation des valeurs vrai et faux aux variables (une « assignation » de valeurs de vérité) telle que ϕ prenne la valeur vrai ? Une telle assignation, si elle existe, *satisfait* la formule.

Il y a 2^n assignations possibles, et on ne connaît pas à ce jour d'algorithme déterministe en temps polynomial pour trouver une assignation qui satisfasse la formule. En revanche, en présence d'une assignation, il est très facile de vérifier en temps linéaire en fonction de la taille de la formule si elle la satisfait.

De nombreux problèmes, dans tous les domaines de l'informatique ou des mathématiques, se sont avérés être NP-complets. Voici deux exemples simples.

Le *problème du sac à dos* consiste à remplir un récipient, le sac à dos, avec des objets qui ont chacun une taille et une valeur, de sorte que la valeur des objets contenus dans le sac à dos soit aussi grande que possible. De manière précise, on se donne un ensemble X de n objets, chaque objet x étant de taille m_x et de valeur v_x. La capacité du sac à dos est c et la valeur totale à atteindre est b. La question est la suivante : existe-t-il un ensemble $Y \subset X$ d'objets tels que la somme des tailles des éléments de Y soit au plus c, et leur valeur au moins b, c'est-à-dire tel que $\sum_{x \in Y} m_x \leq c$ et $\sum_{x \in Y} v_x \geq b$. Ce problème est NP-complet.

Le *problème du coloriage* d'un graphe est le suivant : un coloriage d'un graphe G est une affectation de « couleurs » (représentées par des entiers) aux sommets du graphe telle que deux sommets adjacents soient de couleurs différentes. Le nombre chromatique d'un graphe est le plus petit nombre de couleurs nécessaires pour colorier un graphe. La question posée est : quel est le nombre chromatique d'un graphe G ? Cette question semble de nature diffférente, puisque la réponse n'est pas oui ou non. En fait, elle revient à la succession de questions : est-ce que G est coloriable avec 1, 2, ... couleurs ? On s'arrête dès que la réponse est positive, et on sait qu'elle sera positive au plus tard quand le nombre de couleurs est égal au nombre de sommets du graphe. Ce problème est aussi NP-complet.

5.2 Une remarque finale

Il faut faire attention à une source possible de confusion entre les diverses classifications de problèmes que nous avons introduites. Un problème *indécidable* est un problème pour lequel il n'existe pas d'algorithme, un problème *NP-difficile* que nous avons appelé aussi *infaisable* est un problème pour lequel aucun algorithme polynomial n'est connu, il est *NP-complet* si de plus on connaît un algorithme de vérification polynomial. On constate parfois une confusion avec ce que l'on appelle l'*explosion combinatoire*. Ce phénomène se produit quand le nombre de situations à envisager pour un problème croît exponentiellement. Ce phénomène ne préjuge en rien de la décidabilité du problème : ainsi, le nombre de cas à envisager pour le jeu d'échec est énorme, mais comme il est fini, il « suffit » en théorie de les considérer tous. L'explosion combinatoire est un signe que le problème peut être NP-difficile, mais n'en est pas une preuve ; en effet, rien n'indique que, parmi le nombre exponentiel de cas possible, un argument simple ne permette de les écarter tous à l'exception de quelques-uns.

6 Automates à pile

Des nombreuses variantes des machines de Turing sont définies par des restrictions des possibilités de calcul. Ces restrictions portent notamment sur la gestion de l'accès aux données. Nous avons déjà parlé des données en ligne et hors ligne ; d'autres restrictions ont été introduites sur la taille de la mémoire auxiliaire (comme les automates à mémoire linéairement bornée ou LBA évoqués plus loin) et la façon d'y accéder. Les automates à pile sont de cette nature. Nous les décrivons comme des machines de Turing à capacité restreinte, mais une définition intrinsèque, sans référence aux machines de Turing, est très courante. Nous la donnons plus loin.

Le concept d'automate à pile est fondamental en informatique puisqu'il modélise le mécanisme de fonctionnement d'une pile. À ce titre, un automate constitue la description itérative d'un algorithme récursif, et ce mécanisme est à la base de toute compilation d'un langage évolué vers un langage rudimentaire.

6.1 Une première définition

Un *automate à pile* est une machine de Turing à deux bandes avec quelques modifications et restrictions. D'abord, nous autorisons, en plus

des mouvements vers la gauche et vers la droite, une transition où la tête reste sur la cellule courante. Ce « mouvement » est noté par la lettre I, pour « immobile ». Ensuite, les deux bandes sont limitées comme suit. La première bande, de lecture, a une tête qui se déplace vers la droite ou reste immobile sur une cellule. La deuxième bande, de travail, est organisée en pile. Les modifications ne sont possibles qu'à l'une des extrémités de la bande (à droite pour fixer les idées), et la tête de lecture se trouve au-dessus de la première cellule inoccupée.

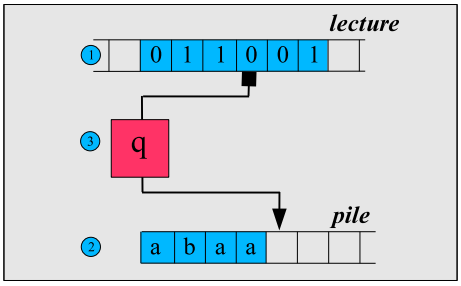

Figure 6.— Un automate à pile ; (1) bande de lecture, (2) bande organisée en pile, (3) dispositif de contrôle.

Les deux opérations importantes sont *l'empilement d'une donnée* et le *dépilement*. L'empilement du symbole a se fait par l'écriture du symbole a dans la cellule vide, suivie du déplacement de la tête vers la droite. Le dépilement se fait en deux étapes : d'abord un déplacement vers la gauche, puis un remplacement du symbole dans la cellule par le blanc. Dans la *configuration initiale*, la tête de la première bande se trouve sur la première lettre de l'entrée, et la bande de travail est vide. Dans une configuration d'*acceptation*, la tête de lecture est à droite du mot d'entrée, et la machine est dans un état d'acceptation sans pouvoir continuer le calcul.

Un mot est reconnu s'il existe un calcul pour ce mot conduisant de la configuration initiale à une configuration d'acceptation. Un langage reconnu par un automate à pile est appelé *context-free* ou *algébrique*. Ces langages se classent, comme on le verra, entre les langages récursivement énumérables et les langages réguliers.

Considérons, comme exemple typique, les systèmes de parenthésages corrects. Étant donné des paires de parenthèses, comme $(,)$ et $[,]$, on forme des mots qui sont correctement parenthésés, et où la parenthèse initiale correspond à la parenthèse finale. Les mots les plus courts sont $()$, $[]$, $([])$, $[()]$, $(())$, $[[]]$. D'autres mots, plus complexes, sont $(([])())[])$ ou $(([])())[((()))([])]$. Parfois, on écrit a ou b pour une parenthèse ouvrante, et $\bar{a}$ ou $\bar{b}$ pour une parenthèse fermante. Ainsi, le mot précédent s'écrit aussi $aab\bar{b}a\bar{a}\bar{a}baaa\bar{a}\bar{a}\bar{a}ab\bar{b}\bar{a}\bar{b}\bar{a}$. Les systèmes de parenthésages corrects sont appelés *mots de Dyck premiers*, les produits de mots de Dyck premiers sont les *mots de Dyck* (tout court). Par exemple, $()[]([])$ est un mot de Dyck, produit de trois mots de Dyck premiers. Le *langage de Dyck* est l'ensemble des mots de Dyck.

$$
\begin{array}{lcl}
(i, a, \square) & \rightarrow & (D, \bot, D, j) \\
(i, b, \square) & \rightarrow & (D, \square, I, e) \\
(j, a, \square) & \rightarrow & (D, a, D, j) \\
(j, \bar{a}, \square) & \rightarrow & (I, \square, G, t) \\
(t, \bar{a}, a) & \rightarrow & (D, \square, G, j) \\
(t, \bar{a}, \bot) & \rightarrow & (D, \square, I, f) \\
(f, a, \square) & \rightarrow & (D, \square, I, e) \\
(f, \bar{a}, \square) & \rightarrow & (D, \square, I, e)
\end{array}
$$

Ci-dessus les instructions de l'automate qui reconnaît les mots de Dyck premiers. Ici, i est l'état initial, f est l'état d'acceptation et e est un état d'échec. Pour la bande d'entrée, on écrit seulement le mouvement (D ou I).

Le calcul d'un automate à pile peut être vu comme un parcours d'arbre (voir le chapitre[Lévy]££). Quand on empile, on descend dans l'arbre ; quand on dépile, on remonte. Le symbole empilé dépend d'une information locale sur l'arbre (étiquette des nœuds, des arêtes, etc). Quand on dépile, on contrôle essentiellement une correspondance entre le résultat du parcours du sous-arbre que l'on vient de quitter et l'information au point de retour.

Comme illustration la plus simple, considérons un arbre quelconque dont les nœuds sont muets. Notons a une descente et $\bar{a}$ une remontée. Le parcours d'un tel arbre est un mot du langage de Dyck et, réciproquement, à chaque mot de Dyck correspond un tel arbre que l'automate à pile « décrit ».

6.2 Une autre définition

Il existe plusieurs variations sur les automates à pile. La définition la plus courante, et que nous présentons maintenant, intègre, dans une transition, une étape de dépilement et plusieurs étapes d'empilement. Nous introduisons d'abord les machines à pile, et ensuite les automates à pile. Les automates à pile sont des machines à pile dotées d'une condition d'acceptation.

Une *machine à pile* est composée d'une unité centrale, dont la configuration est symbolisée par un état, et de deux bandes : l'une contient un mot à analyser et est lue de gauche à droite ; l'autre est organisée en pile, et sert à contenir de l'information auxiliaire, en quantité non bornée. Une *configuration* de la machine est un couple (q, h) formé de l'état courant q et du contenu h de la pile.

Les *transitions* d'une machine à pile sont décrites par un nombre fini de *règles*. Chaque règle

a la forme $(p, z, s) \to (d, q)$, et signifie que si la machine est dans l'état p, que le symbole vu du mot d'entrée est s (lettre ou mot vide), et que la lettre au sommet de la pile est z, la machine peut lire s, effacer z et le remplacer par le mot d, et passer dans l'état q. Une transition de la machine représente donc le passage d'une configuration à une autre. Comme chaque écriture est précédée d'un effacement, la machine se bloque lorsque la pile est vide, puisqu'alors il n'y a rien à effacer.

Chaque effacement est suivi de l'écriture d'un mot d. Si ce mot commence par la lettre que l'on vient d'effacer, c'est comme si on n'avait pas dépilé. Si le mot d est vide, cela revient à dépiler seulement. Il est possible, dans ce modèle, d'empiler plusieurs lettres en une seule étape. Le symbole « lu » sur la bande d'entrée peut être le mot vide. Dans ce cas, cela signifie que la tête de lecture n'avance pas.

Enfin, notons qu'il est fort possible que plusieurs règles s'appliquent à une configuration donnée ; dans ce cas, la machine n'est pas déterministe. Pour que la machine à pile soit *déterministe*, il faut bien sûr que pour chaque état p, pour chaque symbole s du mot d'entrée et pour chaque symbole de pile z, on n'ait jamais deux résultats possibles ; mais il faut aussi que si la règle (p, ϵ, z) est définie, il n'y ait aucune règle de la forme (p, s, z) pour s non vide : si une règle sans lecture est possible, c'est la seule.

Un *automate à pile* est une machine à pile munie d'un ensemble de *configurations terminales* T. Un calcul est *réussi* pour x s'il passe de la configuration initiale à une configuration terminale en lisant x. Un mot x est *reconnu* ou *accepté* par l'automate à pile si x est l'étiquette d'un calcul réussi.

Les trois modes d'acceptation les plus fréquents sont par pile vide (une configuration est terminale si la pile est vide), par états terminaux (une configuration est terminale si son état est terminal), ou par pile vide et états terminaux (c'est la conjonction des deux conditions). Les trois modes d'acceptation sont en général équivalents. Ce n'est plus le cas pour les automates à pile déterministes : la reconnaissance par pile vide est moins puissante que celle par états terminaux, alors que la reconnaissance pile vide et états terminaux équivaut à celle par pile vide.

$$\begin{array}{lcl} (a, \#) & \to & \bot \\ (a, \bot) & \to & \bot\alpha \\ (a, \alpha) & \to & \alpha\alpha \\ (\bar{a}, \alpha) & \to & \varepsilon \\ (\bar{a}, \bot) & \to & \varepsilon \end{array}$$

Ci-dessus, les règles pour un automate reconnaissant l'ensemble des mots de Dyck premiers sur une paire de parenthèses, notées $a, \bar{a}$, et où la reconnaissance se fait par pile vide. L'automate a un seul état que nous n'écrivons pas. Les symboles écrits sur la pile sont des α. Cet automate à pile est déterministe.

Le deuxième exemple est l'ensemble des mots de Dyck sur deux paires de parenthèses que nous notons ici par $a, \bar{a}, b, \bar{b}$. L'automate a deux états i et f, et les règles données ci-dessous. La dernière règle s'interprète comme suit : lorsque sur la pile se trouve le symbole de fond de pile $\#$, alors sans lecture on passe dans l'état f.

Cet automate à pile n'est pas déterministe : devant le fond de pile, on peut lire a ou b, mais aussi ne rien lire. On devine ici le rôle parfois gênant des transitions sans lecture. On peut toujours se passer des règles sans lecture dans les automates à pile non déterministes ; mais les automates à pile déterministes, qui n'ont pas de transitions sans lecture sont moins puissants que ceux qui les admettent.

$$\begin{array}{lcl} (i, a, \#) & \to & (\#\alpha, i) \\ (i, b, \#) & \to & (\#\beta, i) \\ (i, a, \alpha) & \to & (\alpha\alpha, i) \\ (i, a, \beta) & \to & (\beta\alpha, i) \\ (i, b, \alpha) & \to & (\alpha\beta, i) \\ (i, b, \beta) & \to & (\beta\beta, i) \\ (i, \bar{a}, \alpha) & \to & (\varepsilon, i) \\ (i, \bar{b}, \beta) & \to & (\varepsilon, i) \\ (i, \varepsilon, \#) & \to & (\varepsilon, f) \end{array}$$

6.3 Automates à pile et grammaires algébriques

Une *grammaire algébrique* ou *context-free* est une description d'un procédé de construction ou de génération de mots, alors qu'un automate à pile est un procédé de reconnaissance ou d'analyse.

Une grammaire sur un alphabet A est composée d'un ensemble de *variables* et d'un ensemble fini de *règles* de la forme $X \to w$, où X est une variable. On *dérive* en remplaçant successivement une occurrence d'une variable X par un membre droit de règles. On représente ce fait par un arbre de dérivation : pour une règle $X \to w$, la variable X étiquette un nœud et w est la suite des étiquettes des nœuds enfants de X.

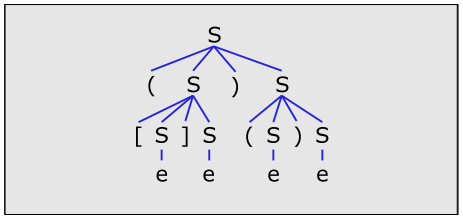

Figure 7.— L'arbre de dérivation du mot de Dyck ([])()

Un mot x sans variables est *engendré* par la grammaire s'il existe un arbre de dérivation dont le mot des feuilles est x et la racine a pour étiquette une variable distinguée appelée l'*axiome* de la grammaire.

Considérons par exemple la grammaire suivante pour le langage de Dyck sur deux paires de parenthèses. Elle possède une unique variable S et les trois règles (le $+$ désigne le « ou ») :

$$S \to (S)S + [S]S + \varepsilon\,.$$

Le mot $([])()$ dérive de l'axiome S dans cette grammaire, comme illustré par l'arbre donné dans la figure 7.

6.4 Propriétés décidables des langages algébriques

Les langages algébriques ou *context-free* jouissent d'un certain nombre de problèmes décidables. Le plus important est le test d'appartenance : il s'agit, pour un langage L donné, de décider si un mot w est dans L. Cette propriété est indécidable pour les langages récursivement énumérables, et devient décidable pour les langages context-free. Il revient au même, pour un mot w donné, de construire effectivement un arbre de dérivation pour w dans une grammaire de L. Cette propriété est à la base de l'analyse syntaxique, qui à son tour est la première étape d'une compilation d'un programme. Sans la décidabilité du test d'appartenance, les langages de programmation ne pourraient être décrits par des grammaires context-free. On peut aussi décider, pour un langage context-free, s'il est vide ou non ; cette propriété est indécidable dans les langages récursivement énumérables. En revanche, l'égalité de deux langages context-free, ou la régularité d'un langage context-free, sont indécidables. Le complémentaire d'un langage context-free n'est pas toujours context-free. En revanche, le complémentaire d'un langage context-free *déterministe* est aussi context-free déterministe.

On sait depuis assez longtemps décider si le langage reconnu par un automate à pile déterministe est rationnel ou non. Récemment, G. Sénizergues (Senizergues, 2002) a démontré que l'égalité est décidable pour les langages context-free déterministes. Ce problème est resté ouvert pendant près de quarante ans (une éternité en informatique), et sa résolution a valu à son auteur le prix Gödel en 2002.

Les différences évoquées ci-dessus entre les automates à pile déterministes et non déterministes montrent que les premiers reconnaissent une sous-famille stricte de celle reconnue par les seconds. Le plus souvent, l'analyse syntaxique d'un langage de programmation concret est assise sur l'existence d'une automate à pile déterministe reconnaissant les programmes syntaxiquement corrects.

7 Classification de Chomsky des langages formels

Noam Chomsky a proposé, dans les années soixante, une classification des langages en quatre catégories, appelés langages de type 0, 1, 2 et 3 basée sur des grammaires plus générales. Une telle grammaire est formée d'un ensemble fini de règles de dérivation $u \to v$. Certaines lettres sont des variables. Une étape de dérivation consiste à remplacer, dans un mot x, une occurrence d'un facteur u par le facteur v. On itère depuis un mot de départ jusqu'à l'obtention d'un mot sans variable.

Le critère employé pour classer les langages est basé sur la forme des règles : elles sont sans contrainte pour les langages de type 0, de la forme $gXd \to gwd$ pour une variable X dans les langages de type 1, de la forme $X \to w$ pour le type 2 et de la forme $X \to wY$, où X, Y sont des variables et w ne contient pas de variables dans le type 3. Il est remarquable que les langages de type 0 sont exactement les langages récursivement énumérables, ceux de type 2 les langages algébriques, ceux de type 3 les langages réguliers. Les langages de type 1, appelé *context-sensitive*, sont reconnus par les machines de Turing dont la mémoire est de taille linéairement bornée en fonction de la longueur de l'entrée (d'où le nom LBA pour *linear bounded automata*).

Cette classification montre une concordance satisfaisante entre les divers procédés employés dans l'informatique théorique pour cerner les modèles de calcul et de reconnaissance. On peut en conclure que les classes de langages et de machines obtenues correspondent à un phénomène naturel, et que l'on dispose d'outils pour le manier.

8 Pour en savoir plus

Il existe de nombreux ouvrages, en français, sur les machines de Turing, la calculabilité, la complexité et les langages formels. Nous avons utilisé ici le livre (Autebert, 1992), et la présentation (van Emde Boas, 1990). Une description brève est donnée dans (Wolper, 2001). Les ouvrages (Salomaa, 1989) et (Floyd et Biegel, 1995) sont des traductions, le deuxième contient une description des RAM. Le dernier ouvrage en français est (Rey, 2004). Sur les langages formels, les livres abondent aussi. Outre le livre (Autebert, 1994), on peut consulter la synthèse (Autebert, Berstel et Boasson, 1997) et le livre (Hopcroft, Motwani et Ullman, 2001).

Bibliographie

Autebert J.-M. (1992), *Calculabilité et décidabilité.* Masson.

Autebert J.-M. (1994), *Théorie des langages et des automates.* Masson.

Autebert J.-M., Berstel J. et Boasson L. (1997), Context-free languages and pushdown automata. In Rozenberg G. et Salomaa A., éditeurs, *Handbook of Formal Languages*, pages 111–174. Springer-Verlag.

Floyd R. et Biegel R. (1995), *Le langage des machines.* Vuibert.

Hopcroft J., Motwani R. et Ullman J. (2001), *Introduction to Automata Theory, Languages and Computation.* Addison Wesley.

Rey J.-F. (2004), *Calculabilité, complexité et approximation.* Vuibert.

Salomaa A. (1989), *Introduction à l'informatique théorique. Calculabilité et complexité.* Armand Colin.

Senizergues G. (2002), $L(A) = L(B)$? A simplified decidability proof. *Theoret. Comput. Sci.*, 281 :555–608.

van Emde Boas P. (1990), Machine models and simulations. In van Leeuwen J., éditeur, *Algorithms and Complexity*, pages 1–66. Elsevier.

Wolper P. (2001), *Introduction à la calculabilité.* Dunod. 2e édition.

Index