Informatique Fondamentale

Ecole polytechnique

Jean-Jacques Lévy

Table des matières

Avant-propos

Ce polycopié est le support écrit du cours « Informatique Fondamentale » de deuxième année. Il porte sur les notions plus avancées de la programmation telles que les algorithmes sur les graphes, la modularité, la programmation par objets, la vérification de programmes, l'analyse syntaxique, l'exploration exhaustive et la concurrence. Bien sûr, chacune de ces notions ne sera vue que partiellement, mais l'ensemble de ces chapitres se veut représenter un bagage informatique minimal pour tout ingénieur moderne. Ce cours suppose connues des notions élémentaires comme la programmation impérative et itérative, ou récursive, manipulant des structures de données scalaires ou composites telles que les tableaux, les enregistrements, les listes ou les arbres.

Le langage de programmation choisi est Java. Nous essaierons de donner des programmes équivalents en Ocaml, et parfois en C. Le choix du langage est secondaire, car l'ensemble du cours est quasiment indépendant du langage. Au début, un style très pascalien de programmation impérative sera suivi. La programmation par objets ne sera développée qu'après quelques chapitres. Ce qui importe dans le choix du langage est d'abord d'avoir un langage fortement typé, et ensuite de disposer d'un système automatique de gestion de la mémoire, permettant de garder les notions développées à un bon niveau d'abstraction, sans ignorer les problèmes de complexité. Mais nous garderons à l'esprit quelques inconvénients dus au choix de Java : lourdeur, inefficacité, par exemple.

Les participants à ce cours ont donc grandement contribué à l'établissement de ce polycopié. Merci donc à François Anceau, Patrice Bertin, Gilles Dowek, Georges Gonthier, Daniel Krob, Philippe Chassignet, Frédéric Chyzak, Ian Mackie, Guillaume Poupard, Laurent Viennot, Dominique Rossin, Didier le Botlan, James Leifer, Alan Schmitt et Simon Patarin. Pour une partie, ce cours s'inspire de l'ancien cours de première année « Algorithmes et Programmation » effectué de 1992 à 2001 avec Robert Cori.

Cette première version du polycopié a bénéficié de la relecture et des nombreuses remarques de Philippe Chassignet, Robert Cori, Jean-Christophe Filliâtre, Georges Gonthier, Luc Maranget, Catuscia Palamidessi et Didier Rémy. Les assertions du chapitre 5 ont été mises au point grâce au programme *Why* de Jean-Christophe Filliâtre.

Enfin, si ce cours est entièrement disponible sur le Web, cela est dû à Luc Maranget et à son très efficace traducteur Hevea de format TeX en format Html. Le polycopié se trouve en

`http ://www.enseignement.polytechnique.fr/informatique/`

sous la rubrique « Informatique Fondamentale ».

Polycopié, version 1.0

Chapitre 1

Graphes

Les graphes sont des structures combinatoires rencontrées dans des applications faisant intervenir des mathématiques discrètes et nécessitant une solution informatique. Circuits électriques, réseaux de transport (ferrés, routiers, aériens), réseaux d'ordinateurs, ordonnancement d'un ensemble de tâches sont les principaux domaines d'application où la structure de graphe intervient. D'un point de vue formel, il s'agit d'un ensemble de points (sommets) et d'un ensemble d'arcs reliants des couples de sommets. Une des premières questions que l'on se pose est de déterminer, étant donné un graphe et deux de ses sommets, s'il existe un chemin (suite d'arcs) qui les relie ; cette question très simple d'un point de vue mathématique pose des problèmes d'efficacité dès que l'on souhaite la traiter à l'aide de l'ordinateur pour des graphes comportant un très grand nombre de sommets et d'arcs. Pour se convaincre de la difficulté du problème il suffit de considérer le jeu d'échecs et représenter chaque configuration de pièces sur l'échiquier comme un sommet d'un graphe, les arcs joignent chaque configuration à celles obtenues par un mouvement d'une seule pièce ; la résolution d'un problème d'échecs revient ainsi à trouver un ensemble de chemins menant d'un sommet à des configurations de *« mat »*. La difficulté du jeu d'échecs provient donc de la quantité importante de sommets du graphe que l'on doit parcourir. Des graphes plus simples comme celui des stations du Métro Parisien donnent lieu à des problèmes de parcours beaucoup plus facilement solubles. Il est courant, lorsque l'on étudie les graphes, de distinguer entre les graphes orientés dans lesquels les arcs doivent être parcourus dans un sens déterminé (de x vers y mais pas de y vers x) et les graphes symétriques (ou non-orientés) dans lesquels les arcs (appelés alors arêtes) peuvent être parcourus dans les deux sens. Nous nous limitons dans ce chapitre aux graphes orientés, car les algorithmes de parcours pour les graphes orientés s'appliquent en particulier aux graphes symétriques : il suffit de construire à partir d'un graphe symétrique G le graphe orienté G' comportant pour chaque arête x, y de G deux arcs opposés, l'un de x vers y et l'autre de y vers x.

1.1 Définitions

Dans ce paragraphe, nous donnons quelques définitions sur les graphes orientés et quelques exemples, nous nous limitons ici aux définitions les plus utiles de façon à passer très vite aux algorithmes.

Définition 1.1 *Un graphe $G = (X, A)$ est donné par un ensemble X de* sommets *et par un sous-ensemble A du produit cartésien $X \times X$ appelé l'ensemble des* arcs *de G.*

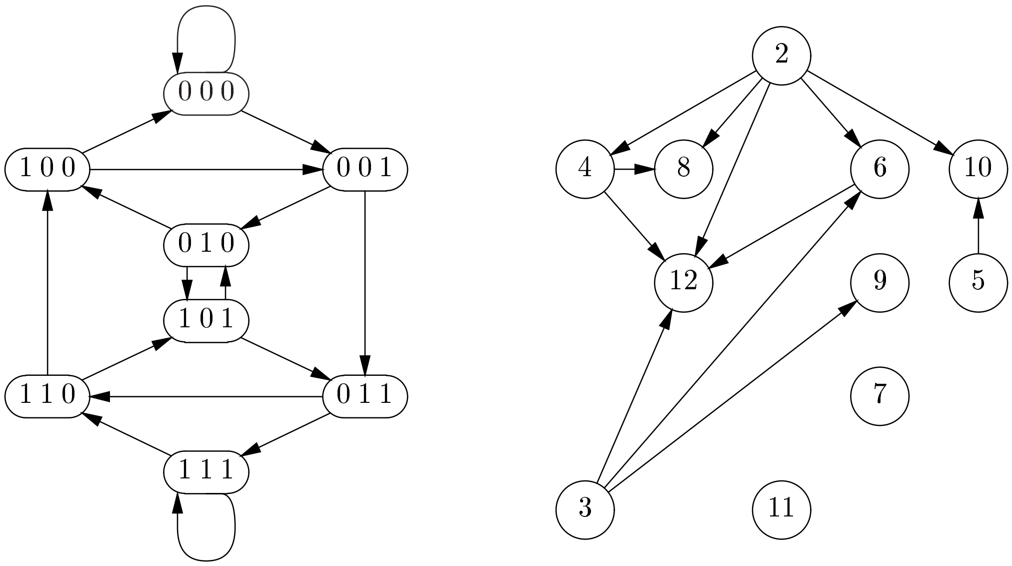


FIG. 1.1 – Graphe de De Bruijn pour $k = 3$; graphe des diviseurs pour $n = 12$

Un arc $a = (x, y)$ a pour *origine* le sommet x et pour *extrémité* le sommet y. On note

$$org(a) = x \quad ext(a) = y$$

Dans la suite, on suppose que tous les graphes considérés sont finis, ainsi X et par conséquent A sont des ensembles finis. On dit que le sommet y est un *successeur* de x si $(x, y) \in A$, on dit aussi que x est un *prédécesseur* de y.

Définition 1.2 *Un chemin f du graphe $G = (X, A)$ est une suite d'arcs $\langle a_1, a_2, \ldots, a_p \rangle$ telle que :*

$$\forall i, 1 \leq i < p \quad org(a_{i+1}) = ext(a_i)$$

L'*origine* d'un chemin f, aussi notée $org(f)$, est celle de son premier arc a_1 et son *extrémité*, notée $ext(f)$ est celle de son dernier arc a_p. La *longueur* du chemin est égale au nombre d'arcs qui le composent, c'est-à-dire p. Un chemin f tel que $org(f) = ext(f)$ est appelé un *circuit*.

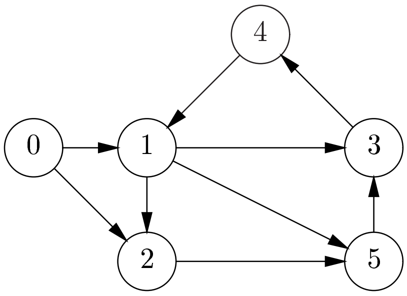
Exemple 1.1 *Graphes de De Bruijn.* Les sommets sont les suites de longueur k formées de symboles 0 ou 1, un arc joint la suite f à la suite g si $f = xh$ et $g = hy$ où x et y sont des symboles (0 ou 1) et où h est une suite quelconque de $k - 1$ symboles.

Exemple 1.2 *Graphes des diviseurs.* Les sommets sont les nombres $\{2, 3, \ldots, n\}$, un arc joint p à q si p divise q.

1.2 Matrices d'adjacence

Une structure de données simple pour représenter un graphe est la matrice d'adjacence M. Pour obtenir M, on numérote les sommets du graphe de façon quelconque :

$$X = \{x_1, x_2 \ldots x_n\}$$



$$M = \begin{pmatrix} F & V & V & F & F & F \\ F & F & V & V & F & V \\ F & F & F & F & F & V \\ F & F & F & F & V & F \\ F & V & F & F & F & F \\ F & F & F & V & F & F \end{pmatrix}$$

FIG. 1.2 – Un graphe avec sa matrice d'adjacence

M est une matrice booléenne carrée $n \times n$ dont les coefficients sont V et F telle que :

$$M_{i,j} = \begin{cases} V & \text{si} \quad (x_i, x_j) \in A \\ F & \text{sinon} \end{cases}$$

Ceci donne alors les déclarations de type et de variables suivantes :

```
class GrapheMat {
  boolean[ ][ ] m;   // la matrice M d'adjacence,
  GrapheMat (int n) {
      m = new boolean[n][n];
  }
}
```

La matrice d'adjacence aurait aussi pu être représentée par une matrice d'entiers (0 pour F, 1 pour V). Un intérêt de cette autre représentation est que la détermination de chemins dans G revient au calcul des puissances successives de la matrice M comme le montre le théorème suivant.

Théorème 1.3 *Soit M^p la puissance p-ième de la matrice M, le coefficient $M^p_{i,j}$ est égal au nombre de chemins de longueur p de G dont l'origine est le sommet x_i et dont l'extrémité est le sommet x_j.*

Démonstration Par récurrence sur p. Pour $p = 1$, le résultat est immédiat car un chemin de longueur 1 est un arc du graphe. Pour $p > 1$, le calcul de M^p donne :

$$M^p_{i,j} = \sum_{k=1}^{n} M^{p-1}_{i,k} M_{k,j}$$

Or tout chemin de longueur p entre x_i et x_j se décompose en un chemin de longueur $p-1$ entre x_i et un certain x_k suivi d'un arc reliant x_k et x_j. Le résultat découle alors de l' hypothèse de récurrence suivant laquelle $M^{p-1}_{i,k}$ est le nombre de chemins de longueur $p-1$ joignant x_i à x_k. $\square$

De ce théorème, on déduit l'algorithme suivant permettant de tester l'existence d'un chemin entre deux sommets x et y :

```
static boolean existeChemin (GrapheMat g, int x, int y) {
  int n = g.m.length;
  boolean r[ ][ ] = new boolean[n][n];
  copie(r, m);
  for (int p = 1; !r[x][y] && p < n; ++p) {
    multiplier (r, r, m);
    additionner (r, r, m);
```

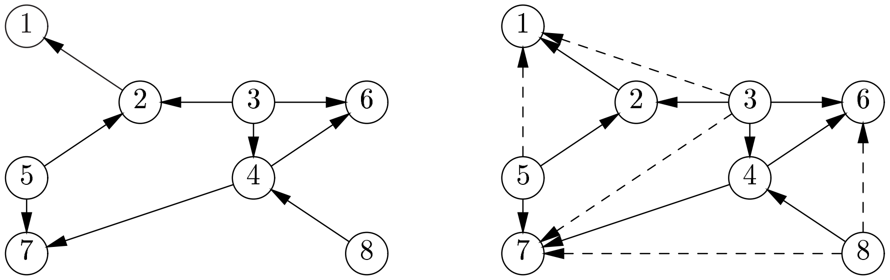


FIG. 1.3 – Un graphe et sa fermeture transitive

```
    }
    return r[x][y];
  }
```

Les fonctions *multiplier(r,a,b)* et *additionner(r,a,b)* sont respectivement des fonctions qui multiplient et ajoutent les deux matrices booléennes a et b (de dimension $n \times n$) en rangeant le résultat dans la matrice r. La fonction *copie(r,m)* fait une copie de m dans la matrice r.

Le nombre d'opérations effectuées par l'algorithme est de l'ordre de n^4, car le produit de deux matrices carrées $n \times n$ demande n^3 opérations et l'on peut avoir à effectuer n produits de matrices. La recherche du meilleur algorithme possible pour le calcul du produit de deux matrices a été très intense ces dernières années. Plusieurs améliorations de l'algorithme élémentaire demandant n^3 multiplications ont été successivement trouvées. Coppersmith et Winograd détiennent le record avec un algorithme en $O(n^{2.5})$; mais ce résultat reste très théorique, car la programmation de leur algorithme n'est pas aisée et l'efficacité espérée n'est atteinte que pour de très grandes valeurs de n. On cherche donc à construire d'autres algorithmes, faisant intervenir des notions différentes.

Enfin, notons que, si on se limite à la recherche de l'existence d'un chemin entre deux sommets x et y donnés, on peut ne calculer qu'une seule ligne de la matrice, ce qui diminue notablement la complexité.

1.3 Fermeture transitive

La fonction *existeChemin*, vue précédemment, ne fait rien de plus qu'un calcul naïf de la notion suivante : la fermeture transitive d'un graphe $G = (X, A)$ est la relation binaire transitive minimale contenant la relation A sur X. Il s'agit d'un graphe $G^* = (X, A^*)$ tel que $(x, y) \in A^*$ si et seulement s'il existe un chemin f dans G d'origine x et d'extrémité y.

Le calcul de la fermeture transitive permet donc de répondre aux questions concernant l'existence de chemins entre tout couple de sommets x et y dans G. On effectue un *pré-traitement* de G en calculant $G^* = (X, A^)$; puis on répond en temps constant, $O(1)$, à toute question sur l'existence de chemins entre x et y. Une autre application du calcul de la fermeture transitive est fréquente en compilation : un graphe est associé à chaque fonction d'un programme, les sommets de ce graphe représentent les variables de la fonction et un arc entre la variable a et la variable b indique que le calcul de la valeur a fait appel au calcul de la valeur de b. La fermeture transitive de ce *graphe de dépendances* donne alors toutes les variables intervenant dans le calcul de a.

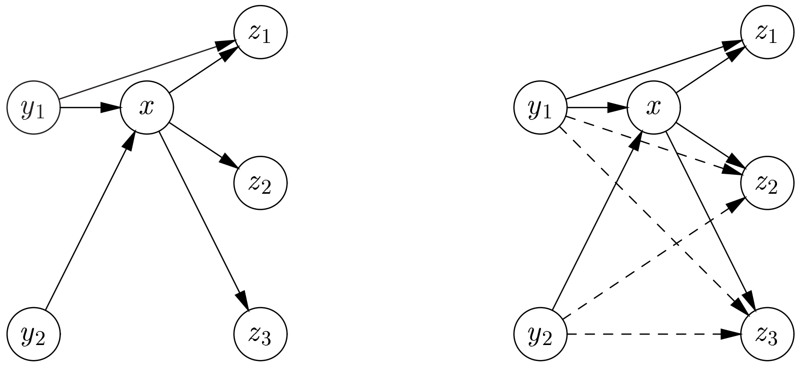


FIG. 1.4 – L'effet de l'opération Φ_x : les arcs ajoutés sont en pointillé

Le calcul de (X, A^*) s'effectue par itération de l'opération de base $\Phi_x(A)$ qui ajoute à A les arcs (y, z) tels que y est un prédécesseur de x et z un de ses successeurs. Posons :

$$\Phi_x(A) = A \cup \{(y, z) \mid (y, x) \in A, (x, z) \in A\}$$

Cette opération satisfait les deux propriétés suivantes :

Proposition 1.4 *Pour tout sommet x, on a*

$$\Phi_x(\Phi_x(A)) = \Phi_x(A)$$

et pour tout couple de sommets (x, y) :

$$\Phi_x(\Phi_y(A)) = \Phi_y(\Phi_x(A))$$

Démonstration La première partie est très simple, on l'obtient en remarquant que $(u, x) \in \Phi_x(A)$ implique $(u, x) \in A$ et que $(x, v) \in \Phi_x(A)$ implique $(x, v) \in A$.

Pour la seconde partie, il suffit de vérifier que si (u, v) appartient à $\Phi_x(\Phi_y(A))$ il appartient aussi à $\Phi_y(\Phi_x(A))$. Le résultat s'obtient ensuite par symétrie. Si $(u, v) \in \Phi_x(\Phi_y(A))$, alors ou bien $(u, v) \in \Phi_y(A)$, ou bien (u, x) et $(x, v) \in \Phi_y(A)$. Dans le premier cas, $\Phi_y(A') \supset \Phi_y(A)$ pour tout $A' \supset A$ implique $(u, v) \in \Phi_y(\Phi_x(A))$. Dans le second cas, il y a plusieurs situations à considérer suivant que (u, x) ou (x, v) appartiennent ou non à A ; l'examen de chacune d'entre elles permet d'obtenir le résultat. Examinons en une à titre d'exemple, supposons que $(u, x) \in A$ et $(x, v) \notin A$. Comme $(x, v) \in \Phi_y(A)$, on a $(x, y) \in A$ et $(y, v) \in A$. Ceci implique $(u, y) \in \Phi_x(A)$ et $(u, v) \in \Phi_y(\Phi_x(A))$. □

Proposition 1.5 *La fermeture transitive A^* est donnée par :*

$$A^* = \Phi_{x_1}(\Phi_{x_2}(\ldots \Phi_{x_n}(A) \ldots))$$

Démonstration On se convainc facilement que A^* contient l'itérée de l'action des Φ_{x_i} sur A, la partie la plus complexe à prouver est que $\Phi_{x_1}(\Phi_{x_2}(\ldots \Phi_{x_n}(A) \ldots))$ contient A^*. Pour cela on considère un chemin joignant deux sommets x et y de G alors ce chemin s'écrit

$$(x, y_1)(y_1, y_2) \ldots (y_p, y)$$

ainsi $(x, y) \in \Phi_{y_1}(\Phi_{y_2}(\ldots \Phi_{y_p}(A)\ldots))$ les propriétés démontrées ci-dessus permettent d'ordonner les y suivant leurs numéros croissants ; le fait que $\Phi_y(A') \supset A'$, pour tout A' permet ensuite de conclure. □

De ces deux résultats, on obtient l'algorithme de Roy et Warshall suivant pour le calcul de la fermeture transitive d'un graphe :

```
static void phi (GrapheMat g, int x) {
  int n = g.m.length;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      g.m[i][j] = g.m[i][j] || g.m[i][x] && g.m[x][j];
}

static public void fermetureTransitive (GrapheMat g) {
  for (int k = 0; k < g.m.length; ++k)
    phi(g, k);
 }
```

Dans la fonction *phi*, la matrice est modifiée en cours de calcul ; grâce à la proposition 1.4, cela ne change pas le résultat final puisque $\Phi_x = \Phi_x^2$. Maintenant, si on déplie l'appel de la fonction *phi* (*inlining* en anglais), on obtient :

```
static public void fermetureTransitive (GrapheMat g) {
  int n = g.m.length;
  for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
         g.m[i][j] = g.m[i][j] || g.m[i][k] && g.m[k][j];
}
```

Cette fonction modifie le graphe G. Une bien meilleure solution consiste à donner comme résultat un nouveau graphe G^* de la manière suivante :

```
static public GrapheMat fermetureTransitive (GrapheMat g) {
  GrapheMat r = copieGraphe (g);
  int n = g.m.length;
  for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
         r.m[i][j] = r.m[i][j] || r.m[i][k] && r.m[k][j];
  return r;
}

GrapheMat copieGraphe (GrapheMat g) {
  int n = g.m.length;
  GrapheMat r = new GrapheMat (n);
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      r.m[i][j] = g.m[i][j];
  return r;
}
```

L'algorithme ci-dessus effectue un nombre d'opérations que l'on peut majorer par n^3, chaque exécution de la fonction *phi* pouvant nécessiter n^2 opérations ; cet algorithme est donc meilleur que le calcul des puissances successives de la matrice d'adjacence.

Finalement, on remarquera que la fermeture transitive calculée dans cette section ne contient pas la relation identité. Elle ne contient pas les chemins de longueur nulle joignant tout sommet x à lui-même. Pour calculer cette fermeture transitive au sens large, on modifie très simplement l'algorithme de Warshall en ajoutant la matrice identité au résultat.

$succ[0] = \langle 1, 2 \rangle$
$succ[1] = \langle 3, 2, 5 \rangle$
$succ[2] = \langle 5 \rangle$
$succ[3] = \langle 4 \rangle$
$succ[4] = \langle 1 \rangle$
$succ[5] = \langle 3 \rangle$

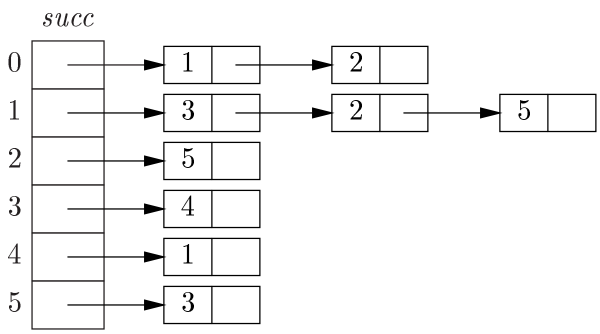


FIG. 1.5 – Représentation des graphes par listes de successeurs

1.4 Listes de successeurs

La matrice d'adjacence peut être très creuse, une façon plus compacte de représenter un graphe consiste à associer à chaque sommet x la liste de ses successeurs. On utilise un tableau de listes que l'on note *succ* ; on suppose que les sommets sont numérotés de 0 à $n-1$; la quantité $succ[x]$ désigne la liste des successeurs de x. Cette représentation est utile pour obtenir tous les successeurs d'un sommet x. Elle permet leur accès en un nombre d'opérations égal au nombre d'éléments de cet ensemble et non pas au nombre total de sommets comme c'est le cas dans la matrice d'adjacence, ce qui peut faire une différence sensible quand n est grand. Pour un graphe $G = (X, A)$, la place prise en mémoire est en $O(|X| + |A|)$ au lieu d'être en $O(|X|^2)$ pour la représentation matricielle. Dans le cas du graphe de la figure 1.2, la représentation se trouve à présent sur la figure 1.5.

Une nouvelle classe *Graphe* correspond à la représentation par listes de successeurs :

```
class Graphe {
  Liste[ ] succ;
  Graphe (int n) { succ = new int[n]; }
}

class Liste {
  int val;
  Liste suivant;
  Liste (int x, Liste ls) { val = x; suivant = ls; }
}
```

La déclaration de la classe *Liste* est la définition classique des listes d'entiers. Le parcours de la liste des successeurs y d'un sommet x s'effectue à l'aide de la suite des instructions suivantes. On retrouvera cette suite d'instructions comme brique de base de beaucoup de constructions d'algorithmes sur les graphes :

```
for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    Traitement de y
}
```

que l'on peut résumer en langue naturelle de la manière suivante :

Pour tout sommet y, successeur du sommet x dans G, faire {
    Traitement de y
}

On peut passer de la représentation d'un graphe par une matrice d'adjacence à la représentation par listes de successeurs en définissant dans la classe *Graphe* le constructeur suivant :

```
Graphe (GrapheMat g) {
  int n = g.m.length;
  succ = new Liste[n];
  for (int i = 0; i < n; ++i) {
    for (int j = n-1; j >= 0; --j)
      if (g.m[i][j])
        succ[i] = new Liste (j, succ[i]);
  }
}
```

L'ordre décroissant sur la deuxième boucle n'est qu'une légère subtilité pour présenter les listes de successeurs triées dans l'ordre croissant. Mais il n'y a aucune obligation à faire de la sorte. Réciproquement le constructeur suivant dans la classe *GrapheMat* permet de passer de la représentation par listes de successeurs à la représentation matricielle.

```
GrapheMat (Graphe g) {
  int n = g.succ.length;
  m = new boolean[n][n];
  for (int x = 0; x < n; ++x) {
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
      int y = ls.val;
      m[x][y] = true;
    }
  }
}
```

1.5 Graphes et arborescences

Définition 1.6 *Une arborescence (X, A, r) de racine r est un graphe (X, A) où r est un élément de X tel que pour tout sommet x il existe un unique chemin d'origine r et d'extrémité x. Soit,*

$$\forall x \quad \exists! \quad y_0, y_1, \ldots, y_p$$

tels que :

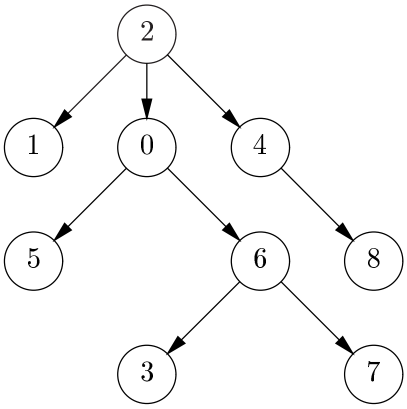
$$y_0 = r, \quad y_p = x, \quad \forall i, \ 0 \le i < p \quad (y_i, y_{i+1}) \in A$$

L'entier p est appelé la *profondeur* du sommet x dans l'arborescence. On montre facilement que, dans une arborescence, la racine r n'admet pas de prédécesseur et que tout sommet y, différent de r, admet un seul prédécesseur. Ceci implique :

$$|A| = |X| - 1$$

La différence entre une arborescence et un arbre est mineure. Dans un arbre, les fils d'un sommet sont ordonnés (on distingue le fils gauche du fils droit). Tel n'est pas le cas dans une arborescence. Les arborescences servent depuis longtemps pour représenter des arbres généalogiques. Le vocabulaire utilisé pour les arborescences emprunte beaucoup de termes relevant des relations familiales.

L'unique prédécesseur d'un sommet (différent de r) est appelé son *père*, l'ensemble $y_0, y_1, \ldots y_p$ $(p \ge 0)$, formant le chemin de $r = y_0$ à $x = y_p$, est appelé l'ensemble des *ancêtres* de x, les successeurs de x sont aussi appelés ses *fils*. L'ensemble des sommets



$$\begin{array}{rccccccccc} i: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ pere(i): & 2 & 2 & 2 & 6 & 2 & 0 & 0 & 6 & 4 \end{array}$$

FIG. 1.6 – Une arborescence et son vecteur *pere*

extrémités d'un chemin d'origine x est l'ensemble des *descendants* de x ; il constitue une arborescence de racine x, celle-ci est l'union de $\{x\}$ et des arborescences formées des descendants des fils de x. Pour des raisons de commodité d'écriture qui apparaîtront dans la suite, nous adoptons la convention que tout sommet x est à la fois ancêtre et descendant de lui-même. Nous représenterons une arborescence par le vecteur *pere*, qui à chaque sommet différent de la racine associe son père. Par convention, la racine sera le père d'elle-même.

La transformation d'un arborescence vue comme un graphe avec ses listes de successeurs en une arborescence représenté par le vecteur *pere* s'exprime simplement par le constructeur suivant :

```
class Arborescence {
  int[ ] pere;

  Arborescence (int n) { pere = new int[n]; }

  Arborescence (Graphe g, int r) {
    int n = g.succ.length;
    pere = new int[n];
    pere[r] = r;
    for (int x = 0; x < n; ++x)
      for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
        int y = ls.val;
        pere[y] = x;
      }
  }
}
```

Dans la suite, on suppose que l'ensemble des sommets X est l'ensemble des entiers compris entre 0 et $n-1$, une arborescence est dite *préfixe* si, pour tout sommet i, l'ensemble des descendants de i est un intervalle de l'ensemble des entiers dont le plus petit élément est i.

Dans une arborescence préfixe, les intervalles de descendants s'emboîtent les uns dans les autres comme des systèmes de parenthèses ; ainsi, si y n'est pas un descendant

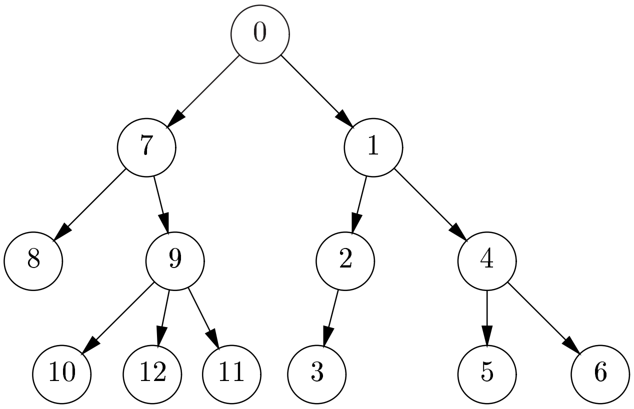


FIG. 1.7 – Une arborescence préfixe

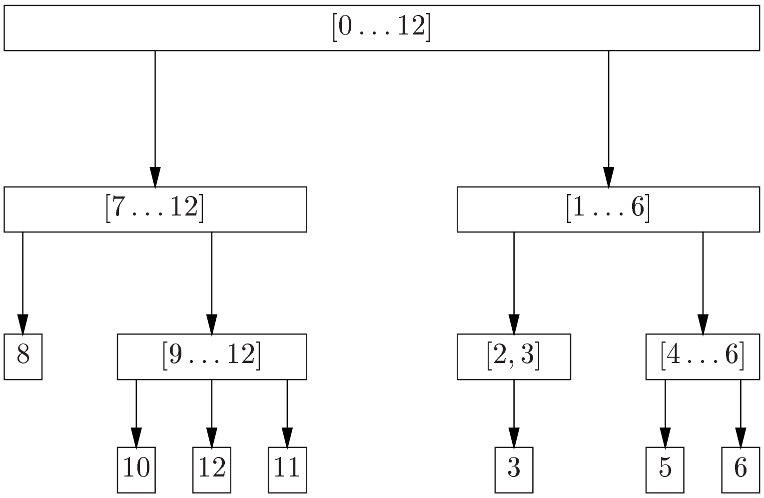


FIG. 1.8 – Emboitement des descendants dans une arborescence préfixe

de x, ni x un descendant de y, les descendants de x et de y forment des intervalles disjoints. En revanche, si x est un ancêtre de y, l'intervalle des descendants de y est inclus dans celui des descendants de x.

Proposition 1.7 *Pour toute arborescence* (X, A, r), *il existe une re-numérotation des éléments de* X *qui la rend préfixe.*

Démonstration Pour trouver cette numérotation, on applique l'algorithme récursif suivant :

- La racine est numérotée 0.
- Un de ses fils x_1 de la racine est numéroté 1.
- L'arborescence des descendants de x_1 est numérotée par appels récursifs de l'algorithme, on obtient ainsi des sommets numérotés de 1 à p_1.
- Un autre fils de la racine est numéroté $p_1 + 1$; les descendants de ce fils sont numérotés récursivement de $p_1 + 1$ à p_2.
- On procède de même et successivement pour tous les autres fils de la racine.

La démonstration que la numérotation obtenue est bien préfixe se fait par récurrence sur le nombre de sommets de l'arborescence. □

La re-numérotation préfixe d'une arborescence ne fait rien de plus que le parcours d'un arbre en ordre préfixe. On en déduit la construction d'une arborescence préfixe à partir de la représentation d'une arborescence vue comme un graphe avec ses listes de successeurs, comme suit :

```
int i = -1;

static int[ ] numPrefixe (Graphe g, int r) {
  int n = g.succ.length;
  int[ ] num = new int[n];
  numPrefixe1 (g, r, num);
  return num;
}

static void numPrefixe1 (Graphe g, int x, int[ ] num, int i) {
  numero[x] = ++i;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    numPrefixe1 (g, y, num);
  }
}

static void appliquerNum (int[ ] pere, int[ ] num) {
  int n = pere.length;
  for (int i = 0; i < n; ++i)
    pere[num[i]] = num[pere[i]];
}

static Arborescence arboPrefixe (Graphe g, int r) {
  Arborescence a = new Arborescence (g, r);
  appliquerNum (a.pere, numPrefixe(g, r));
  return a;
}
```

Exercice 1 Ecrire la fonction *numPrefixe* sans utiliser la variable globale i, c'est-à-dire en transformant *numOrdre* en variable locale.

Exercice 2 Construire directement l'arborescence préfixe sans utiliser le constructeur *Arborescence*.

1.6 Arborescences de Trémaux

Si les arbres peuvent être considérés comme un cas particulier des graphes, on associe souvent des arborescences ou *arbres de recouvrement* (*spanning trees* en anglais) à des parcours de graphes quelconques. En effet, pour parcourir tous les sommets d'un graphe, à la différence des arbres il faut éviter de boucler, notamment dans le cas des graphes cycliques.

En fait, le parcours de graphe le plus populaire a été mis au point par un ingénieur du 19ème siècle, Trémaux, dont les travaux sont cités dans un des premiers livres sur les graphes dû à Sainte Lagüe. Son but était de résoudre le problème de la sortie d'un labyrinthe. Depuis l'avènement de l'informatique, nombreux sont ceux qui ont redécouvert l'algorithme de Trémaux. Certains en ont donné une version bien plus précise et ont montré qu'il pouvait servir à résoudre de façon très astucieuse beaucoup de problèmes algorithmiques sur les graphes. Il est maintenant connu sous l'appellation de *depth-first search* (parcours en profondeur), nom que lui a donné un de ses brillants promoteurs : R. E. Tarjan. Ce dernier a découvert, entre autres, le très efficace algorithme de recherche des composantes fortement connexes que nous décrirons à la fin de ce chapitre. L'algorithme consiste à démarrer d'un sommet et à avancer dans le graphe en ne repassant pas deux fois par le même sommet. Lorsque l'on est bloqué, on revient sur ses pas jusqu'à pouvoir repartir vers un sommet non visité. Cette opération de retour sur ses pas est très élégamment prise en charge par l'écriture d'une fonction récursive. Trémaux qui n'avait pas cette possibilité à l'époque utilisait un fil d'Ariane lui permettant de se souvenir par où il était arrivé à cet endroit dans le labyrinthe. Le programme récursif suivant construit une arborescence (dite de Trémaux) pour un graphe g quelconque à partir d'un de ses sommets x :

```
static Arborescence arboTremaux (Graphe g, int x) {
  int n = g.succ.length;
  Arborescence a = new Arborescence (n);
  for (int i = 0; i < n; ++i) a.pere[i] = -1;
  a.pere[x] = x;
  tremaux(g, x, a);
  return a;
}

static void tremaux (Graphe g, int x, Arborescence a) {
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (a.pere[y] == -1) {
      a.pere[y] = x;
      tremaux(g, y, a);
    }
  }
}
```

Le calcul effectif de l'arborescence de Trémaux de racine x s'effectue en initialisant à la valeur -1 pour signaler que le sommet correspondant n'a pas encore été visité. La figure 1.9 explique l'exécution de l'algorithme sur un exemple, les appels de la fonction sont dans l'ordre :

```
tremaux (g, 0, a)
    tremaux (g, 1, a)
    tremaux (g, 2, a)
        tremaux (g, 5, a)
```

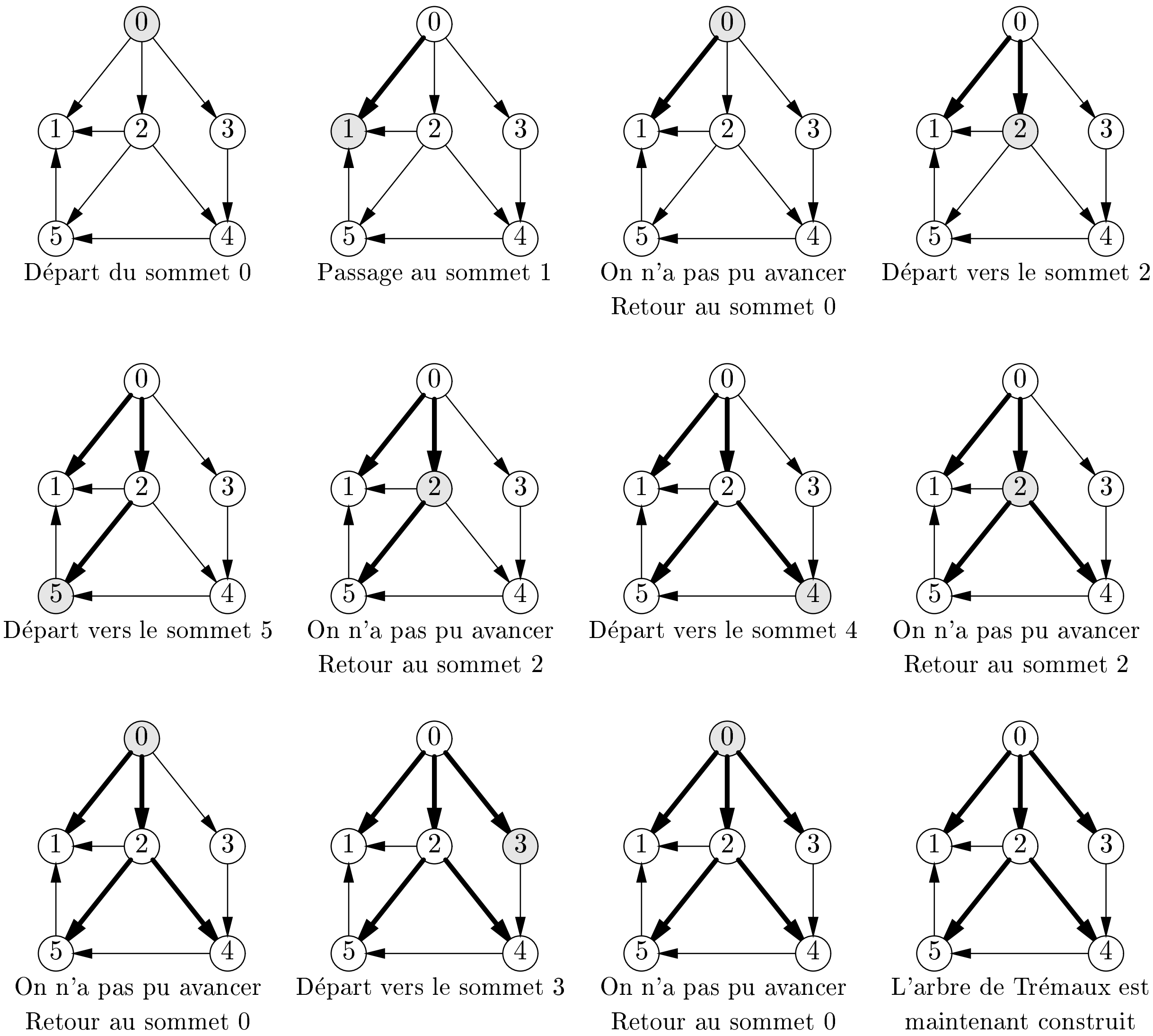


Fig. 1.9 – Exécution de l'algorithme de Trémaux

```
        tremaux (g, 4, a)
    tremaux (g, 3, a)
```

Une version non récursive de cet algorithme est obtenue en suivant le principe général qui associe à tout programme récursif un programme itératif manipulant une pile avec les primitives associées : *Pile.ajouter*, *Pile.supprimer*, *Pile.estVide* pour ajouter un élément au sommet de la pile, pour retirer le sommet de la pile tout en le renvoyant comme résultat, pour tester si la pile est vide.

```
static Arborescence arboTremauxPile (Graphe g, int x) {
  int n = g.succ.length;
  Arborescence a = new Arborescence (n);
  for (int i = 0; i < n; ++i) a.pere[i] = -1;
  Pile p = new Pile(n);
  a.pere[x] = x;
  Pile.ajouter (p, x);
  while ( !Pile.estVide(p) ) {
    x = Pile.supprimer (p);
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
      int y = ls.val;
      if (a.pere[y] == -1) {
        a.pere[y] = x;
        Pile.ajouter (p, y);
      }
    }
  }
  return a;
}
```

Le parcours n'est pas tout à fait le même que dans le programme récursif, puisque l'ordre de parcours des successeurs d'un même sommet se fait dans l'ordre de la liste des successeurs dans la version récursive, alors qu'il se fait dans l'ordre inverse dans la version itérative. Mais les deux parcours correspondent bien à des arborescences de Trémaux, puisqu'ils s'effectuent bien en privilégiant la profondeur.

L'ensemble des sommets atteignables à partir du sommet x est formé des sommets tels que *pere[y]* est différent de -1 à la fin de l'algorithme. On a donc un algorithme qui répond à la question *existeChemin(g, x, y)* examinée plus haut avec un nombre d'opérations qui est de l'ordre du nombre d'arcs du graphe (lequel est inférieur à n^2), ce qui est bien meilleur qu'avec l'algorithme utilisant des matrices d'adjacence.

L'ensemble des arcs du graphe $G = (X, A)$ qui ne sont pas dans l'arborescence de Trémaux (Y, T, x) est divisé en quatre sous-ensembles :

1. les arcs dont l'origine n'est pas dans Y, ce sont les arcs issus d'un sommet qui n'est pas atteignable à partir de x ;
2. les arcs de *descente*, il s'agit des arcs de la forme (y, z) où z est un descendant de y dans (Y, T, x), mais n'est pas un de ses successeurs dans cette arborescence ;
3. les arcs de *retour*, il s'agit des arcs de la forme (y, z) où z est un ancêtre de y dans (Y, T, x) ;
4. les arcs *transverses*, il s'agit des arcs de la forme (y, z) où z n'est pas un ancêtre, ni un descendant de y dans (Y, T, x).

On remarquera que, si (y, z) est un arc transverse, on aura rencontré z avant y dans l'algorithme de Trémaux.

Sur le graphe de la figure 1.10, les différentes sortes d'arcs y sont représentés par des lignes particulières. Les arcs de l'arborescence sont en traits gras ; les arcs dont

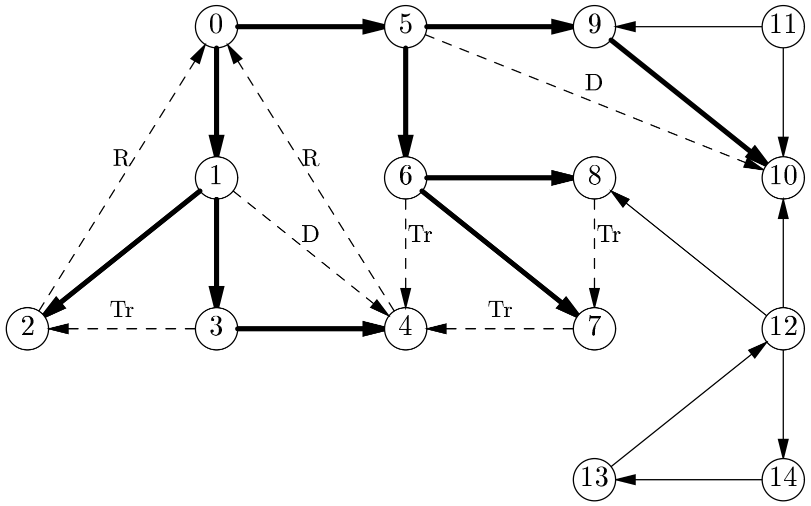


FIG. 1.10 – Les arcs obtenus par Trémaux

l'origine n'est pas dans Y sont dessinés en trait simple ; les arcs de descente, de retour ou transverses sont en tiretés et munis d'une étiquette permettant de les reconnaître, D, R ou Tr. Les sommets ont été numérotés suivant l'ordre dans lequel on les rencontre par l'algorithme de Trémaux. Ainsi, les arcs de l'arborescence et les arcs de descente vont d'un sommet à un sommet d'étiquette plus élevée et c'est l'inverse pour les arcs de retour ou transverses.

1.7 Arborescences des plus courts chemins.

Une autre arborescence souvent associée à un graphe quelconque est l'arborescence des plus courts chemins. Elle correspond à un parcours en largeur (*breadth-first search* en anglais). On parcourt le graphe à partir d'un sommet en émettant une onde à partir de ce sommet et en rencontrant en priorité les sommets à égales distances de ce sommet.

Définition 1.8 *Dans un graphe $G = (X, A)$, pour chaque sommet x, une arborescence des plus courts chemins de racine x est une arborescence (Y, B, x) telle que :*

- *Un sommet y appartient à Y si et seulement s'il existe un chemin d'origine x et d'extrémité y.*
- *La longueur du plus court chemin de x à y dans G est égale à la profondeur de y dans l'arborescence (Y, B, x).*

Cette arborescence existe bien puisque, si $a_1, a_2, \ldots, a_p$ est un plus court chemin entre $org(a_1)$ et $ext(a_p)$, alors le chemin $a_1, a_2, \ldots, a_i$ est aussi un plus court chemin entre $org(a_1)$ et $ext(a_i)$ pour tout i vérifiant $1 \leq i \leq p$.

Théorème 1.9 *Pour tout graphe $G = (X, A)$ et tout sommet x de G, il existe une arborescence des plus courts chemins de racine x.*

Démonstration On considère la suite d'ensembles de sommets construite de la façon suivante :

- $Y_0 = \{x\}$.

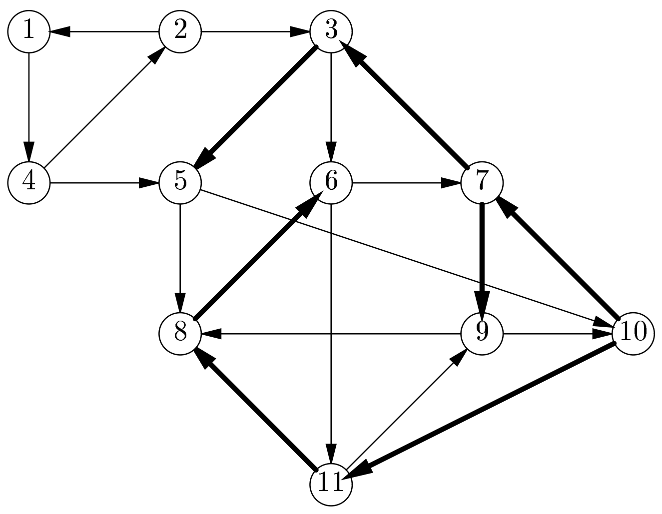


Fig. 1.11 – Une arborescence des plus courts chemins de racine 10

– Y_1 est l'ensemble des successeurs de x, duquel il faut éliminer x si le graphe possède un arc ayant x pour origine et pour extrémité.
– Y_{i+1} est l'ensemble des successeurs d'éléments de Y_i qui n'appartiennent pas à $\bigcup_{k=1,i} Y_i$.

D'autre part pour chaque Y_i ($i > 0$), on construit l'ensemble d'arcs B_i contenant pour chaque $y \in Y_i$ un arc ayant comme extrémité y et dont l'origine est dans Y_{i-1}. On pose ensuite : $Y = \bigcup Y_i, B = \bigcup B_i$. Le graphe (Y, B) est par construction une arborescence. C'est une arborescence des plus courts chemins grâce à la remarque ci-dessus. □

La figure 1.11 donne un exemple de graphe et une arborescence des plus courts chemins de racine 10, celle-ci est représentée en traits gras, les ensembles Y_i et B_i sont les suivants :

$Y_0 = \{10\}$
$Y_1 = \{7, 11\}$ $B_1 = \{(10, 7), (10, 11)\}$
$Y_2 = \{3, 9, 8\}$ $B_2 = \{(7, 3), (7, 9), (11, 8)\}$
$Y_3 = \{5, 6\}$ $B_3 = \{(3, 5), (8, 6)\}$

A nouveau, la preuve de ce théorème se transforme très simplement en un algorithme de construction de l'arborescence (Y, B). Cet algorithme est souvent appelé algorithme de *parcours en largeur*. Nous le décrivons ci dessous, il utilise une file d'attente avec les primitives associées : *FIFO.ajouter*, *FIFO.supprimer*, *FIFO.estVide* pour ajouter un élément en bout de file, pour retirer le premier élément dans la file tout en le renvoyant comme résultat, pour tester si la file est vide. La file gère les ensembles Y_i. On ajoute les éléments des Y_i successivement dans la file, qui contient donc les Y_i les uns à la suite des autres. La vérification de ce qu'un sommet n'appartient pas à $\bigcup_{k=1,i} Y_i$ se fait en testant si *pere[y]* ne vaut pas -1.

```
static Arborescence arboPlusCourt (Graphe g, int x) {
  int n = g.succ.length;
  Arborescence a = new Arborescence (n);
  for (int i = 0; i < n; ++i) a.pere[i] = -1;
  FIFO f = new FIFO(n);
```

```
    a.pere[x] = x;
    FIFO.ajouter (f, x);
    while ( !FIFO.estVide(f) ) {
      x = FIFO.supprimer (f);
      for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
        int y = ls.val;
        if (a.pere[y] == -1) {
          a.pere[y] = x;
          FIFO.ajouter (f, y);
        }
      }
    }
    return a;
  }
```

On remarque que ce programme est le même que le programme itératif pour calculer une arborescence de Trémaux, en remplaçant une pile par une file d'attente.

1.8 Parcours de graphes

Nous avons considéré les parcours de graphes en profondeur et en largeur, pour traverser un graphe sans boucler, ni passer plus d'une fois par chaque sommet. A nouveau, comme la structure d'arbre (ou d'arborescence) est un cas particulier de la structure de graphe, on peut se demander quels sont, sur les graphes, les équivalents des parcours préfixes ou postfixes sur les arbres.

Au lieu de renvoyer une arborescence, nous considérons une nouvelle version du programme *tremaux* qui retourne une numérotation des sommets qui donne le numéro d'ordre de chaque sommet dans l'ordre où on l'a rencontré. Pour cela, on colorie les sommets traditionnellement avec trois couleurs : blanc pour les sommets non encore explorés, gris pour un sommet partiellement traité, noir pour un sommet complètement traité.

```
final static int BLANC = 0, GRIS = 1, NOIR = 2;
int numOrdre = -1;

static int[ ] tremauxPrefixe (Graphe g) {
  int n = g.succ.length;
  int[ ] couleur = new int[n], num = new int[n];
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  for (int x = 0; x < n; ++x)
    if (couleur[x] == BLANC)
      tremauxPref (g, x, couleur, num);
  return num;
}

static void tremauxPref (Graphe g, int x, int[ ] couleur, int[ ] num) {
  couleur[x] = GRIS;
  num[x] = ++numOrdre;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (couleur[y] == BLANC)
      tremauxPref (g, y, couleur, num);
  }
  couleur[x] = NOIR;
}
```

Remarquons que, si la coloration en gris se faisait avant l'appel à *tremauxPref*, cela ne ferait aucune différence, puisque le parcours en profondeur traite immédiatement tout nouveau sommet rencontré.

On obtient une numérotation postfixe pour un parcours en profondeur en changeant simplement l'emplacement où se fait l'affectation du numéro de chaque sommet.

```
static int[ ] tremauxPostfixe (Graphe g) {
  int n = g.succ.length;
  int[ ] couleur = new int[n], num = new int[n];
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  for (int x = 0; x < n; ++x)
    if (couleur[x] == BLANC)
      tremauxPost (g, x, couleur, num);
  return num;
}

static void tremauxPost (Graphe g, int x, int[ ] couleur, int[ ] num) {
  couleur[x] = GRIS;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (couleur[y] == BLANC)
      tremauxPost (g, y, couleur, num);
  }
  num[x] = ++numOrdre;
  couleur[x] = NOIR;
}
```

Un certain nombre de propriétés sur la couleur des sommets pendant le parcours récursif peuvent être énoncées. Par exemple, dans une arborescence de Trémaux, un sommet noir ne peut jamais être l'origine d'un arc vers un sommet blanc. De la même manière, un sommet blanc extrémité d'un chemin complètement blanc issu d'un sommet gris se retrouvera descendant de ce sommet gris dans l'arbre de recouvrement ; etc.

Exercice 3 Ecrire les fonctions *tremauxPrefixe* et *tremauxPostfixe* en transformant la variable globale *numOrdre* en variable locale.

Le parcours en largeur est aussi plus compréhensible en ne raisonnant que sur la couleur des sommets :

```
static int[ ] largeurPref (Graphe g, int x) {
  int n = g.succ.length, numOrdre = -1;
  int[ ] couleur = new int[n], num = new int[n];
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  FIFO f = new FIFO(n);
  couleur[x] = GRIS;
  FIFO.ajouter (f, x);
  while ( !FIFO.estVide(f) ) {
    x = FIFO.supprimer (f);
    num[x] = ++numOrdre;
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
      int y = ls.val;
      if (couleur[y] == BLANC) {
        couleur[y] = GRIS;
        FIFO.ajouter (f, y);
      }
    }
    couleur[x] = NOIR;
  }
  return num;
}
```

Dans ce parcours en largeur, la coloration en gris d'un sommet doit se faire avant

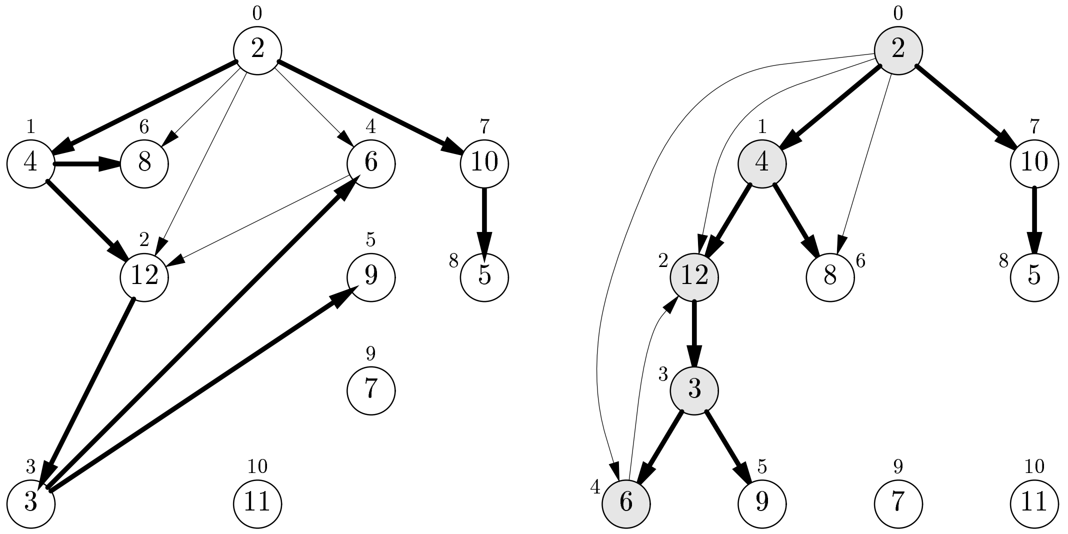


FIG. 1.12 – Un graphe avec cycle et son arbre de recouvrement

de le placer dans la file d'attente, sous peine de l'y mettre plusieurs fois, et de devoir faire intervenir la valeur du résultat *num* qu'il est plus sage de distinguer du contrôle du parcours.

Quant au parcours postfixe, il s'obtient en déplaçant l'instruction de numérotation d'un sommet, et en la plaçant avant la coloration en noir. Mais le résultat *num* serait alors le même que celui obtenu dans le parcours préfixe, puisque, dans le parcours en largeur, rien ne se passe entre le rangement dans la file des successeurs d'un sommet et le traitement du sommet en question.

Tous les parcours en profondeur ou en largeur ne passent qu'une seule fois par chaque sommet et font un nombre d'opérations proportionnel au nombre d'arcs. La complexité de tels parcours dans un graphe $G = (X, A)$ est donc en $O(|X|+|A|)$, qu'on écrit souvent simplement par $O(X + A)$.

1.9 Graphes acycliques

Les graphes sans cycle (*directed acyclic graphs* en anglais ou plus simplement *dag*s) interviennent souvent dans les problèmes d'ordonnancement : organisation des différents travaux sur un chantier, dépendance entre modules dans un projet de programmation, etc. Un arbre est un *dag* particulier ; mais, dans un *dag*, contrairement aux arbres, on peut partager des sous-*dag*s. Un graphe est dit *acyclique* s'il ne contient pas de circuit.

Un algorithme pour tester l'acyclicité d'un graphe se fait facilement par un parcours en profondeur. En effet, si un graphe contient un circuit, cela signifie que, dans un arbre de recouvrement, un sommet x est l'origine d'un arc dont l'extrémité est un ancêtre de x dans l'arbre de recouvrement, comme indiqué sur la figure 1.12 en considérant $x = 6$ duquel part un arc vers le sommet gris 12. Cela veut dire que dans le parcours en profondeur le successeur d'un sommet gris pourra être gris. Si cela ne se produit jamais, le graphe est acyclique. D'où le programme suivant :

```
static boolean acyclique (Graphe g) {
```

```
    int n = g.succ.length;
    int[ ] couleur = new int[n];
    for (int x = 0; x < n; ++x) couleur[x] = BLANC;
    for (int x = 0; x < n; ++x)
      if ( (couleur[x] == BLANC) && cycleEn (g, x, couleur) )
          return false;
    return true;
  }

  static boolean cycleEn (Graphe g, int x, int[ ] couleur) {
    couleur[x] = GRIS;
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
      int y = ls.val;
      if ( (couleur[y] == GRIS) ||
           (couleur[y] == BLANC) && cycleEn (g, y, couleur) )
          return true;
    }
    couleur[x] = NOIR;
    return false;
  }
```

La complexité de cet algorithme est donc en $O(X + A)$. On se convainc qu'on ne peut faire mieux, puisqu'il faudra bien considérer tous les arcs du graphe avant de décider de son acyclicité.

Un problème classique sur les graphes sans cycle est le tri topologique. Il consiste à organiser l'agenda d'un certain nombre de tâches données par un graphe de dépendances. Prenons le cas de la lecture d'un livre, par exemple le livre de Barendregt sur le lambda-calcul [27]. Dans ses premières pages, on voit le diagramme (assez compliqué) de la figure 1.13 décrivant l'ordre de lecture des différents chapitres. Ainsi pour lire le chapitre 16, il faut avoir lu les chapitres 4, 8 et 15. Un lecteur courageux veut lire le strict minimum pour appréhender le chapitre 21. Il faut donc qu'il transforme l'ordre partiel indiqué par les dépendances du diagramme en un ordre total déterminant la liste des chapitres nécessaires au chapitre 21. Bien sûr, ceci n'est pas possible si le graphe de dépendance contient un cycle. L'opération qui consiste à mettre ainsi en ordre les sommets d'un graphe dirigé sans cycle est appelée le tri topologique.

Le tri topologique ordonne les sommets d'un *dag* en une suite dans laquelle l'origine de chaque arc apparaît avant son extrémité. Pour un sommet s donné, on construit une liste formée de tous les sommets origines d'un chemin d'extrémité s. Cette liste doit en plus satisfaire la condition énoncée plus haut. On applique l'algorithme de descente en profondeur d'abord (Trémaux) sur le graphe inverse. (Au lieu de considérer les successeurs *succ[x]* du sommet x, on considère ses prédécesseurs *pred[x]*.) Au cours de cette recherche, quand on a fini de visiter un sommet, on le met en tête de liste. En fin de l'algorithme, on calcule l'image mirroir de la liste. Pour tester l'existence de cycles, on doit vérifier lorsqu'on rencontre un sommet déjà visité que celui-ci figure dans la liste résultat.

```
  final static BLANC = 0, GRIS = 1, NOIR = 2;

  static Liste triTopologique (Graphe g, int u) {
    int n = g.succ.length;
    int[ ] etat = new int[n];
    for (int x = 0; x < n; ++x) etat[x] = BLANC;
    return Liste.mirroir (DFS (g, u, etat, null));
  }
```

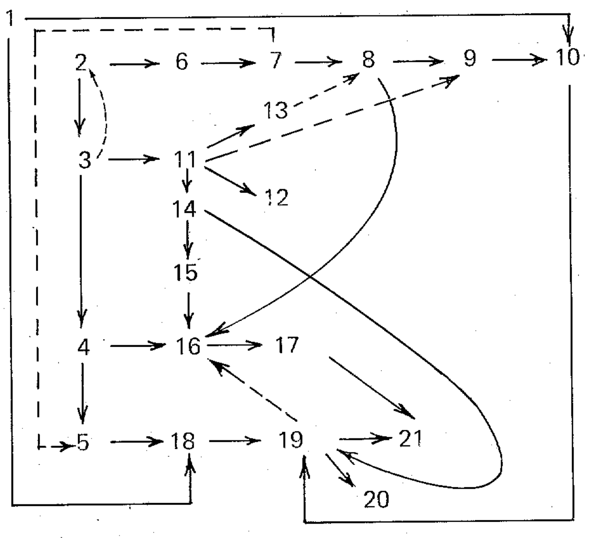


FIG. 1.13 – Un exemple de graphe acyclique

```
static Liste DFS (Graphe g, int x, int[ ] etat, Liste a_faire) {
  etat[x] = GRIS;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (etat[y] == GRIS) throw new Error ("Le graphe a un cycle");
    if (etat[y] == BLANC) {
      etat[y] = GRIS;
      a_faire = DFS (g, y, etat, a_faire);
  } }
  etat[x] = NOIR;
  return Liste.ajouter (x, a_faire);
}
```

La complexité de cet algorithme est celle d'une recherche en profondeur d'abord, don en $O(X+A)$, suivie de l'image mirroir qui ne dépasse pas $O(A)$. Au total la complexité est en $O(X+A)$.

1.10 Connexité dans un graphe non-orienté

Comme déjà mentionné, nous représentons les graphes non-orientés par des graphes orientés symétriques. Pour tous sommets x et y, s'il existe un arc (x, y), il existe aussi un arc (y, x). Donc, s'il existe un chemin de x à y, il existe aussi un chemin de y à x.

Définition 1.10 *Soit $G = (X, A)$ un graphe non-orienté. La composante connexe du sommet x est l'ensemble des sommets y reliés par un chemin à x. Un graphe est connexe s'il ne contient qu'une seule composante connexe.*

L'appartenance à une même composante connexe est une relation d'équivalence. Les composantes connexes forment une partition du graphe en sous-graphes déconnectés.

Dans un graphe non-orienté, un arbre de recouvrement ne contient jamais d'arc transverse. Plus exactement, l'ensemble des arcs du graphe qui ne sont pas dans un arbre de recouvrement de racine x est divisé en trois sous-ensembles :

1. Les arcs *déconnectés*, dont ni l'origine, ni l'extrémité ne sont dans l'arbre de recouvrement. Il n'existe pas de chemin de x vers l'origine ou l'extrémité de ces arcs.
2. Les arcs de *descente*, il s'agit des arcs de la forme (y, z) où z est un descendant non direct de y dans l'arbre de recouvrement.
3. Les arcs de *retour*, il s'agit des arcs de la forme (y, z) où z est un ancêtre de y dans l'arbre de recouvrement.

On en déduit que l'ensemble des sommets d'un arbre de recouvrement constitue exactement la composante connexe de sa racine. D'où le programme suivant pour imprimer les composantes connexes d'un graphe non-orienté G.

```
static void imprimerCompConnexes (Graphe g) {
  int n = g.succ.length;
  int[ ] couleur = new int[n];
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  for (int x = 0; x < n; ++x) {
    if (couleur[y] == BLANC) {
      imprimerComp (g, x, couleur);
      System.out.println();
    }
  }
}

static void imprimerComp (Graphe g, int x, int[ ] couleur) {
  couleur[x] = GRIS;
  System.out.print (x + " ");
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (couleur[y] == BLANC)
      imprimerComp (g, y, couleur);
  }
}
```

Il s'agit encore d'un parcours en profondeur. Le programme imprime les composantes connexes, ligne par ligne, en temps $O(X + A)$. De manière identique, on peut écrire un programme de sortie d'un labyrinthe, puisqu'il suffit de tester si la sortie s est dans la composante connexe de l'entrée x. Nous voulons alors retourner un chemin de x vers s, chemin que nous représentons par la liste des sommets par lesquels il passe.

```
static Liste sortieDeLabyrinthe (Graphe g, int x, int s) {
  int n = g.succ.length;
  int[ ] couleur = new int[n];
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  return chemin (g, x, s, couleur);
}

static void chemin (Graphe g, int x, int s, int[ ] couleur) {
  couleur[x] = GRIS;
  if (x == s)
    return new Liste (s, null);
  else {
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
      int y = ls.val;
      if (couleur[y] == BLANC)
        Liste ch = chemin (g, y, s, couleur);
        if (ch != null)
          return new Liste (x, ch);
    }
```

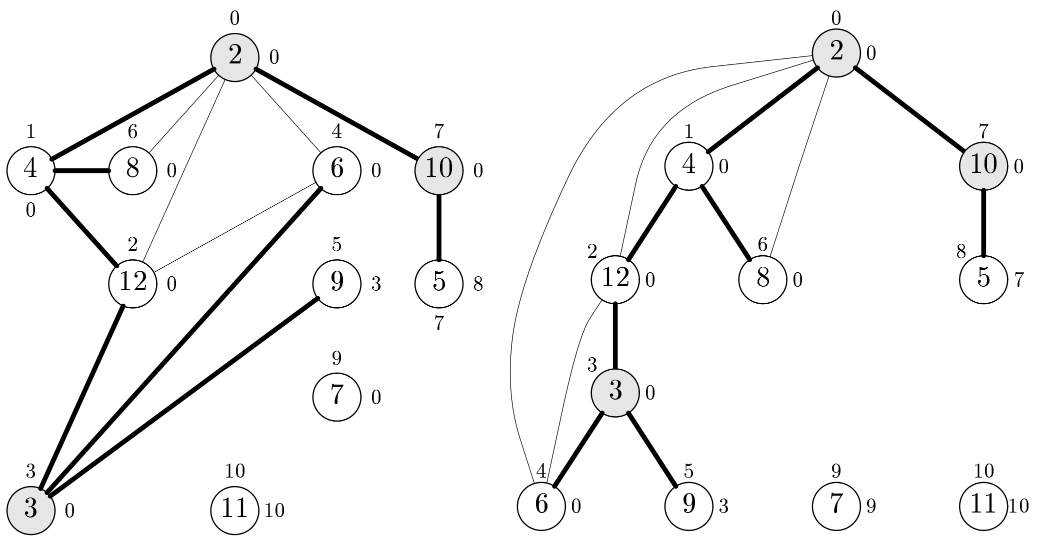


FIG. 1.14 – Points d'articulation dans un graphe non-orienté

```
    }
    return null;
  }
```

Exercice 4 Ecrire un programme qui imprime tous les chemins simples (c'est à dire ne passant pas deux fois par un même sommet) de l'entrée vers la sortie d'un labyrinthe.

Exercice 5 Ecrire un programme qui calcule un chemin le plus court reliant l'entrée à la sortie du labyrinthe.

1.11 Biconnexité dans un graphe non-orienté

Dans une composante connexe, il existe toujours un chemin pour relier deux sommets. Dans une composante biconnexe, il doit en exister toujours deux. Ce problème se pose par exemple quand on conçoit un plan de circulation ; on veut éviter les endroits névralgiques, obligatoires pour se rendre d'un endroit à un autre. De la même manière, dans un réseau informatique, on évite les sites qui déconnectent le réseau quand ils tombent en panne.

Définition 1.11 *Soit $G = (X, A)$ un graphe non-orienté. Un sommet x est un* point d'articulation *si la suppression de x déconnecte deux sommets de G.*

Dans le graphe de la figure 1.14 (à gauche), les points d'articulation sont indiqués en grisé. La suppression du sommet 3 déconnecte le sommet 9 ; de même pour le sommet 2 qui déconnecte l'ensemble $\{10, 5\}$; et pour 10 qui déconnecte 5. Tarjan a trouvé une solution remarquable pour calculer les points d'articulation dans un graphe. Sa solution ne fait intervenir qu'un parcours en profondeur et la numérotation préfixe correspondante. En effet, caractérisons les points d'articulations sur un arbre de recouvrement. Dans l'arbre de recouvrement, un point d'articulation est un sommet x possédant un fils dont tous les descendants ne sont pas des origines d'arcs de retour vers des ancêtres stricts de x dans l'arbre. Par exemple, sur l'arbre de recouvrement de la figure 1.14 (à droite), le sommet 3 est un point d'articulation, puisque son fils 9 ne possède qu'un arc de retour vers 3 dans l'arbre de recouvrement. Mais, le sommet 12 n'est pas un point d'articulation, puisque son unique fils 3 a comme descendant 6 dont 2 est un successeur. Or 2 est un ancêtre strict de 12 dans l'arbre de recouvrement. Intuitivement, on voit bien que, si on enlève le sommet 3, on déconnecte le sous-arbre 9, alors qu'on ne déconnecte rien si on enlève le sommet 12.

Avec ce seul critère, la racine d'un arbre de recouvrement serait toujours un point d'articulation. Il faut donc faire un cas particulier pour décider si la racine est un point d'articulation. Le critère est alors simple : la racine est un point d'articulation si elle contient plusieurs fils dans l'arbre de recouvrement. Sur la figure 1.14, c'est bien le cas puisque 4 et 10 sont deux fils de 2.

Définition 1.12 *Soit $G = (X, A)$ un graphe, x un sommet et (Y, T, x) une arborescence de Trémaux induisant une numérotation préfixe num des sommets. Le* point d'attache *$at(y)$ d'un sommet y de Y est le sommet de plus petit numéro, extrémité d'un chemin de G d'origine y et contenant au plus un arc (u, v) tel que $num[u] > num[v]$.*

Cela signifie qu'un point d'attache est un sommet atteignable par un chemin ne comportant au plus qu'un arc transverse ou de retour, puisque dans le cas des graphes non-orientés, les arcs transverses n'existent pas. Sur la figure 1.14, pour chaque sommet y, son numéro $num[y]$ dans l'ordre préfixe est placé au-dessus de y, le numéro de son point d'attache $num[at(y)]$ est placé à droite. Les points d'attache se calculent (dans un graphe orienté ou pas) grâce à la remarque suivante :

Proposition 1.13 *Le point d'attache $at(y)$ du sommet y est le sommet de plus petit numéro parmi les sommets suivants :*

- *le sommet y.*
- *les points d'attaches des fils de y dans (Y, T, x).*
- *les extrémités des arcs transverses ou de retour dont l'origine est y.*

Pour détecter les points d'articulation, il reste donc à réaliser les deux types de tests pour un sommet ordinaire de l'arbre et pour la racine. Un peu d'arithmétique sur la numérotation préfixe effectue le premier test. Un sommet x, différent de la racine, sera un point d'articulation si et seulement s'il possède un fils y tel que :

$$num[x] \leq num[at(y)]$$

On en déduit l'algorithme suivant pour détecter les points d'articulation. Le résultat est rangé dans un tableau de booléens *articulation* indiquant, pour chaque sommet x, s'il est un point d'articulation.

```
static int numOrdre; static int[ ] num;

static boolean[ ] trouverArticulations (Graphe g) {
  int n = g.succ.length;
  id = -1; num = new int[n];
  boolean[ ] articulation = new boolean[n];
  for (int x = 0; x < n; ++x) num[x] = -1;
  for (int x = 0; x < n; ++x)
    if (num[x] == -1) {
      int r = attache1(g, x, articulation);
    }
  return articulation;
```

```
}

static int attache (Graphe g, int x, boolean[ ] articulation) {
  int min = num[x] = ++numOrdre;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val; int m;
    if (num[y] == -1) {
      m = attache (g, y, articulation);
      if (m >= num[x])
        articulation[x] = true;
    } else
      m = num[y];
    min = Math.min (min, m);
  }
  return min;
}
```

La fonction *attache1* est pratiquement identique à *attache* sauf que, pour la racine, il faut compter le nombre de ses fils dans l'arbre de recouvrement.

```
static int attache1 (Graphe g, int x, boolean[ ] articulation) {
  int min = num[x] = ++numOrdre;
  int nfils = 0;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val; int m;
    if (num[y] == -1) {
      ++nfils;
      m = attache (g, y, articulation);
      if (m >= num[x])
        articulation[x] = true;
    } else m = num[y];
    min = Math.min (min, m);
  }
  articulation[x] = nfils > 1;
  return min;
}
```

On peut déplier l'appel de *attache1* dans *trouverArticulations* :

```
static boolean[ ] trouverArticulations (Graphe g) {
  int n = g.succ.length;
  id = -1; num = new int[n];
  boolean[ ] articulation = new boolean[n];
  for (int x = 0; x < n; ++x) num[x] = -1;
  for (int x = 0; x < n; ++x)
    if (num[x] == -1) {
      num[x] = ++numOrdre;
      int nfils = 0;
      for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
        int y = ls.val;
        if (num[y] == -1) {
          ++nfils;
          int m = attache (g, y, articulation);
        }
      }
      articulation[x] = nfils > 1;
    }
  return articulation;
}
```

La détection des points d'articulation se fait donc avec un simple parcours en profondeur. Sa complexité est donc en en $O(X + A)$. Il reste à calculer les composantes bi-connexes, ce qui est un petit peu plus délicat, mais réalisable avec la même com-

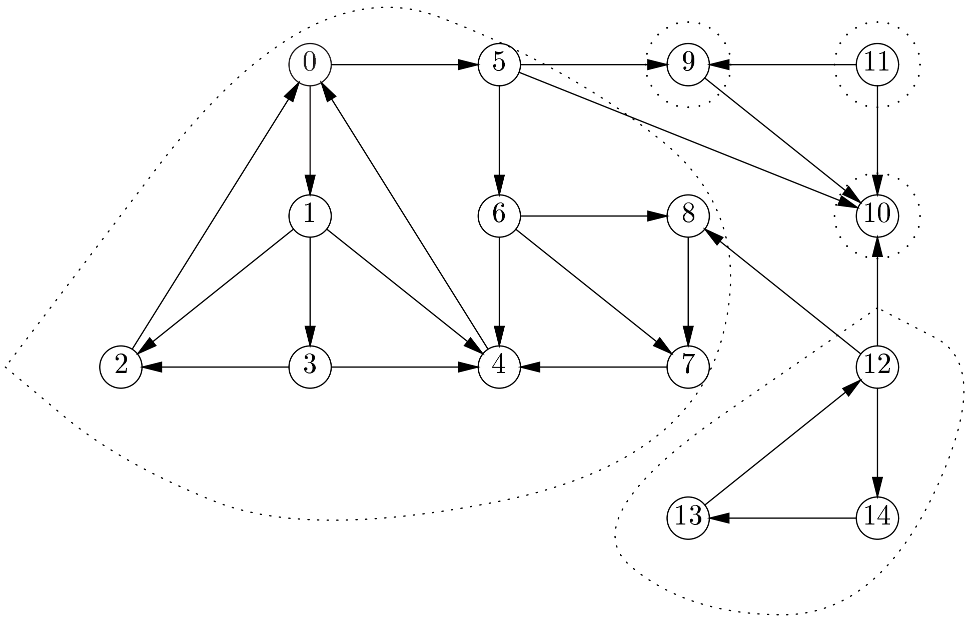


Fig. 1.15 – Composantes fortement connexes du graphe de la figure 1.10

plexité.

Exercice 6 Donner une défintion précise de la notion de composantes biconnexes, et trouver un algorithme pour les calculer.

1.12 Composantes fortement connexes

Dans un graphe orienté, la notion de connexité n'est pas aussi simple que dans un graphe non-orienté, puisque deux sommets x et y peuvent être connectés par un chemin de x à y, sans qu'il existe un chemin de y à x.

Définition 1.14 *Soit $G = (X, A)$ un graphe. Notons $\equiv_G$ la relation d'équivalence suivante entre sommets : $x \equiv_G y$ si $x = y$ ou s'il existe un chemin joignant x à y et un chemin joignant y à x. Les classes d'équivalences définies par $\equiv_G$ sont les* composantes fortement connexes *de G.*

La relation $\equiv_G$ est clairement une relation d'équivalence. La symétrie et la réflexivité sont évidentes. La transitivité résulte de ce que l'on peut concaténer un chemin entre x et y et un chemin entre y et z pour obtenir un chemin entre x et z. Sur la figure 1.15, le graphe comporte 5 composantes fortement connexes, trois ne contiennent qu'un seul sommet, une est constituée d'un triangle et la dernière comporte 9 sommets. Lorsque la relation $\equiv_G$ n'a qu'une seule classe, le graphe est dit *fortement connexe*. Un exemple de graphe fortement connexe est celui des voies de circulation d'une ville en tenant compte des sens uniques.

Un algorithme de recherche des composantes fortement connexes débute nécessairement par un parcours à partir d'un sommet x, les sommets qui n'appartiennent pas à l'arborescence ainsi construite ne sont certainement pas dans la composante fortement connexe de x, mais la réciproque n'est pas vraie : un sommet y qui est dans l'arbores-

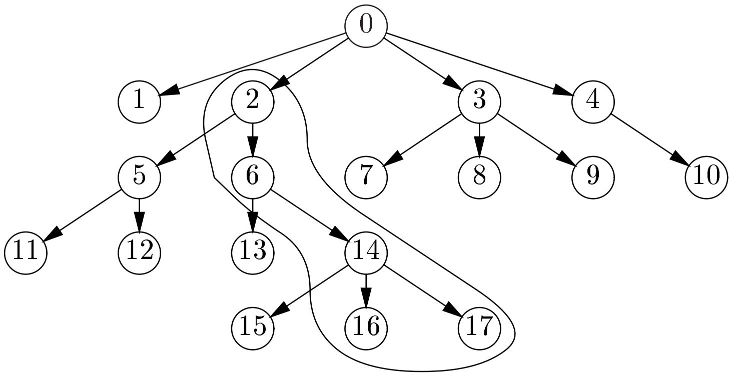


FIG. 1.16 – Un exemple de sous-arborescence

cence issue de x n'est pas nécessairement dans sa composante fortement connexe car il se peut qu'il n'y ait pas de chemin allant de y à x.

Une manière simple de procéder pour le calcul de ces composantes consiste à itérer l'algorithme suivant pour chaque sommet x dont la composante n'a pas encore été construite :

– Déterminer les sommets extrémités de chemins d'origine x, par exemple en utilisant l'algorithme de Trémaux à partir de x.
– Retenir parmi ceux ci les sommets qui sont l'origine d'un chemin d'extrémité x. On peut, pour ce faire, construire le graphe opposé de G obtenu en renversant le sens de tous les arcs de G et appliquer l'algorithme de Trémaux sur ce graphe à partir de x.

Cette manière de procéder est peu efficace lorsque le graphe possède de nombreuses composantes fortement connexes, car on peut être amené à parcourir tout le graphe autant de fois qu'il y a de composantes. Nous allons voir que la construction de l'arborescence de Trémaux issue de x va permettre de calculer toutes les composantes connexes des sommets descendants de x en un nombre d'opérations proportionnel au nombre d'arcs du graphe.

Les liens entre arborescence de Trémaux (Y, T, x) et les composantes fortement connexes sont dus à la proposition suivante.

Définition 1.15 *Une* sous-arborescence *(Y',T',r') de l'arborescence* (Y, T, r) *est une arborescence telle que* $Y' \subset Y$ *et* $T' \subset T$.

Ainsi tout élément de Y' est extrémité d'un chemin d'origine r' et ne contenant que des arcs de T'.

Proposition 1.16 *Soit* $G = (X, A)$ *un graphe. Soient* $x \in X$, *et* (Y, T, x) *une arborescence de Trémaux. Pour tout sommet* u *de* Y, *la composante fortement connexe contenant* u *est une sous-arborescence de* (Y, T, x).

Démonstration Notons $C(u)$ la composante fortement connexe $C(u)$ de u (contenant u). Cette proposition contient en fait deux conclusions ; d'une part elle assure l'existence d'un sommet u_0 dans $C(u)$ tel que tous les éléments de $C(u)$ sont des descendants de u_0 dans (Y, T, x), d'autre part elle affirme que pour tout v de $C(u)$ tous les sommets du chemin de (Y, T, x) joignant u_0 à v sont dans $C(u)$.

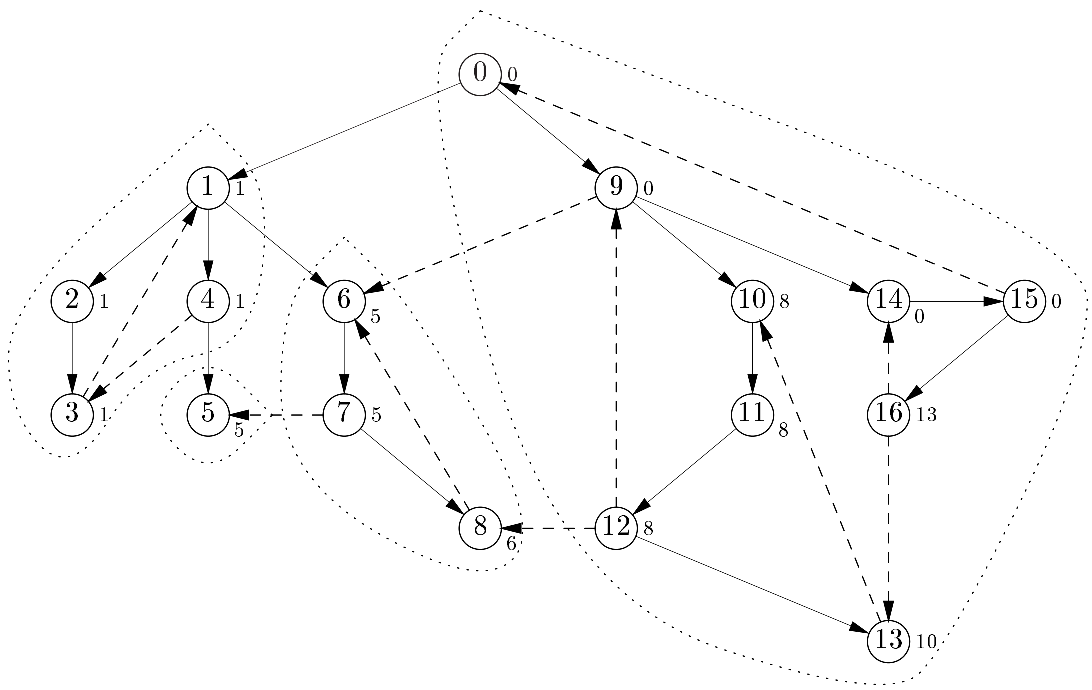


FIG. 1.17 – Les points d'attaches des sommets d'un graphe

La deuxième affirmation est simple à obtenir car, dans un graphe, tout sommet situé sur un chemin joignant deux sommets appartenant à la même composante fortement connexe est aussi dans cette composante. Pour prouver la première assertion, prenons pour u_0 le sommet de plus petit numéro de $C(u)$, et montrons que tout sommet v de $C(u)$ est un descendant de u_0 dans (Y, T, x). Supposons le contraire. Comme v est dans la même composante que u_0, il existe un chemin f d'origine u_0 et d'extrémité v. Soit w le premier sommet de f qui n'est pas un descendant de u_0 dans (Y, T, x) et soit w' le sommet qui précède w dans f. L'arc (w', w) n'est pas un arc de T, ni un arc de descente, c'est donc un arc de retour ou un arc transverse et on a

$$num[u_0] \leq num[w] \leq num[w']$$

L'arborescence (Y, T) étant préfixe on en déduit que w est descendant de u_0 d'où la contradiction cherchée. $\square$

On remarquera qu'un chemin qui conduit d'un sommet y à son point d'attache est ou bien vide (le point d'attache est alors y lui même), ou bien contient une suite d'arcs de T suivis par un arc de retour ou un arc transverse. En effet, une succession d'arcs de T partant de y conduit à un sommet de numéro plus grand que y, d'autre part les arcs de descente ne sont pas utiles dans la recherche du point d'attache, ils peuvent être remplacés par des chemins formés d'arcs de T.

Dans la figure 1.17, on a calculé les points d'attaches des sommets d'un graphe, ceux-ci ont été numérotés dans l'ordre où on les rencontre dans l'algorithme de Trémaux (on suppose pour simplifier que $x = num[x]$ pour tout x) ; le point d'attache est indiqué en petit caractère à droite du sommet en question.

Comme dans le cas de la biconnexité, le calcul des composantes fortement connexes à l'aide des $at(u)$ se fait par un peu d'arithmétique sur la numérotation préfixe et les points d'attache. C'est une conséquence du théorème suivant :

Théorème 1.17 *Si u est un sommet de Y satisfaisant :*
(i) $u = at(u)$
(ii) Pour tout descendant v de u dans (Y, T, x) on a $at(v) < v$
Alors, l'ensemble $desc(u)$ des descendants de u dans (Y, T, x) forme une composante fortement connexe de G.

Démonstration Montrons d'abord que tout sommet de $desc(u)$ appartient à $C(u)$. Soit v un sommet de $desc(u)$, il est extrémité d'un chemin d'origine u, prouvons que u est aussi extrémité d'un chemin d'origine v. Si tel n'est pas le cas, on peut supposer que v est le plus petit sommet de $desc(u)$ à partir duquel on ne peut atteindre u, soit f le chemin joignant v à $at(v)$, le chemin obtenu en concaténant f à un chemin de (Y, T, x) d'origine u et d'extrémité v contient au plus un arc de retour ou transverse ainsi :

$$u = at(u) \leq at(v) < v$$

Comme (Y, T, x) est préfixe, $at(v)$ appartient à $desc(u)$ et d'après l'hypothèse de minimalité il existe un chemin d'origine $at(v)$ et d'extrémité u qui concaténé à f fournit la contradiction cherchée.

Il reste à montrer que tout sommet w de $C(u)$ appartient aussi à $desc(u)$. Un tel sommet est extrémité d'un chemin g d'origine u, nous allons voir que tout arc dont l'origine est dans $desc(u)$ a aussi son extrémité dans $desc(u)$, ainsi tous les sommets de g sont dans $desc(u)$ et en particulier w. Soit $(v_1, v_2) \in A$ un arc tel que $v_1 \in desc(u)$, si $v_2 > v_1$, v_2 est un descendant de v_1 il appartient donc à $desc(v)$; si $v_2 < v_1$ alors le chemin menant de u à v_2 en passant par v_1 contient exactement un arc de retour ou transverse, ainsi :

$$u = at(u) \leq v_2 < v_1$$

et la préfixité de (Y, T, x) implique $v_2 \in desc(u)$. □

Remarquons qu'il existe toujours un sommet satisfaisant les conditions du théorème. En effet, si x est la racine de (Y, T, x), on a $at(x) = x$. Si x satisfait *(ii)*, l'ensemble Y en entier constitue une composante fortement connexe. Sinon il existe un descendant y de x tel que $y = at(y)$. En répétant cet argument plusieurs fois et puisque le graphe est fini, on finit par obtenir un sommet satisfaisant les deux conditions.

La recherche des composantes fortement connexes est alors effectuée par la détermination d'un sommet u tel que $u = at(u)$, obtention d'une composante égale à $desc(u)$, suppression de tous les sommets de $desc(u)$ et itération des opérations précédentes jusqu'à obtenir tout le graphe.

Sur la figure 1.17, on peut se rendre compte du procédé de calcul. Il y a 4 composantes fortement connexes, les sommets u satisfaisant $u = at(u)$ sont au nombre de 3, il s'agit de $0, 1, 5$. La première composante trouvée se compose du sommet 5 uniquement, il est supprimé et le sommet 6 devient alors tel que $u = at(u)$. Tous ses descendants forment une composante fortement connexe $\{6, 7, 8\}$. Après leur suppression, le sommet 1 satisfait $u = at(u)$ et il n'a plus de descendant satisfaisant la même relation. On trouve ainsi une nouvelle composante $\{1, 2, 3, 4\}$. Une fois celle-ci supprimée, 0 est le seul sommet qui satisfait la relation $u = at(u)$ d'où la composante $\{0, 9, 10, 11, 12, 13, 14, 15, 16\}$.

L'algorithme ci-dessous calcule en même temps $at(u)$ pour tous les descendants u de x et obtient successivement toutes les composantes fortement connexes de $desc(x)$. Il utilise le fait que la suppression des descendants de u lorsque $u = at(u)$ ne modifie pas les calculs des $at(v)$ en cours.

```
static int id = -1;

static void imprimerCompFConnexes (Graphe g) {
  int n = g.succ.length;
  int[ ] num = new int[n];
  for (int x = 0; x < n; ++x) num[x] = -1;
  for (int x = 0; x < n; ++x) {
    if (num[y] == -1)
      imprimerCompF (g, x, num);
  }
}

static int imprimerCompF (Graphe g, int x, int[ ] num) {
  num[x] = ++numOrdre; Pile.ajouter(x, p);
  int min = id;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val; int m;
    if (num[y] == -1)
      m = imprimerCompF (g, y, num);
    else m = num[y];
    min = Math.min (min, m);
  }
  if (min == num[x]) {
    int y;
    do {
      y = Pile.supprimer(p);
      System.out.print (y + " ");
      num[y] = g.succ.length;
    } while (y != x);
    System.out.println();
  }
  return min;
}
```

Ce bel algorithme, dû à Tarjan, illustre la puissance de la notion de parcours en profondeur ; il a une complexité en $O(X + A)$.

1.13 Programmes en OCaml

```
let m =
 [| [| 1.0; 0.0; 0.0 |];
    [| 0.0; 1.0; 0.0 |];
    [| 0.0; 0.0; 1.0 |] |];;

let g =
 [| [| 0; 1; 1; 0; 0; 0 |];
    [| 0; 0; 1; 1; 0; 1 |];
    [| 0; 0; 0; 0; 0; 1 |];
    [| 0; 0; 0; 0; 1; 0 |];
    [| 0; 1; 0; 0; 0; 0 |];
    [| 0; 0; 0; 1; 0; 0 |] |];;

let nb_lignes m = Array.length m
and nb_colonnes m =
```

```
  if Array.length m = 0 then failwith "nb_colonnes"
  else Array.length m.(0);;

(* Calcule le produit des matrices a et b *)
let multiplier a b =
  if nb_colonnes a <> nb_lignes b
  then failwith "multiplier" else
  let c =
     Array.make_matrix (nb_lignes a) (nb_colonnes b) 0 in
   for i = 0 to nb_lignes a - 1 do
     for j = 0 to nb_colonnes b - 1 do
      let cij = ref 0 in
      for k = 0 to nb_colonnes a - 1 do
        cij := a.(i).(k) * b.(k).(j) + !cij;
      done;
      c.(i).(j) <- !cij
    done
  done;
  c;;
```

```
(* Calcule la somme des matrices a et b *)
let ajouter a b =
  if nb_colonnes a <> nb_lignes b
  then failwith "ajouter" else
  let c =
    Array.make_matrix (nb_lignes a) (nb_colonnes b) 0 in
  for i = 0 to nb_lignes a - 1 do
    for j = 0 to nb_colonnes b - 1 do
      c.(i).(j) <- a.(i).(j) + b.(i).(j)
    done
  done;
  c;;
```

```
(* Élève la matrice m à la puissance i *)
let rec puissance m i =
  match i with
  | 0 -> failwith "puissance"
  | 1 -> m
  | n -> multiplier m (puissance m (i - 1));;
```

```
let nombre_de_chemin_de_longueur n i j m =
  (puissance m n).(i).(j);;

let sigma i m =
  let rec pow i mp =
    match i with
    | 1 -> mp
    | n -> ajouter mp (pow (i - 1) (multiplier m mp)) in
  pow i m;;

let existe_chemin i j m =
  (sigma (nb_colonnes m) m).(i).(j) <> 0;;
```

```
let phi m x =
 for u = 0 to nb_colonnes m - 1 do
   if m.(u).(x) = 1 then
     for v = 0 to nb_colonnes m - 1 do
```

```
      if m.(x).(v) = 1 then m.(u).(v) <- 1
    done
 done;;

let fermeture_transitive m =
  let résultat =
    Array.make_matrix (nb_lignes m) (nb_colonnes m) 0 in
  for i = 0 to nb_lignes m - 1 do
    for j = 0 to nb_colonnes m - 1 do
     résultat.(i).(j) <- m.(i).(j)
    done
  done;
  for x = 0 to nb_colonnes m - 1 do
    phi résultat x done;
  résultat;;
```

```
(* Tableaux de successeurs *)

type graphe_point = (int list) array;;
let omega = -1;;

let succ_of_mat m =
  let nb_max_succ = ref 0 in
  for i = 0 to nb_lignes m - 1 do
    nb_max_succ := 0;
    for j = 0 to nb_colonnes m - 1 do
      if m.(i).(j) = 1 then
        nb_max_succ := max j !nb_max_succ
    done;
  done;
  let succ =
    Array.make_matrix (nb_lignes m) (!nb_max_succ + 1) 0 in
  let k = ref 0 in
  for i = 0 to nb_lignes m - 1 do
    k := 0;
    for j = 0 to nb_colonnes m - 1 do
      if m.(i).(j) = 1 then begin
        succ.(i).(!k) <- j;
        incr k
      end
    done;
    succ.(i).(!k) <- omega
  done;
  succ;;
```

```
(* Listes de successeurs *)
let liste_succ_of_mat m =
  let gpoint = Array.make (nb_colonnes m) [] in
  for i = 0 to nb_lignes m - 1 do
    for j = 0 to nb_colonnes m - 1 do
      if m.(i).(j) = 1
      then gpoint.(i) <- j :: gpoint.(i)
    done
  done;
  gpoint;;
```

```
let numéro = Array.make (Array.length succ) (-1);;
let num = ref (-1);;
```

```
let rec num_prefixe k = begin
  incr num;
  numéro.(k) <- !num;
  List.iter
    (function x -> if numéro.(x) = -1 then
         num_prefixe (x))
    succ.(k)
  end;;

let numPrefixe() = begin
  Array.iter
    (function x -> if numéro.(x) = -1 then
         num_prefixe (x))
    numéro
  end;;
```

```
let num_largeur k =
  let f = file_vide() in begin
  fajouter k f;
  while not (fvide q) do
    let k = fvaleur(f) in begin
    fsupprimer f;
    incr num;
    numéro.(k) <- !num;
    List.iter
      (function x -> if numéro.(x) = -1 then
         begin
         fajouter x f;
         numéro.(x) <- 0
         end)
      succ.(k)
    end
  done
  end;;

let numLargeur() = begin
  Array.iter
    (function x -> if numéro.(x) = -1 then
         num_largeur (x))
    numéro
  end;;
```

```
(* calcule la composante connexe
   de k et retourne son point d'attache *)
let rec comp_connexe k = begin
    incr num; numéro.(k) <- !num;
    Pile.ajouter k p;
    let min = ref !num in begin
      List.iter
        (function x ->
          let m = if numéro.(x) = -1 then
                comp_connexe (x)
            else
                numéro.(x)
            in if m < !min then
                min := m)
         succ.(k);
      if !min = numéro.(k) then
```

```
      (try while true do
          printf "%d " (Pile.sommet p);
          numéro.(Pile.sommet p) <- max_int;
          Pile.supprimer p;
          if (Pile.sommet p) = k then raise Exit
          done
      with Exit -> printf "\n");
    !min
    end
  end;;

let compConnexe() = begin
  Array.iter
    (function x -> if numéro.(x) = -1 then
         comp_connexe (x))
    numéro
  end;;
```

Chapitre 2

Analyse Syntaxique

Un compilateur transforme un programme écrit en langage évolué en une suite d'instructions élémentaires exécutables par une machine. La construction de compilateurs a longtemps été considérée comme une des activités importantes en informatique, elle a suscité le développement de nombreuses techniques et théories maintenant classiques. La compilation d'un programme est réalisée en au moins trois phases, la première (analyse lexicale) consiste à découper le programme en petites entités : opérateurs, mots réservés, variables, constantes numériques, alphabétiques, etc. La deuxième phase (analyse syntaxique) consiste à expliciter la structure du programme sous forme d'un arbre, appelé arbre de syntaxe abstraite, chaque noeud de cet arbre correspond à un opérateur et ses fils aux opérandes sur lesquels il agit. La troisième phase (génération de code) construit la suite d'instructions du micro-processeur à partir de l'arbre de syntaxe abstraite ; elle peut donner lieu à des développements sophistiqués, constituant toujours un domaine de recherche très actif.

Nous nous limiterons ici à une vision simple des deux premières étapes. Elle permettent d'illustrer l'utilisation des notions d'automates (analyse lexicale) et de grammaires (analyse syntaxique). En outre, l'analyse syntaxique fait partie des nombreuses situations où l'on transforme une entité, qui se présente sous une forme plate et difficile à manipuler, en une forme structurée adaptée à un traitement efficace. Le calcul symbolique ou formel, le traitement automatique du langage naturel constituent d'autres exemples. Notre but n'est pas de donner ici toutes les techniques permettant d'écrire un analyseur syntaxique, mais de suggérer à l'aide d'exemples comment il faudrait faire. L'ouvrage de base pour l'étude de la compilation est celui de A. Aho, R. Sethi et J. Ullman [21]. On peut aussi se référer aux premiers chapitres de [30] pour une présentation plus théorique.

2.1 Caractères

Les caractères constituent souvent dans un langage de programmation un type primitif. C'est bien le cas en Java. Les constantes de type caractère sont notées entre apostrophes `'a'`, `'b'`, `'c'`, etc. Les caractères spéciaux sont définis avec la barre renversée `'\n'` (*line-feed*), `'\r'` (retour charriot), `'\t'` (tabulation), `'\\'` (*backslash*), `'\''` (apostrophe), `'\"'` (guillemet). La différence entre *line-feed* et retour-charriot est hélas subtile, car très dépendante du système d'exploitation. Dans le système Unix, le passage à la ligne suivante se fait avec le seul caractère *line-feed*. En MacOS, c'est retour-charriot. En Windows, c'est la séquence retour-charriot suivi de *line-feed*, une remarque à savoir pour le seul caractère figurant sur chaque ligne ! Souvent, les éditeurs de texte lisent indifféremment les trois formats.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	nul	soh	stx	etx	eot	enq	ack	bel	`\\`	`\t`	`\n`	vt	np	`\r`	so	si
10	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
20		!	”	#	$	%	&	’	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	¡	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	ˆ	_
60	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	&	}	~	del

FIG. 2.1 – Le code ASCII en hexadécimal

Selon les langages de programmation, la distinction entre caractères et entiers est plus ou moins stricte. En Java, les caractères forment un sous-ensemble des entiers. Ce sont des entiers courts non-signés. Ils occupent complètement un mot de 16 bits, et ne peuvent être convertis implicitement qu'à des entiers signés sur 32 bits de type *int*.

La fonction associant à chaque caractère un entier est la fonction de codage des caractères. Plusieurs codes ont successivement existé ; ces codes deviennent plus complexes avec l'internationalisation des langages de programmation pour représenter les caractères par exemple en indi, en chinois, en coréen, ou en alphabet cyrillique, ainsi que les caractères accentués du français, de l'espagnol et des langages d'Europe centrale. Le consortium *Unicode* et l'ISO (*International Organization for Standardization*) ont défini un code universel, standard ISO 10646-1 utilisant 65534 valeurs.

Les 127 premières valeurs constituent le code ASCII (*American Standard Codes for Information Interchange*) des caractères américains (c'est-à-dire les caractères latins non-accentués) donné en hexadecimal par la table 2.1. On remarque que les codes des chiffres sont des valeurs consécutives, ainsi que pour les lettres minuscules et majuscules. Les 20 premières valeurs, non imprimables, peuvent être obtenues en appuyant simultanément sur la touche `CTRL` (contrôle) du clavier et sur celle du caractère correspondant quatre lignes plus bas dans le tableau. Ainsi *nul*, *soh*, `\t`, `\n` s'obtiennent en effectuant `CTRL-@`, `CTRL-A`, `CTRL-I`, `CTRL-J`.

Les 256 premières valeurs de l'Unicode constituent le code ISO 8859-1 (Latin-1) avec les lettres accentuées ou les caractères particuliers latins.

En fait, l'Unicode a maintenant dépassé les 16 bits, sa nouvelle norme (codage ISO 10646-2) demande 31 bits pour inclure encore plus de langues. Malheureusement, beaucoup de programmes utilitaires des vieux systèmes d'exploitation tels que le système Unix, conçu en 1970, (ou son incarnation publique Linux), utilisent la représentation ASCII ; par exemple, les caractères des noms de fichiers ne peuvent dépasser les 7 bits de l'ASCII. Pour rester compatible (compatibilité ascendante), une représentation en longueur variable, l'UTF-8, a été définie, comme indiqué sur la table 2.2. En UTF-8, les codes ASCII sont inchangés et tiennent toujours sur un octet ; les codes ISO-Latin tiennent sur deux octets (le premier commençant par les 3 bits 110 ; le deuxième par les 2 bits 10 ; la valeur significative tenant facilement sur les 11 bits restants) ; les autres codes tiennent sur trois octets (le premier commençant par 1110 ; les deux autres par 10 ; la valeur significative tenant aussi facilement sur les 16 bits restants). Le nombre des premiers bits valant 1 donnent le nombre d'octets utilisés pour le cas non-ASCII ;

Unicode	Code UTF-8		
0000 – 007f	0xxxxxxx		
0080 – 07ff	110xxxxx	10xxxxxx	
0800 – ffff	1110xxxx	10xxxxxx	10xxxxxx

Fig. 2.2 – Le code de longueur variable UTF-8 en binaire

la garde 10 en tête de chacun des autres octets permet la synchronisation en cas de perte d'octets.

En Java, quelques fonctions permettent de caractériser les caractères sans avoir à considérer leur code. Par exemple *isLetter*, *isDigit*, *isLetterOrDigit* donnent un résultat booléen signifiant que leur argument est une lettre, un chiffre ou l'un des deux. On pourrait les définir ainsi (dans le cas ASCII) ;

```
static boolean isDigit (char c) {
  return '0' <= c && c <= '9';
}
static boolean isLetter (char c) {
  return 'a' <= c && c <= 'z' && 'A' <= c && c <= 'Z';
}
static boolean isLetterOrDigit (char c) {
  return isLetter(c) || isDigit(c);
}
```

La fonction inverse du codage prend une valeur entière et essaie de la traduire en caractère. Il faut faire attention à ne pas confondre les deux valeurs. Ainsi le caractère correspondant au chiffre x sera donné par la formule `'0'` $+ x$, c'est à dire les valeurs hexadécimales `c0` pour 0, `c1` pour 1, `c2` pour 2, etc.

2.2 Chaînes de caractères

Les chaînes de caractères sont en première approximation des tableaux de caractères. Mais, souvent leur représentation est plus compacte. En Java, les chaînes de caractères sont représentés par des objets de la classe *String*. La longueur de la chaîne s est donnée par *x.length()*. Attention, c'est un appel à la méthode *length* (avec des parenthèses) et non un champ de donnée comme dans le cas des tableaux. Quelques autres méthodes utiles permettent d'écrire *s.equals(t)* pour tester l'égalité de s et t, ou *s.indexOf(c)* pour donner la première occurence de c dans s, ou *s.charAt(i)* pour rendre le caractère à la position i dans s. L'opérateur + produit la concaténation de deux chaînes. Ainsi $s + t$ est la chaîne constituée de s puis de t. Si l'une des deux opérandes n'est pas une chaîne, l'autre est transformée en chaîne de caractère en lui appliquant la méthode *toString* (de sa classe). Enfin, les chaînes de caractères sont *immuables*, c'est-à-dire des constantes non modifiables. Ceci simplifie le partage des chaînes de caractères, par exemple, entre plusieurs processus concurrents (cf. le chapitre sur la concurrence).

L'expression *Integer.parseInt(s)* associe à la chaîne de caractères s l'entier qu'elle dénote en décimal. Symétriquement, $x +$ `""` produit la chaîne de caractères correspondant à la représentation décimale de x. La fonction *parseInt* peut être définie par :

```
static int parseInt (String s) throws NumberFormatException {
  int r = 0;
```

```
    for (int i = 0; i < s.length(); ++i) {
      if (!Character.isDigit (s.charAt(i)))
        throw new NumberFormatException();
      r = 10 * r + s.charAt(i) - '0';
    }
    return r;
  }
```

La fonction *parseInt* ne fait que calculer, pour la chaîne s constituée des caractères s_0, s_1, $\ldots s_p$, le polynôme $\Sigma_{i=0}^{p} s_i 10^{p-i}$ par la méthode de Horner. Réciproquement, la fonction symétrique, appelons-la *integerToString*, s'écrit :

```
  static String integerToString (int x) {
    String s = "";
    while (x != 0) {
      s = s + ('0' + x % 10);
      x = x / 10;
    }
    return s;
  }
```

Cette dernière fonction existe aussi en standard en Java. On l'obtient en calculant l'expression *new Integer(x).toString()* ou plus simplement `x+""` comme déjà mentionné.

Exercice 7 Ecrire un programme pour la méthode *toString* de la classe *Integer*.

Exercice 8 Ecrire une fonction *integerToString* avec deux arguments x et b retournant la chaîne de caractères représentant l'entier x dans la base b.

Toutefois, la fonction *integerToString* n'est pas très efficace, puisqu'à chaque itération une nouvelle chaîne de caractères s est construite avec seulement un caractère de plus. Au total, elle prend un espace-mémoire en $O(\ell^2)$ si on pose $\ell = \log_{10} x$. Il est impossible de faire autrement avec les variables de la classe *String* dont les valeurs sont imuables. Fort heureusement, la classe *StringBuffer* permet de manipuler des chaînes de caractères modifiables en réservant un espace mémoire paramétrable à leur création. Le programme précédent qui s'écrit comme suit prend maintenant un espace-mémoire en $O(\ell)$:

```
  static String integerToString (int x) {
    StringBuffer s = new StringBuffer();
    while (x != 0) {
      s = s.append ('0' + x % 10);
      x = x / 10;
    }
    return new String(s);
  }
```

2.3 Alphabets, mots, langages

D'un point de vue théorique, une chaîne de caractères est appelé un *mot*; les caractères sont des *lettres*; l'ensemble des lettres constitue un *alphabet*. Autrement dit, si A est un alphabet, un mot construit sur A est une suite finie $f = a_1 a_2 \ldots a_n$ de lettres prises dans A ($n \geq 0$) ; l'entier n est la longueur de f. On note par ϵ le *mot vide*, c'est le mot de longueur 0. Le *produit* de deux mots f et g est obtenu en écrivant f puis g à la suite, on le note fg. Donc la longueur de fg est égale à la somme des longueurs de f et de g. Un mot f est un *facteur* de g s'il existe deux mots g' et g'' tels que $g = g'fg''$;

f est *facteur gauche* de g si $g = fg''$; c'est un *facteur droit* si $g = g'f$. L'ensemble des mots sur l'alphabet A est noté A^*. On remarque qu'on a trivialement $\epsilon f = f\epsilon = f$ et que $(fg)h = f(gh)$ (lois d'un monoïde).

Exemple 2.1 *Mots sans carré.* Soit l'alphabet $A = \{a, b, c\}$. On construit la suite de mots suivante $f_0 = a$, et, pour $n \geq 0$, on obtient f_{n+1} à partir de f_n en remplaçant a par abc, b par ac et c par b. Ainsi :

$$f_0 = a \qquad f_1 = abc \qquad f_2 = abcacb \qquad f_3 = abcacbabcbac$$

Il est assez facile de voir que f_n est un facteur gauche de f_{n+1} pour $n \geq 0$, et que la longueur de f_n est $3 \times 2^{n-1}$ pour $n \geq 1$. On peut aussi montrer que, pour tout n, aucun facteur de f_n n'est un carré, c'est à dire que, si gg est un facteur de f_n, alors $g = \epsilon$. Remarquons à ce propos que, si A est un alphabet composé des deux lettres a et b, les seuls mots sans carré sont a, b, ab, ba, aba, bab. La construction ci-dessus, montre l'existence de mots sans carré de longueur arbitrairement grande sur un alphabet de trois lettres.

Exemple 2.2 *Expressions préfixées.* Les expressions arithmétiques en notation (polonaise) préfixe peuvent être considérées comme des mots sur l'alphabet $A = \{\texttt{+}, *, a\}$, on remplace tous les nombres par la lettre a pour en simplifier l'écriture. En voici deux exemples,

$$f = *aa \qquad g = *\texttt{+}a*aa\texttt{+}*aa*aa$$

Exemple 2.3 *Un mini-langage de programmation.* Considérons l'alphabet A suivant, où les « lettres » sont des mots sur un autre alphabet :

$$A = \{\texttt{begin}, \texttt{end}, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{while}, \texttt{do}, \texttt{;}, \texttt{p}, \texttt{q}, \texttt{x}, \texttt{y}, \texttt{z}\}$$

Alors

$$f = \texttt{while p do begin if q then x else y ; z end}$$

est un mot de longueur 13, qui peut se décomposer en

$$f = \texttt{while p do begin}\, g \texttt{ ; z end} \qquad g = \texttt{if q then x else y}$$

Un *langage* est un ensemble de mots. Par exemple, si $A = \{a, b\}$, le langage $L = \{a^n b^n \mid n \geq 0\}$ des mots contenant n fois la lettre a suivis d'autant de fois de b. On peut aussi considérer le langage des L' des mots contenant autant de fois a que de fois b. Bien sûr, on a $L \subset L'$. Si $A' = \{\texttt{(}, \texttt{)}\}$, on peut considérer le langage D des mots sur A' bien parenthésés, c'est-à-dire avec exactement une parenthèse fermante pour chaque parenthèse ouvrante. Par exemple, $\texttt{()(()())} \in D$. Si on remplace la parenthèse ouvrante par la lettre a et la parenthèse fermante par la lettre b, on obtient un langage L'' bien parenthésé sur A, et on a $L \subset L'' \subset L' \subset A^*$.

Le produit de mots peut être étendu au produit de langages. Si L et L' sont deux langages sur l'alphabet A, le produit LL' désignera l'ensemble des mots obtenus par produit d'un mot de L et de L'.

$$LL' = \{fg \mid f \in L,\ g \in L'\}$$

On notera aussi L^n pour le produit $LL \ldots L$ itéré n fois. Par convention $L^0 = \{\epsilon\}$. Enfin l'opération étoile qui désigne les mots formés par produits de mots de L est définie par :

$$L^* = L^0 \cup L \cup L^2 \ldots \cup L^n \ldots$$

Remarquons que cette notation est bien cohérente avec l'écriture utilisée, A^*, pour désigner tous les mots sur l'alphabet A.

2.4 Expressions régulières

Les expressions régulières permettent de dénoter simplement un ensemble de mots. Par exemple pour désigner tous les mots du langage $\{a^n b \mid n \geq 0\}$, on écrit a^*b, c'est-à-dire les mots contenant un nombre quelconque (éventuellement nul) de a suivi d'un b. De même b^*a représente l'ensemble des mots commençant par un nombre quelconque de b suivi d'un seul a. Pour représenter l'union de ces deux ensembles, on écrit a^*b+b^*a.

Définition 2.1 *Soit A un alphabet. Une* expression régulière *est toute expression formée à partir du mot vide ϵ, des lettres de A avec les opérations produit, somme et étoile. Le langage $[\![e]\!]$ dénoté par l'expression régulière e est défini récursivement par :*

$$\begin{array}{lll} [\![\epsilon]\!] = \{\epsilon\} & [\![a]\!] = \{a\} & [\![(e)]\!] = [\![e]\!] \\ [\![ee']\!] = [\![e]\!][\![e']\!] & [\![e^*]\!] = [\![e]\!]^* & [\![e + e']\!] = [\![e]\!] \cup [\![e']\!] \end{array}$$

Parfois l'opération somme $e + e'$ est aussi écrite avec une barre verticale, donnant $e|e'$. La première notation est plus utilisée en informatique théorique, la deuxième dans les manuels de description des langages de programmation. Nous utilisererons indistinctement les deux notations. Une autre notation fréquente est e^+ pour désigner e^* sans la chaîne vide ϵ. Ainsi $e^* = \epsilon + e^+$.

Exemple 2.4 *Mots contenant un facteur donné.* Soit $A = \{a, b\}$ et $u \in A^*$. L'ensemble des mots ayant u en facteur s'écrit $(a+b)^*u(a+b)^*$. Par exemple, si $u = aab$, on obtient $(a + b)^*aab(a + b)^*$.

Exemple 2.5 *Mots contenant un nombre pair de lettres.* Sur l'alphabet $A = \{a, b\}$. L'expression régulière $(b^*ab^*a)^*$ désigne les mots contenant un nombre pair de a. On peut aussi utiliser $(b+ab^*a)^*$ pour ce même ensemble. Maintenant, si on exige un nombre pair de a et de b, on arrive à l'expression pas si évidente $((a + ba(aa)^*b)(b(aa)^*b)^*(a + ba(aa)^*b) + b(aa)^*b)^*$.

Exemple 2.6 *Noms de fichiers.* Dans les systèmes de fichiers, on opère souvent sur tous les fichiers dont le nom comporte un suffixe donné, par exemple`.html`. On les désigne par une expression régulière de la forme `*.html`. Le symbole `?` est aussi parfois utilisé. Formellement `?` est un *joker* dénotant l'alphabet de toutes les lettres de code ASCII, et `*` est une abbréviation pour $(\texttt{?})^*$. Ainsi `a*.java` désigne tous les fichiers dont le nom commence par un a et de suffixe `.java` ; de même, `.???*` dénote tous les fichiers dont le nom a au moins 3 lettres commençant par un point.

Exemple 2.7 *Identificateurs et nombres.* Soit $A = \{a, b, \ldots z\}$ et $C = \{0, 1, 2, \ldots 9\}$. Ecrivons $[a - z]$ pour dénoter A et $[0 - 9]$ pour C. Un nom de variable ou de fonction (un identificateur) est représenté par l'expression régulière $[a - z]([a - z] + [0 - 9])^*$. Une constante entière est dénotée par $[0 - 9]^+$. Donc un identificateur est une suite de chiffres ou de lettres commençant toujours par une lettre ; un nombre est une suite non vide de chiffres.

Exemple 2.8 *Motifs de filtrage.* Dans les systèmes informatiques, les expressions régulières sont aussi utilisées pour retrouver des motifs dans des fichiers, par exemple pour imprimer les lignes contenant le motif (comme le fait la commande *grep* du système Unix – g*et* r*egular* e*xpression*). On utilise des caractères fictifs `^` et `$` pour dénoter le début et la fin de ligne. Alors les lignes vides sont obtenues par le motif `"^$"`; les lignes blanches par `"^␣*$"`; une suite non vide de caractères blancs est `"␣␣*"`; etc. Ici le sens de $*$ n'est pas le même que pour les noms de fichiers; $a*$ a le sens normal a^* des expressions régulières.

Les expressions régulières ont fait l'objet d'études intensives, leur structure est bien plus riche qu'on ne pourrait le penser a priori. Elles n'ont pas vraiment de représentations canoniques. Mais, si on rajoute 0 pour $\emptyset$ et si on pose $1 = \epsilon$ et $e \leq e'$ pour $e + e' = e'$, on obtient un certains nombres de lois définissant ce qu'on appelle une algèbre de Kleene :

$$
\begin{array}{rlrl}
(1) & e + f = f + e & & \\
(2) & e + (f + g) = (e + f) + g & & \\
(3) & e + 0 = e & & \\
(4) & e + e = e & & \\
(5) & e(fg) = (ef)g & & \\
(6) & 1\,e = e\,1 = e & & \\
(7) & e(f + g) = ef + eg & & \\
(8) & (e + f)g = eg + fg & & \\
(9) & 0\,e = e\,0 = 0 & & \\
(10) & 1 + ee^* = e^* & & \\
(11) & 1 + e^*e = e^* & & \\
(12) & f + eg \leq g \;\Rightarrow\; e^*f \leq g & (12') & eg \leq g \;\Rightarrow\; e^*g \leq g \\
(13) & f + ge \leq g \;\Rightarrow\; fe^* \leq g & (13') & ge \leq g \;\Rightarrow\; ge^* \leq g
\end{array}
$$

En fait, toutes les égalités entre expressions régulières peuvent être obtenues à partir de ces 13 équations (Les équations 12-12' et 13-13' sont équivalentes). Par exemple, pour montrer que $a^*a^* = a^*$, on note que $1 + aa^* = a^*$ par (10). D'où $aa^* \leq a^*$ par définition de $\leq$. Donc $a^*a^* \leq a^*$ par (12'). Réciproquement, $a^*a^* = (1 + aa^*)a^*$ par (10). D'où $a^*a^* = a^* + aa^*a^*$ par (8) et (6). Donc $a^* \leq a^*a^*$.

Exercice 9 Montrer qu'on a dans les algèbres de Kleene : $e \leq e' \leq e \;\Leftrightarrow\; e = e'$.

Exercice 10 Montrer en utilisant les lois des algèbres de Kleene que $(a^*)^* = a^*$, $(a^*b)^*a^* = (a + b)^*$, $a(ba)^* = (ab)^*a$, et $a^* = (aa)^* + a(aa)^*$.

Exercice 11 Montrer les équations précédentes par des raisonnements directs sans utiliser les algèbres de Kleene.

Exercice 12 Montrer que $e = f + ge$ implique que $e = g^*f$.

2.5 Analyse lexicale

On veut donc extraire un flot de lexèmes dans la très longue chaîne de caractères constituant le texte d'un programme. On élimine tous les caractères inutiles ou redondants. Par exemple, les caractères blancs (espace, tabulation, retour-charriot, *line-feed*) sont souvent inutiles; un seul suffit plutôt qu'une bonne dizaine consécutifs. De la même manière, les commentaires sont inutiles (pour compiler un programme). Un lexème est une entité importante pour la compilation telle qu'un identificateur, une constante numérique, un opérateur, une chaîne de caractères, une contante caractère, etc.

Comme vu dans l'exemple 2.7, un identificateur ou une constante entière peuvent être définis par des expressions régulières. Il en va de même pour tous les autres lexèmes. Un nombre flottant est une suite de chiffres suivis éventuellement d'un point (.) et d'une suite de chiffres. Pour la notation de l'ingénieur, c'est un peu plus compliqué puisqu'il faut expliquer les formats de la partie significative et de la partie exposant. Une chaîne de caractères commence et finit par un guillemet ("); à l'intérieur c'est une suite quelconque de caractères différent de guillemet. Toutefois après une barre renversée (\) on peut aussi mettre un guillemet à l'intérieur de la chaîne.

Les définitions de lexèmes sous forme d'expressions régulières sont données par :

$$\begin{array}{rcl}
\mathit{lettre} & = & a|b|\ldots z|A|B|\ldots Z|_ \\
\mathit{chiffre} & = & 0|1|2|3|4|5|6|7|8|9 \\
\mathit{identificateur} & = & \mathit{lettre}(\mathit{lettre}|\mathit{chiffre})^* \\
\mathit{nombre} & = & \mathit{chiffre}^+ \\
\mathit{nombre_flottant} & = & \mathit{nombre}(\epsilon \mid .\mathit{nombre}) \\
\mathit{chaine} & = & \texttt{"}((\sim \texttt{"}) + \backslash\texttt{"})^*\texttt{"}
\end{array}$$

en adoptant la notation $\sim c$ pour désigner les caractères différents de c. Avec de telles définitions, nous pouvons enfin penser à écrire un programme pour rechercher des lexèmes dans une très longue chaîne de caractères. Nous définissons d'abord la représentation des lexèmes dans notre langage de programmation favori. Dans une deuxième phase, on produira le programme correspondant à la description par expressions régulières.

Les lexèmes sont représentés par des objets de la classe *Lexeme* suivante avec trois champs de données : le champ *nature* indique si le lexème est un nombre, un identificateur, un opérateur ou un délimiteur comme parenthèse ou crochet ou la fin de fichier ; le champ *valeur* stocke la valeur d'un entier dans le cas où le lexème est un nombre ; le champ *nom* est la chaîne de caractères représentant le nom d'un identificateur dans le cas où le lexème est un identificateur. Ces trois champs ne sont pas utiles simultanément, mais, pour simplifier, nous considérons ces trois champs pour chaque lexème.

```
class Lexeme {
  final static int L_Nombre = 0, L_Id = 1, L_Plus = '+', L_Moins = '-',
        L_Mul = '*', L_Div = '/', L_ParG = '(', L_ParD = ')',
        L_CroG = '[', L_CroD = ']', L_EOF = -1;
  int nature;
  int valeur;
  String nom;

  Lexeme (int i) { nature = L_Nombre; valeur = i; }
  Lexeme (String s) { nature = L_Id; nom = s; }
  Lexeme (char op) { nature = op; }
}
```

La fonction *Lexeme.suivant* retourne le prochain lexème dans le flot d'entrée de l'analyseur lexical. Elle saute les caractères blancs (espace, tabulation, *line-feed*, retour-chariot) et retourne, selon le premier caractère non-blanc rencontré, un lexème identificateur, nombre entier ou symbole opérateur-délimiteur. La définition des fonctions *ident* ou *nombre* suit exactement la valeur de l'expression régulière définissant les lexèmes identificateur ou nombre. La fonction *avancer* passe au caractère suivant dans le flot d'entrée et le stocke dans la variable globale *c*.

```
static char c;   // caractère courant

static Lexeme suivant() {
  sauterLesBlancs();
  if (Character.isLetter(c)) return new Lexeme (ident());
  else if (Character.isDigit(c)) return new Lexeme (nombre());
  else switch (c) {
  case '+': case '-': case '*': case '/':
  case '(': case ')': avancer(); return new Lexeme (c);
  default: throw new Error ("Caractère illégal");
 }
}

static void avancer() { c = in.readChar(); }

static void sauterLesBlancs() {
  while ( Character.isWhitespace (c) )
    avancer();
}

static String ident() {
  StringBuffer r;
  while (Character.isLetterOrDigit (c)) {
    r = r.append (c);
    avancer();
  }
  return new String(r);
}

static int nombre() {
  int r = 0;
  while (Character.isDigit (c)) {
    r = r * 10 + c - '0';
    avancer();
  }
  return r;
}
```

Exercice 13 Compléter le programme pour chercher des lexèmes pour les nombres flottants et les chaînes de caractères de telle manière que sur le texte suivant

```
class PremierProg {
  public static void main (String[ ] args){
    System.out.println ("Bonjour tout le monde!");
    System.out.println ("fib(20) = " + fib(20));
  }
}
```

les résultats successifs produits par appel à la fonction *Lexeme.suivant* seront :

```
L_Id(class) L_Id(PremierProg) L_AccG L_Id(public) L_Id(static)
L_Id(void) L_Id(main) L_ParG L_Id(String) L_CroG L_CroD L_Id(args)
L_ParD L_AccG L_Id(System) L_Point L_Id(out) L_Point L_Id(println)
L_ParG L_Chaine(Bonjour tout le monde!) L_ParD L_PointVirgule
L_Id(System) L_Point L_Id(out) L_Point L_Id(println) L_ParG
L_Chaine(fib(20) = ) L_Plus L_Id(fib) L_ParG L_Nombre(20) L_ParD
L_ParD L_PointVirgule L_AccD L_AccD L_EOF
```

Exercice 14 Modifier l'analyseur lexical pour sauter les commentaires comme en Java. (Il va y avoir un problème pour la fin de ligne).

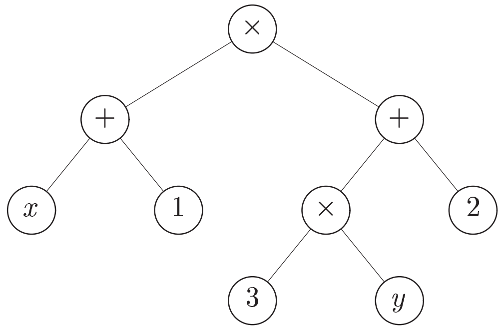


FIG. 2.3 – Arbre de syntaxe abstraite

Exercice 15 En se servant d'une pile, simuler une calculette HP, utilisant la notation polonaise suffixe, pour évaluer des expressions arithmétiques.

Notre présentation simplifiée de l'analyse lexicale ne doit pas faire oublier qu'il existe des outils très sophistiqués (comme *lex* de M. Lesk) permettant d'obtenir automatiquement de très bons analyseurs lexicaux à partir d'une description de haut niveau avec des expressions régulières. Cela simplifie grandement l'écriture de compilateurs, car souvent l'analyse lexicale est délicate à réaliser. En outre, ces outils sont très efficaces pour générer rapidement des automates finis de taille quasi minimale. (La notion d'expression régulière est reliée à la notion d'automate fini que nous verrons plus tard).

2.6 Arbres de syntaxe abstraite

Un compilateur produit un arbre de syntaxe abstraite à partir du texte du programme. En effet, un compilateur teste si le programme est syntaxiquement correct (grâce à la phase d'analyse syntaxique que nous verrons plus loin), mais il en produit aussi une représentation structurée sous forme arborescente, un *arbre de syntaxe abstraite*. Prenons le simple exemple des expressions arithmétiques construites à partir des nombres et de noms de variables avec les opérateurs $+$, $-$, $\times$ et $/$. Un arbre de syntaxe abstraite pour l'expression $(x+1) \times (3 \times y + 2)$ est représenté sur la figure 2.3.

La représentation de ces arbres de syntaxe abstraite, ou ASA, ou termes va se faire grâce à une classe *Terme* qui définit ce qu'on appelle un type disjonctif dans d'autres langages de programmation (ML) : l'ensemble des termes est l'union d'arbres binaires dont les noeuds sont des opérateurs pris parmi $+$, $-$, $\times$ et $/$, et les feuilles sont des variables ou des constantes entières. Ce qui nous donne :

```
class Terme {
  final static int ADD = 0, SUB = 1, MUL = 2, DIV = 3, MINUS = 4,
                 VAR = 5, CON = 6;
  int nature;
  int valeur;
  String nom;
  Terme a1, a2;

  Terme (int t, Terme a) {nature = t; a1 = a; }
  Terme (int t, Terme a, Terme b) {nature = t; a1 = a; a2 = b; }
  Terme (String s) {nature = VAR; nom = s; }
  Terme (int v) {nature = CON; valeur = v; }
}
```

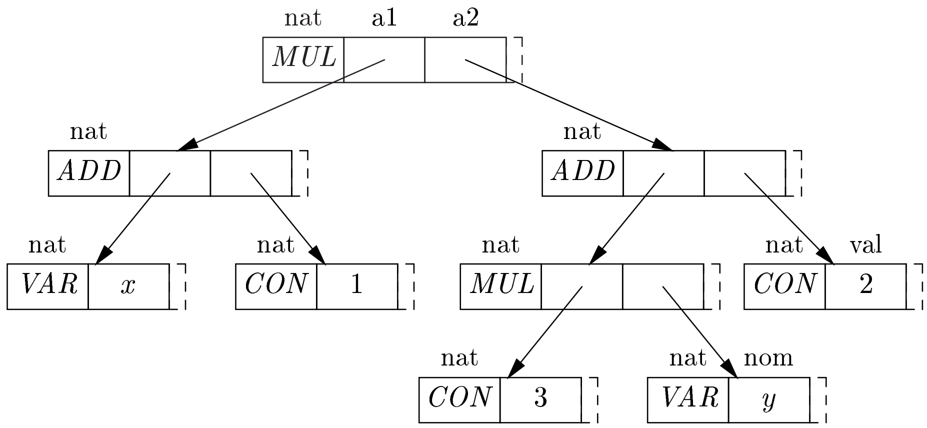


FIG. 2.4 – Représentation d'un arbre de syntaxe abstraite

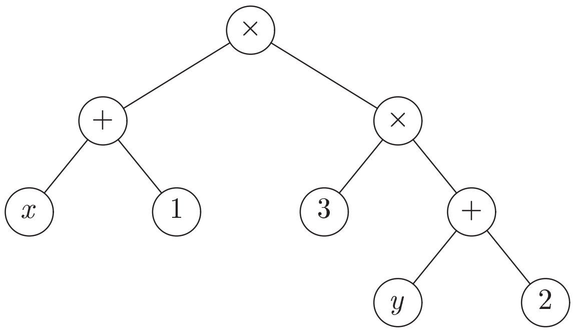


FIG. 2.5 – Un autre arbre de syntaxe abstraite

Comme pour la représentation précédente des lexèmes, les champs de données utilisés correspondent à tous les cas possibles. Le champ *nature* indique si le noeud correspond à une addition, soustraction, multiplication, un moins unaire, ou s'il s'agit d'une feuille pour les variables et les constantes ; le champ *nom* donne l'identificateur pour les feuilles qui sont des variables ; le champ *valeur* donne la constante entière pour les feuilles qui sont des constantes. Dans le cas de l'exemple de la figure 2.3, l'arbre de syntaxe abstraite est implanté comme indiqué sur la figure 2.4. On peut le construire par les instructions suivantes :

```
Terme t = new Terme (MUL,
            new Terme (ADD, new Terme ("x"), new Terme (1)),
            new Terme (ADD,
               new Terme (MUL, new Terme (3), new Terme ("y")),
               new Terme (2)));
```

Les arbres de syntaxe abstraite (ASA) constituent une des structures les plus abstraites associées à un programme. Dans l'exemple précédent, les textes de programme $(x+1)\times(3\times y+2)$, ou $(x+1)\times((3\times y)+2)$ ou $((x+1)\times(3\times y+(2)))$ correspondent au même ASA. Les détails de la « syntaxe concrète », avec leurs excès de parenthèses, ont été oubliés. Mais cette expression est, bien sûr, différente de l'expression de la figure 2.5 dont l'ASA est différent. Il correspond à l'expression $(x + 1) \times (3 \times (y + 2))$.

2.7 Grammaires

Pour construire des ensembles de mots, on utilise la notion de *grammaire*. Une grammaire $\mathcal{G}$ comporte deux alphabets A et V_N, un *axiome* S qui est une lettre appartenant à V_N et un ensemble $\mathcal{R}$ de *règles*.

- L'alphabet A est dit alphabet *terminal*, tous les mots construits par la grammaire sont constitués de lettres de A.
- L'alphabet V_N est dit alphabet *auxiliaire* ou *non-terminal*, ses lettres servent de variables intermédiaires servant à engendrer des mots. Une lettre S de V_N, appelée axiome, joue un rôle particulier.
- Les règles sont toutes de la forme $S \to u$ où S est une lettre de V_N et u un mot comportant des lettres dans $A \cup V_N$.

Exemple 2.9 $A = \{a, b\}$, $V_N = \{S, T, U\}$, l'axiome est S. Les règles sont données par :

$$\begin{array}{lll} S \to aTbS & T \to aTbT & U \to bUaU \\ S \to bUaS & T \to \epsilon & U \to \epsilon \\ S \to \epsilon & & \end{array}$$

Pour engendrer des mots à l'aide d'une grammaire, on applique le procédé suivant :

> On part de l'axiome S et on choisit une règle de la forme $S \to u$. Si u ne contient aucune lettre auxiliaire, on a terminé. Sinon, on écrit $u = u_1 T u_2$. On choisit une règle de la forme $T \to v$. On remplace u par $u' = u_1 v u_2$. On répète l'opération sur u' et ainsi de suite jusqu'à obtenir un mot qui ne contient que des lettres de A.

Dans la mesure où il y a plusieurs choix possibles à chaque étape, on voit que le nombre de mots engendrés par une grammaire est souvent infini. Mais certaines grammaires peuvent n'engendrer aucun mot. C'est le cas par exemple des grammaires dans lesquelles tous les membres droits des règles contiennent un lettre de V_N. On peut formaliser le procédé qui engendre les mots d'une grammaire de façon un peu plus précise en définissant la notion de *dérivation*. Etant donnés deux mots u et v contenant des lettres de $A \cup V_N$, on dit que *u dérive directement de v* pour la grammaire $\mathcal{G}$, et on note $v \to u$, s'il existe deux mots w_1 et w_2 et une règle de grammaire $S \to w$ de $\mathcal{G}$ tels que $v = w_1 S w_2$ et $u = w_1 w w_2$. On dit aussi que v se dérive directement en u. On dit que *u dérive de v*, ou que v se dérive en u, si u s'obtient à partir de v par une suite finie de dérivations directes. On note alors : $v \xrightarrow{*} u$, ce qui signifie l'existence de w_0, w_1, ... w_n $(n \geq 0)$ tels que $w_0 = v$, $w_n = u$ et, pour tout i $(0 \leq i < n)$, on a $w_i \to w_{i+1}$.

Un mot est engendré par une grammaire $\mathcal{G}$, s'il dérive de l'axiome et ne contient que des lettres de A, l'ensemble de tous les mots engendrés par la grammaire $\mathcal{G}$, est le *langage* engendré par $\mathcal{G}$; il est noté $\mathcal{L}(\mathcal{G})$.

Reprenons la grammaire $\mathcal{G}$ de l'exemple 2.9 et effectuons quelques dérivations en partant de S. Choisissons $S \to aTbS$, puis appliquons la règle $T \to \epsilon$. On obtient :

$$S \to aTbS \to abS$$

On choisit alors d'appliquer $S \to bUaS$. Puis, en poursuivant, on construit la suite

$$S \to aTbS \to abS \to abbUaS \to abbbUaUaS \to abbbaUaS \to abbbaaS \to abbbaa$$

D'autres exemples de mots $\mathcal{L}(\mathcal{G})$ sont $bbaa$ et $abbaba$ obtenus à l'aide de calculs similaires :

$$S \to bUaS \to bbUaUaS \to bbaUaS \to bbaaS \to bbaa$$

$$S \to aTbS \to abS \to abbUaS \to abbaS \to abbabUaS \to abbabaS \to abbaba$$

Plus généralement, on peut montrer que, pour cet exemple, $\mathcal{L}(\mathcal{G})$ est constitué de tous les mots qui contiennent autant de lettres a que de lettres b.

2.8 Exemples de Grammaires

2.8.1 Les systèmes de parenthèses

Le langage des systèmes de parenthèses joue un rôle important tant du point de vue de la théorie des langages que de la programmation. Dans les langages à structure de blocs, les `begin end` ou les `{ }` se comportent comme des parenthèses ouvrantes et fermantes. Dans des langages comme Lisp, le décompte correct des parenthèses fait partie de l'habileté du programmeur. Dans ce qui suit, pour simplifier l'écriture, on note a une parenthèse ouvrante et b une parenthèse fermante. Un mot de $\{a, b\}^*$ est un système de parenthèses s'il contient autant de a que de b et si tous ses facteurs gauches contiennent un nombre de a supérieur ou égal au nombre de b. Une autre définition possible est récursive, un système de parenthèses f est ou bien le mot vide ($f = \epsilon$) ou bien formé par deux systèmes de parenthèses f_1 et f_2 encadrés par a et b ($f = af_1bf_2$).

Cette nouvelle définition se traduit sous la forme de la grammaire suivante :

$A = \{a, b\}$, $V_N = \{S\}$, l'axiome est S, les règles sont données par :

$$S \to aSbS \qquad\qquad S \to \epsilon$$

On notera la simplicité de cette grammaire, la définition récursive rappelle celle des arbres binaires, un tel arbre est construit à partir de deux autres comme un système de parenthèses f l'est à partir de f_1 et f_2. La grammaire précédente a la particularité, qui est parfois un inconvénient, de contenir une règle dont le membre droit est le mot vide. On peut alors utiliser une autre grammaire déduite de la première qui engendre l'ensemble des systèmes de parenthèses non réduits au mot vide, dont les règles sont :

$$S \to aSbS \qquad S \to aSb \qquad S \to abS \qquad S \to ab$$

Cette transformation peut se généraliser et on peut ainsi pour toute grammaire G trouver une grammaire qui engendre le même langage, au mot vide près, et qui ne contient pas de règle de la forme $S \to \epsilon$.

2.8.2 Les expressions arithmétiques préfixées

Ce sont grosso modo les expressions que l'on trouve dans le langage Lisp. Dans une expression arithmétique préfixée, les opérateurs figurent avant leurs opérandes. Le nombre de leurs opérandes est délimité par un système de parenthèses. Pour simplifier, les opérandes atomiques (constantes entières ou variables) sont représentées par la lettre a. Leur définition récursive se traduit immédiatement par la grammaire suivante :

$A = \{+, \times, (,), a\}$, $V_N = \{S\}$, l'axiome est S, les règles sont données par :

$$S \to (+\ S\ S) \qquad S \to (\times\ S\ S) \qquad S \to a$$

Deux exemples d'expressions arithmétiques préfixées sont engendrés comme suit :

$$S \to (T\ S\ S) \xrightarrow{*} (\times\ a\ a)$$

$$\begin{aligned} S &\to (T\ S\ S) \\ &\xrightarrow{*} (T(T\ S\ S)(T\ S\ S)) \\ &\xrightarrow{*} (T(T\ S(T\ S\ S))(T(T\ S\ S)(T\ S\ S))) \\ &\xrightarrow{*} (\times(+\ a(\times\ a\ a))(+(\times\ a\ a)(\times\ a\ a))) \end{aligned}$$

Cette grammaire peut être généralisée pour traiter des expressions faisant intervenir d'autres opérateurs d'arité quelconque. Ainsi, pour ajouter les symboles $\sqrt{\ }$, $-$ et $/$, il suffit de considérer deux nouveaux éléments T_1 et T_2 dans V_N et prendre comme nouvelles règles :

$$\begin{array}{llllll} S \to (T_1 S) & S \to (T_2 S S) & S \to a & & & \\ T_1 \to \sqrt{\ } & T_1 \to - & T_2 \to + & T_2 \to \times & T_2 \to - & T_2 \to / \end{array}$$

2.8.3 Les expressions arithmétiques

Il s'agit maintenant de donner une grammaire pour les expressions arithmétiques infixes, c'est-à-dire notées comme en mathématiques. Pour simplifier, on ne considère que les deux opérateurs $\times$ et $+$. Les parenthèses permettent de regrouper les opérations, elles ne sont pas nécessaires pour délimiter un produit dans une somme, puisque que $\times$ est plus prioritaire que $+$. Dans un premier temps, nous supposons que les opérandes atomiques sont représentées par le symbole a. Les expressions arithmétiques sont donc engendrées par la grammaire suivante :

$A = \{+, \times, (,), a\}$, $V_N = \{E, P, F\}$, l'axiome est E, les règles sont données par :

$$\begin{array}{lll} E \to P & P \to F & F \to a \\ E \to P + E & P \to F \times P & F \to (E) \end{array}$$

Un mot engendré par cette grammaire est par exemple :

$$(a + a) \times (a \times a + a)$$

Il représente l'expression

$$(x + 1) \times (3 \times y + 2)$$

dans laquelle les opérandes atomiques sont remplacées par le symbole a.

On peut se convaincre que cette expression est engendrée par la grammaire en commençant la dérivation qui l'engendre à partir de l'axiome :

$$\begin{aligned} E &\to P \to F \times P \to (E) \times P \to (P + E) \times P \\ &\to (F + E) \times P \to (a + E) \times P \to (a + P) \times P \\ &\to (a + F) \times P \to (a + a) \times P \to (a + a) \times F \\ &\to (a + a) \times (E) \to \cdots \to (a + a) \times (a \times a + a) \end{aligned}$$

Les lettres de l'alphabet auxiliaire ont été choisies pour rappeler la *signification sémantique* des mots qu'elles engendrent. Ainsi E, P et F représentent respectivement les expressions, produits et facteurs. Dans cette terminologie, on constate que toute expression est somme de termes et que tout produit est produit de facteurs. Chaque facteur est ou bien réduit à la variable a ou bien formé d'une expression entourée de parenthèses. Ceci traduit les dérivations suivantes de la grammaire.

$$E \to P + E \to P + P + E \ldots \xrightarrow{*} P + P + P \ldots + P$$

$$P \to F \times P \to F \times F \times P \ldots \xrightarrow{*} F \times F \times F \ldots \times F$$

La convention usuelle de priorité de l'opération $\times$ sur l'opération $+$ explique que l'on commence par engendrer des sommes de termes avant de décomposer les termes en produits de facteurs, en règle générale pour des opérateurs de priorités quelconques on commence par engendrer les symboles d'opérations ayant la plus faible priorité pour terminer par ceux correspondant aux plus fortes.

On peut généraliser la grammaire pour faire intervenir beaucoup plus d'opérateurs. Il suffit d'introduire de nouvelles règles comme par exemple

$$E \to E - P \qquad\qquad P \to P \;/\; F$$

si l'on souhaite introduire des soustractions et des divisions. Comme ces deux opérateurs ont la même priorité que l'addition et la multiplication respectivement, il n'a pas été nécessaire d'introduire de nouveaux éléments dans V_N. Il faudrait faire intervenir de nouvelles variables auxiliaires si l'on introduit de nouvelles priorités.

Nous avons abusivement considéré le symbole a comme représentant unique des facteurs les plus simples. En fait, les expressions arithmétiques sont aussi formées avec des constantes numériques et des noms de variables. En fait, l'alphabet terminal est l'ensemble des lexèmes définis par un analyseur lexical. Dans notre cas, une syntaxe plus réaliste sera définie comme suit :

$A = \{+, \times, (,), id, nombre\}$, $V_N = \{E, P, F\}$, l'axiome est E, les règles sont données par :

$$\begin{array}{lll} E \to P & P \to F & F \to id \\ E \to P + E & P \to F \times P & F \to nombre \\ & & F \to (E) \end{array}$$

2.8.4 Notations abrégées

Plusieurs abréviations existent pour raccourcir la longueur d'une grammaire. Par exemple, on peut factoriser les parties gauche en utilisant la barre verticale ($|$) pour séparer les parties droites possibles. Ainsi la grammaire précédente s'écrirait :

$$\begin{array}{ll} E \to P \mid P + E & P \to F \mid F \times P \\ F \to id \mid nombre \mid (E) & \end{array}$$

Souvent aussi le symbole (:=) tend à remplacer la flèche ($\to$), donnant ainsi pour l'exemple précédent :

$$\begin{array}{ll} E := P \mid P + E & P := F \mid F \times P \\ F := id \mid nombre \mid (E) & \end{array}$$

Mais la forme la plus usuelle de notation abrégée est la BNF, sous laquelle on représente les grammaires des langages de programmation La BNF (*Backus Naur Form*) tient son nom de John Backus, le concepteur de FORTRAN, et de Peter Naur qui l'ont utilisé pour donner une définition formelle du langage Algol 60.

Dans la convention d'écriture adoptée pour la BNF, les éléments de V_N sont des suites de lettres et symboles comme *MultiplicativeExpression*, *UnaryExpression*. Les règles ayant le même élément dans leur partie gauche sont regroupées et cet élément n'est pas répété pour chacune d'entre elles. Le symbole $\rightarrow$ est remplacé par *:* suivi d'un passage à la ligne. Quelques conventions particulières permettent de raccourcir l'écriture : *one of* permet d'écrire plusieurs règles sur la même ligne ; les éléments de l'alphabet terminal A sont les mots-clé, comme `class`, `if`, `then`, `else`, `for`, ..., et les opérateurs ou séparateurs comme `+ * / - ; , ( ) [ ] = == < > .`

Dans les grammaires données en annexe, on compte dans la grammaire de Java 131 lettres pour l'alphabet auxiliaire et 251 règles. Il est hors de question de traiter ici ce trop long exemple. Nous nous limitons aux exemples donnés plus haut dans lesquels figurent déjà toutes les difficultés que l'on peut trouver par ailleurs.

Notons toutefois que l'on trouve la grammaire des expressions arithmétiques sous forme BNF dans l'exemple de la grammaire de Java donnée en annexe. On trouve en effet à l'intérieur de cette grammaire :

```
AdditiveExpression:
        MultiplicativeExpression
        AdditiveExpression + MultiplicativeExpression
        AdditiveExpression - MultiplicativeExpression

MultiplicativeExpression:
        UnaryExpression
        MultiplicativeExpression * UnaryExpression
        MultiplicativeExpression / UnaryExpression
        MultiplicativeExpression % UnaryExpression

UnaryExpression:
        PreIncrementExpression
        PreDecrementExpression
        + UnaryExpression
        - UnaryExpression
        UnaryExpressionNotPlusMinus

UnaryExpressionNotPlusMinus:
        PostfixExpression
        ~ UnaryExpression
        ! UnaryExpression
        CastExpression

PostfixExpression:
        Primary
        Name
        PostIncrementExpression
        PostDecrementExpression

Primary:
        PrimaryNoNewArray
        ArrayCreationExpression

PrimaryNoNewArray:
        Literal
        this
        ( Expression )
        ClassInstanceCreationExpression
```

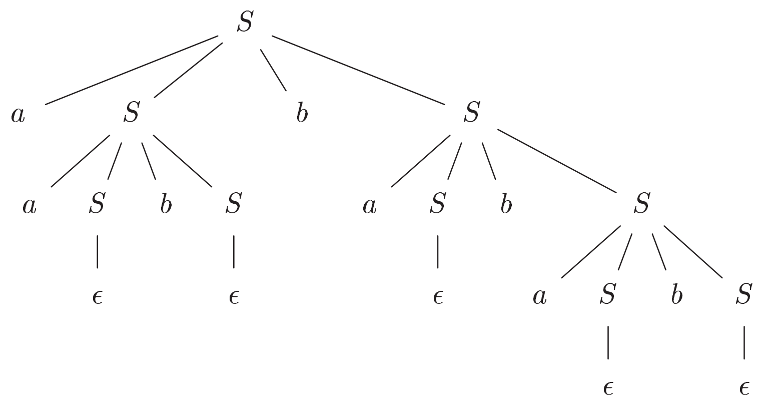
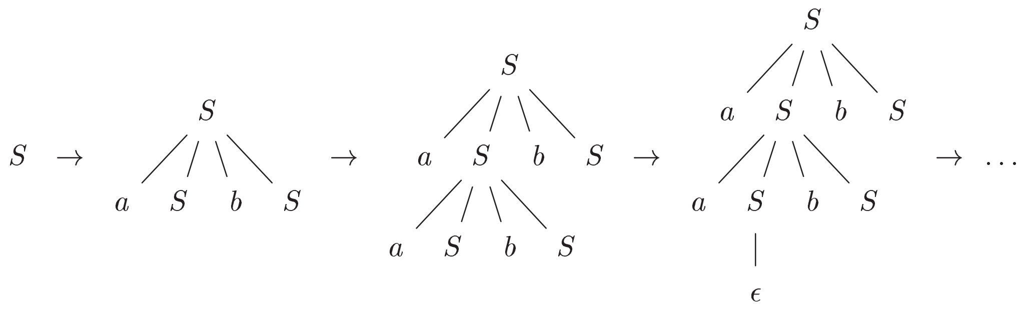


FIG. 2.6 – Arbre de dérivation de *aabbabab*

FieldAccess
MethodInvocation
ArrayAccess

2.9 Analyse syntaxique

Le but de l'analyse syntaxique est de déterminer si un mot appartient ou non au langage engendré par une grammaire. Il s'agit donc, étant donné un mot f de construire la suite des dérivations qui a permis de l'engendrer. Si l'on pratique ceci à la main pour de petits exemples, on peut utiliser la technique classique dite « essais-erreurs » consistant à tenter de deviner à partir d'un mot la suite des dérivations qui ont permis de l'engendrer. Cette suite se présente plus clairement sous forme d'un arbre, dit *arbre de dérivation* ou *arbre syntaxique*, dont des exemples sont donnés figures 2.6 et 2.7. Il s'agit de ceux obtenus pour les mots *aabbabab* et $(x+1)\times(3\times y+2)$ engendrés respectivement par la grammaire des systèmes de parenthèses et par celle des expressions arithmétiques. L'arbre de dérivation de la figure 2.6 est construit progressivement comme suit :



L'étiquette S de la racine de l'arbre est l'axiome de la grammaire. Chaque noeud interne est étiqueté par une variable X non-terminale (appartenant à V_N) ; en outre, le mot u obtenu en lisant de gauche à droite les étiquettes de ses fils est tel que $X \to u$ est une règle de la grammaire. Enfin, le mot f dont on fait l'analyse est constitué des étiquettes des feuilles lues de gauche à droite.

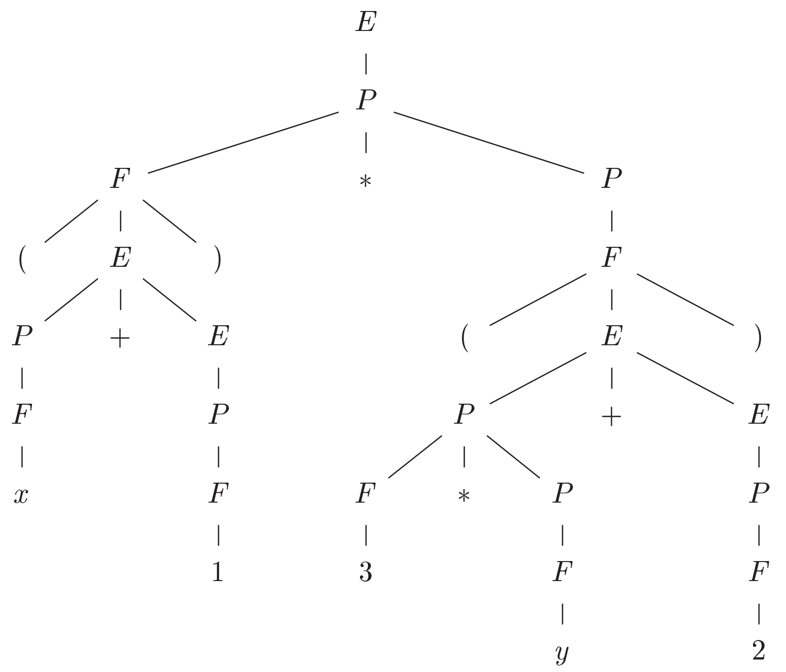


FIG. 2.7 – Arbre de dérivation d'une expression arithmétique

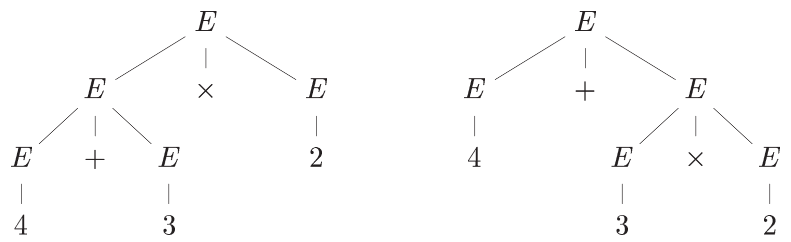


FIG. 2.8 – Grammaire ambiguë

Plusieurs dérivations peuvent correspondre à un même arbre syntaxique. Mais, en général, il n'existe qu'un seul arbre de dérivation pour un mot donné du langage engendré. L'existence de plusieurs arbres de dérivations pour un même mot signifie souvent qu'il existe plusieurs interprétations possibles pour celui-ci. On dit qu'alors la grammaire est ambiguë. Toutes les grammaires données plus haut sont non-ambiguës. En revanche, la grammaire suivante pour les expressions arithmétiques est ambiguë.

$A = \{+, \times, (,), id, nombre\}$, $V_N = \{E\}$, l'axiome est E, les règles sont données par :

$$\begin{array}{lll} E \to E + E & E \to id & E \to (E) \\ E \to E \times E & E \to nombre & \end{array}$$

En effet, le mot $4 + 3 \times 2$ a deux arbres de dérivation distincts comme indiqué sur la figure 2.8. Ils correspondent aux expressions $(4 + 3) \times 2 = 14$ et $4 + (3 \times 2) = 10$.

L'arbre de dérivation est parfois appelé arbre de syntaxe concrète pour le distinguer de *l'arbre de syntaxe abstraite*. L'arbre de syntaxe abstraite est bien plus compact que le précédent. Il s'obtient par des transformations simples à partir de l'arbre de dérivation. C'est bien ce que fait un analyseur syntaxique.

2.10 Analyse descendante récursive

Deux principales techniques sont utilisées pour effectuer l'analyse syntaxique. Etant donnés une grammaire G et un mot f, il s'agit de construire la suite des dérivations de G conduisant de l'axiome au mot f,

$$S \to u_1 \to u_2 \dots u_{n-1} \to u_n = f$$

La première technique consiste à démarrer de l'axiome S et à tenter de retrouver u_1, puis u_2 jusqu'à obtenir $u_n = f$, c'est *l'analyse descendante*. La seconde, *l'analyse ascendante* procède en sens inverse, il s'agit de commencer par deviner u_{n-1} à partir de f puis de remonter à u_{n-2} et successivement jusqu'à l'axiome S. Nous décrivons ici sur des exemples les techniques d'analyse descendante, l'analyse ascendante sera traitée dans un paragraphe suivant.

La première méthode que nous considérons s'applique à des grammaires très particulières, mais que l'on rencontre très souvent. Dans ces cas, l'algorithme d'analyse syntaxique devient une traduction fidèle de l'écriture de la grammaire. On utilise pour cela autant de fonctions qu'il y a d'éléments dans V_N, chacune d'entre elles est destinée à reconnaître un mot dérivant de l'élément correspondant de V_N. Prenons le cas des expressions arithmétiques de la section 2.8.3, l'analyseur récursif descendant s'écrit :

```
static Lexeme lc; // lexème courant
static void avancer() {lc = Lexeme.suivant(); }

static Terme expression() {
  Terme t = produit(); switch (lc.nature) {
  case Lexeme.L_Plus: avancer(); return new Terme (ADD, t, expression());
  case Lexeme.L_Moins: avancer(); return new Terme (MINUS, t, expression());
  default: return t;
 }
}

static Terme produit() {
  Terme t = facteur(); switch (lc.nature) {
  case Lexeme.L_Mul: avancer(); return new Terme (MUL, t, produit());
  case Lexeme.L_Div: avancer(); return new Terme (DIV, t, produit());
  default: return t;
 }
}

static Terme facteur() {
  Terme t; switch (lc.nature) {
  case Lexeme.L_ParG: avancer(); t = expression();
    if (lc.nature != Lexeme.L_ParD) throw new Error ("Il manque ')'");
    break;
  case Lexeme.L_Nombre: t = new Terme (lc.val); break;
  case Lexeme.L_Id: t = new Terme (lc.nom); break;
  default: throw new Error ("Erreur de syntaxe");
  }
  avancer();
  return t;
}
```

Ce programme réutilise les constantes et les fonctions définies pour l'analyseur lexical présenté plus haut. Les fonctions *expression*, *produit* et *facteur* correspondent aux variables auxiliaires E, P et F. Le corps de chaque fonction consiste à considérer les différentes parties droites possibles pour les règles de grammaires associées à ces variables. Ainsi, pour E, on choisit entre les trois cas $P + E$, $P - E$ et P selon le lexème courant rencontré après avoir lu un produit. Si c'est l'opérateur $+$ ou $-$, on est dans le cas où la partie droite est $P + E$, ou $P - E$. On fait de même pour choisir entre $F \times P$, $F\,/\,P$ et F dans le cas d'un produit en décidant sur les lexèmes $\times$ ou `/`. Pour un facteur, la décision entre `(`E`)`, *id* ou *nombre* se fait en considérant le lexème courant qui alors ne peut être que `(`, *L_Id* ou *L_Nombre*.

Cette technique fonctionne bien ici car les membres droits des règles de grammaire ont une forme particulière. Pour chaque élément X de V_N, l'ensemble des membres droits $\{u_1, u_2 \ldots u_p\}$ de règles, dont le membre gauche est X, satisfait les conditions suivantes : les premières lettres des u_i qui sont dans A, sont toutes distinctes et les u_i qui commencent par une lettre de V_N sont factorisables à gauche. Beaucoup de grammaires de langages de programmation satisfont ces conditions : Pascal, Java.

Remarquons que notre grammaire des expressions arithmétiques ne contient pas de récursivité à gauche, c'est-à-dire que V_N ne contient aucun X pouvant se dériver vers un mot Xu. Mais, si nous avions plutôt pris les règles de grammaire suivantes :

$$\begin{array}{lll} E \rightarrow P & P \rightarrow F & F \rightarrow id \\ E \rightarrow E + P & P \rightarrow P \times F & F \rightarrow nombre \\ & & F \rightarrow (E\,) \end{array}$$

la grammaire possède maintenant des récursivités à gauche, puisqu'on a $E \xrightarrow{*} Eu$ et $P \xrightarrow{*} Pv$. Alors l'analyseur récursif descendant associé à cette nouvelle grammaire se mettrait à boucler en commençant par chercher une expression dans une expression, ou un produit dans un produit. La méthode d'analyse récursive descendante ne marche donc pas si la grammaire est récursive à gauche.

La valeur retournée par notre analyseur syntaxique est un arbre de syntaxe abstraite. C'est bien ce que l'on cherche à construire. Comme annoncé, l'analyseur syntaxique vérifie la conformité syntaxique de l'expression sur le flot d'entrée et construit l'objet de la classe *Terme* correspondant, dans le cas où l'entrée est un texte de programme syntaxiquement correct.

La grammaire ne peut donc être récursive à gauche ; on reprend donc la grammaire de la section 2.8.3. Cette grammaire aboutit à la construction d'arbres de syntaxe abstraite qui privilégient l'association à droite pour les opérateurs de même précédence. Ainsi $P + P + P$ est interprété comme $P + (P + P)$; de même $F \times F \times F$ correspond à $F \times (F \times F)$. Malheureusement, ce choix n'est pas le bon pour les opérateurs $-$ ou `/`, puisque $P - P - P$ correspond d'habitude à $(P - P) - P$ et non à $P - (P - P)$ comme généré par notre grammaire. Pour y remédier, tout en respectant l'absence de récursivité à gauche, on change l'analyseur pour générer les arbres de syntaxe abstraites avec un accumuleur. On modifie donc les fonctions *expression* et *produit* comme suit :

```
static Terme expression() {
  Terme t = produit();
  while (lc.nature == Lexeme.L_Plus) || (lc.nature == Lexeme.L_Moins) {
    avancer();
    switch (lc.nature) {
```

```
        case Lexeme.L_Plus: t = new Terme (ADD, t, produit()); break;
        case Lexeme.L_Moins: t = new Terme (MINUS, t, produit()); break;
      }
     return t;
    }
  }

  static Terme produit() {
    Terme t = facteur();
    while (lc.nature == Lexeme.L_Mul) || (lc.nature == Lexeme.L_Div) {
      avancer();
      switch (lc.nature) {
        case Lexeme.L_Mul: t = new Terme (MUL, t, facteur()); break;
        case Lexeme.L_Div: t = new Terme (DIV, t, facteur()); break;
      }
     return t;
    }
  }
```

Notre analyseur syntaxique récursif descendant s'arrête dès qu'il a lu une expression arithmétique. Pourtant il se peut que le résultat ne correspond pas à tout le flot d'entrée. Par exemple `3␣2` sera considéré comme correct et reconnu comme la constante 3 ; de même $(x+1)(3 \times y+2)$ est reconnu comme l'expression $x+1$. Il faut donc vérifier si, après la lecture de l'expression, le lexème courant arithmétique est bien la fin de fichier *L_EOF*. La grammaire correspondante est alors :

$A = \{+, \times, (,), id, nombre, eof\}$, $V_N = \{S, E, P, F\}$, l'axiome est S, les règles sont données par :

$$\begin{array}{lll} E \to P & P \to F & F \to id \\ E \to P + E & P \to F \times P & F \to nombre \\ S \to E\ eof & & F \to (E) \end{array}$$

2.11 Autres méthodes d'analyse syntaxique

2.11.1 Méthode récursive descendante avec retours en arrière

Une technique plus générale d'analyse syntaxique consiste à procéder comme suit. On construit itérativement des mots u dont on espère qu'ils vont se dériver en f. Au départ on a $u = S$ (l'axiome de la grammaire). A chaque étape de l'itération, on cherche la première lettre de u qui n'est pas égale à son homologue dans f. On a ainsi

$$u = gyv \quad f = gxh \quad x \neq y$$

Si $y \in A$ (alphabet terminal), on ne peut dériver f de u, et il faut faire repartir l'analyse du mot qui a donné u. Sinon $y \in V_N$ et on recherche toutes les règles dont y est le membre gauche.

$$y \to u_1 \quad y \to u_2 \quad \cdots \quad y \to u_k$$

On applique à u successivement chacune de ces règles, on obtient ainsi des mots $v_1, v_2, \ldots v_k$. On poursuit l'analyse, chacun des mots $v_1, v_2, \ldots v_k$ jouant le rôle de u. L'analyse est terminée lorsque $u = f$. La technique est celle de l'exploration arborescente qui sera développée au chapitre 4. On peut la représenter par la fonction suivante donnée sous une forme informelle.

```
static boolean analyse (String u, String f) {
  if (f.equals(u))
    return true;
  else {
    Mettre f et u sous la forme f = gxh, u = gyv où x ≠ y
    if ( y ∉ A )
      Pour toute règle y → w faire
        if (analyse (g + w + v, f))
          return true;
    return false;
  }
}
```

Remarquons que cette fonction ne marche toujours pas lorsque la grammaire est récursive à gauche, comme dans la grammaire des expressions arithmétiques de la page 62. En outre, cette technique d'analyse engendre une détection d'erreur imprécise, car les erreurs de syntaxe ne sont trouvées qu'après avoir fait tous les choix de règle. Elle est aussi coûteuse en temps car on effectue tous les essais successifs des règles et on peut parfois se rendre compte, après avoir pratiquement terminé l'analyse, que la première règle appliquée n'était pas la bonne. Il faut alors tout recommencer avec une autre règle et éventuellement répéter plusieurs fois ce mécanisme. La complexité de l'algorithme est ainsi une fonction exponentielle de la longueur du mot f à analyser.

Si on suppose qu'aucune règle ne contient un membre droit égal au mot vide, on peut diminuer la quantité de calculs effectués en débutant la procédure d'analyse par un test vérifiant si la longueur de u est supérieure à celle de f. Dans ce cas, la procédure d'analyse doit avoir pour résultat *false*. Dans ces conditions, la fonction *analyse* donne un résultat même dans le cas de grammaires récursives à gauche.

2.11.2 Analyse LL

Une technique pour éviter les retours en arrière de l'analyse descendante récursive consiste à deviner la règle à appliquer en examinant les premières lettres du mot f à analyser. Plus généralement, lorsque l'analyse a déjà donné le mot u et que l'on cherche à obtenir f, on écrit comme ci-dessus

$$u = gSv \quad f = gh$$

et les premières lettres de h doivent permettre de retrouver la règle qu'il faut appliquer à S. Cette technique n'est pas systématiquement possible pour toutes les grammaires, mais c'est le cas sur un grand nombre de grammaires utiles, comme par exemple celle des expressions préfixées ou une grammaire modifiée (par factorisation à gauche) des expressions infixes. On dit alors que la grammaire est LL.

Exemple 2.10 *Expressions préfixées.* Nous considérons la grammaire de ces expressions : $A = \{+, *, (,), a\}$, $V_N = \{S\}$, l'axiome est S, les règles sont données par :

$$S \to (+\ S\ S) \qquad S \to (*\ S\ S) \qquad S \to a$$

Pour un mot f de A^*, il est immédiat de déterminer u_1 tel que

$$S \to u_1 \xrightarrow{*} f$$

En effet, si f est de longueur 1, ou bien $f = a$ et le résultat de l'analyse syntaxique se limite à $S \rightarrow a$, ou bien f n'appartient pas au langage engendré par la grammaire.

Si f est de longueur supérieure à 1, il suffit de connaître les deux premières lettres de f pour pouvoir retrouver u_1. Si ces deux premières lettres sont (+, c'est la règle $S \rightarrow (+S\ S)$ qui a été appliquée. Si ces deux lettres sont $(*$ alors c'est la règle $S \rightarrow (*S\ S)$. Tout autre début de règle conduit à un message d'erreur.

Ce qui vient d'être dit pour retrouver u_1 en utilisant les deux premières lettres de f se généralise sans difficulté à la détermination du $(i+1)^{\text{ème}}$ mot u_{i+1} de la dérivation à partir de u_i. On décompose d'abord u_i et f en :

$$u_i = g_i S v_i \qquad f = g_i f_i$$

et on procède en fonction des deux premières lettres de f_i.

- Si f_i commence par a, alors $u_{i+1} = g_i a v_i$
- Si f_i commence par (+, alors $u_{i+1} = g_i(+SS)v_i$
- Si f_i commence par $(*$, alors $u_{i+1} = g_i(*SS)v_i$
- Un autre début pour f_i signifie que f n'est pas une expression préfixée correcte, il y a une erreur de syntaxe.

On en déduit une fonction de construction d'un arbre de syntaxe abstraite pour les expressions préfixées en reprenant les définitions antérieures des classes pour les lexèmes et les termes :

```
static Terme expressionPrefixe() {
  Terme t; switch (lc.nature) {
    case Lexeme.L_Id: t = new Terme (lc.nom); break;
    case Lexeme.L_ParG: avancer();
      switch (lc.nature) {
        case Lexeme.L_Plus: avancer(); t = expressionPrefixe();
          t = new Terme (ADD, t, expressionPrefixe()); break;
        case Lexeme.L_Mul: avancer(); t = expressionPrefixe();
          t = new Terme (MUL, t, expressionPrefixe()); break;
        default: throw new Error ("Erreur de syntaxe");
      }
      if (lc.nature != Lexeme.L_ParD) throw new Error ("Il manque ')'");
    default: throw new Error ("Erreur de syntaxe");
  }
  avancer();
  return t;
}
```

Cette fonction ne fait donc aucun retour en arrière. Cette technique d'analyse syntaxique donné s'étend à toute grammaire LL. Ici on détermine la règle à appliquer en regardant deux lexèmes à l'avance dans le flot de lexèmes d'entrée. La grammaire est dite $LL(k)$ si on peut détermier la règle à appliquer dans l'analyse descendante en regardant k lexèmes à l'avance. Alors on peut énoncer le résultat suivant :

Théorème 2.2 *Si G est une grammaire $LL(k)$, il existe un algorithme en $O(n)$ qui effectue l'analyse syntaxique descendante d'un mot f de longueur n.*

En fait, cet algorithme sert principalement pour $k \leq 2$. On peut le contruire en prenant la grammaire en paramètre. C'est ce que font des méta-analyseurs syntaxiques comme *javacc*. En outre, une définition formelle des analyseurs LL peut être donnée en considérant la théorie des automates à pile.

Pour finir, reconsidérons le cas des expressions arithmétiques infixées. La grammaire suivante n'est pas LL, car il est difficile de distinguer les applications des règles $E \rightarrow P$ et $E \rightarrow P + E$ en regardant k lettres à l'avance.

$$\begin{array}{lll} E \rightarrow P & P \rightarrow F & F \rightarrow a \\ E \rightarrow P + E & P \rightarrow F \times P & F \rightarrow (E) \end{array}$$

Cependant, par factorisation à gauche, on construit la grammaire $LL(1)$ équivalente suivante, sur laquelle on retrouve la fonction d'analyse syntaxique considérée en 2.10.

$$\begin{array}{lll} E \rightarrow PE' & P \rightarrow FP' & \\ E' \rightarrow \epsilon & P' \rightarrow \epsilon & F \rightarrow a \\ E' \rightarrow +PE' & P' \rightarrow \times FP' & F \rightarrow (E) \end{array}$$

2.11.3 Analyse ascendante

Les algorithmes d'analyse ascendante sont légèrement plus compliqués que ceux de l'analyse descendante. Ils s'appliquent à un plus grand nombre de grammaires. C'est pour cette raison qu'on les utilise souvent. Ils sont ainsi à la base du système *yacc* de S. Johnson qui sert à écrire des compilateurs sous le système *Unix*. Rappelons que l'analyse ascendante consiste à retrouver la dérivation

$$S_0 \rightarrow u_1 \rightarrow u_2 \ldots u_{n-1} \rightarrow u_n = f$$

en commençant par u_{n-1} puis u_{n-2} et ainsi de suite jusqu'à remonter à l'axiome S_0. On effectue ainsi ce que l'on appelle des *réductions* car il s'agit de remplacer un membre droit d'une règle par le membre gauche correspondant, celui-ci est en général plus court.

Un exemple de langage qui n'admet pas d'analyse syntaxique descendante simple, mais sur lequel on peut effectuer une analyse ascendante est le langage des systèmes de parenthèses. Rappelons sa grammaire :

$$S \rightarrow aSbS \qquad S \rightarrow aSb \qquad S \rightarrow abS \qquad S \rightarrow ab$$

On voit que les règles $S \rightarrow aSbS$ et $S \rightarrow aSb$ peuvent engendrer des mots ayant un facteur gauche commun arbitrairement long, ce qui interdit tout algorithme de type $LL(k)$. Cependant, nous allons donner un algorithme simple d'analyse ascendante d'un mot f.

Partons de f et commençons par tenter de retrouver la dernière dérivation, celle qui a donné $f = u_n$ à partir d'un mot u_{n-1}. Nécessairement u_{n-1} contenait un S qui a été remplacé par ab pour donner f. L'opération inverse consiste donc à remplacer un ab par S, mais ceci ne peut pas être effectué n'importe où dans le mot, ainsi si on a

$$f = ababab$$

il y a trois remplacements possibles donnant

$$Sabab \qquad abSab \qquad ababS$$

Les deux premiers ne permettent pas de poursuivre l'analyse. En revanche, à partir du troisième, on retrouve abS et finalement S. D'une manière générale, on remplace ab par S chaque fois qu'il est suivi de b ou qu'il est situé en fin de mot. Les autres règles de grammaires s'inversent aussi pour donner des règles d'analyse syntaxique. Ainsi :

– Réduire aSb en S s'il est suivi de b ou s'il est situé en fin de mot.
– Réduire ab en S s'il est suivi de b ou s'il est situé en fin de mot.
– Réduire abS en S quelle que soit sa position.
– Réduire $aSbS$ en S quelle que soit sa position.

On a un algorithme du même type pour l'analyse des expressions arithmétiques infixes engendrées par la grammaire :

$$\begin{array}{lll} E \to P & P \to F & F \to a \\ E \to E + P & P \to P \times F & F \to (E) \\ E \to E - P & & \end{array}$$

Pour effectuer une réduction, cet algorithme tient compte de la première lettre qui suit le facteur que l'on envisage de réduire (et de ce qui se trouve à gauche de ce facteur). On dit que la grammaire est $LR(1)$. La théorie complète de ces grammaires mériterait un plus long développement ; elle correspond à la notion de langages générés par des automates déterministes. Nous nous contentons de donner ici ce qu'on appelle *l'automate* $LR(1)$ qui effectue l'analyse syntaxique de la grammaire, récursive à gauche, des expressions infixes.

On lit le mot à analyser de gauche à droite et on effectue les réductions suivantes dès qu'elles sont possibles :

– Réduire a en F quelle que soit sa position.
– Réduire (E) en F quelle que soit sa position.
– Réduire F en T s'il n'est pas précédé de $*$.
– Réduire $T * F$ en T quelle que soit sa position.
– Réduire T en E s'il n'est pas précédé de $+$ et s'il n'est pas suivi de $*$.
– Réduire $E + T$ en E s'il n'est pas suivi de $*$.
– Réduire $E - T$ en E s'il n'est pas suivi de $*$.

On peut gérer le mot réduit à l'aide d'une pile. Les opérations de réduction consistent à supprimer des éléments dans celle-ci, les tests sur ce qui précède ou ce qui suit se font très simplement en consultant les premiers symboles de la pile. L'analyse se fait en une seule passe sur le mot à analyser, donc sans retours en arrière. Comme pour le cas $LL(k)$, on a le théorème suivant pour les grammaires $LR(k)$.

Théorème 2.3 *Si G est une grammaire $LR(k)$, il existe un algorithme en $O(n)$ qui effectue l'analyse syntaxique descendante d'un mot f de longueur n.*

2.12 Programmes sur les arbres de syntaxe abstraite

L'analyse syntaxique consiste à construire des arbres de syntaxe abstraite. Dans cette section, nous donnons quelques exemples d'utilisation de cette structure.

2.12.1 Evaluation

Pour évaluer la valeur d'un terme correspondant à une expression arithmétique, il faut se donner une valuation des variables, c'est-à-dire une fonction associant une valeur à chacune des variables contenues dans ce terme. Le graphe de cette fonction est appelé un environnement. Le programme pour évaluer une expression correspond au morphisme suivant sur les arbres de syntaxe abstraite :

```
static int evaluer (Terme t, Environnement e) {
  switch (t.nature) {
  case ADD: return evaluer (t.a1, e) + evaluer (t.a2, e);
  case SUB: return evaluer (t.a1, e) - evaluer (t.a2, e);
  case MUL: return evaluer (t.a1, e) * evaluer (t.a2, e);
  case DIV: return evaluer (t.a1, e) / evaluer (t.a2, e);
  case CONST: return t.val;
  case VAR: return Environnement.assoc (t.nom, e);
  default: throw new Error ("Erreur dans evaluation");
  }
}
```

où la classe *Environnement* définit la liste d'association suivante :

```
class Environnement {
  String nom;
  int val;
  Environnement suivant;

  static int assoc (String s, Environnement e) {
    if (e == null) throw new Error ("Variable non définie");
    if (e.nom.equals(s)) return e.val;
    else return assoc (s, e.suivant);
  }
}
```

Parfois, on dit aussi que la fonction *evaluer* est définie par *induction structurelle* sur la classe des termes.

2.12.2 Impression

Un autre programme classique sur les arbres de syntaxe abstraite consiste à les imprimer. C'est la fonction inverse de l'analyse syntaxique, puisqu'il s'agit de générer une chaîne de caractères, dont l'analyse syntaxique produirait le même arbre. Dans le cas des expressions arithmétiques, plusieurs programmes d'impression sont possibles. Le plus simple est de produire les expressions complètement parenthésées. Par exemple, avec l'ASA de la figure 2.3, on obtiendrait $((x+1)\times((3\times y)+2))$. L'idéal est plutôt de produire une impression avec le minimum de parenthèses, comme $(x+1)\times(3\times y+2)$. Notre programme sera le symétrique de l'analyseur syntaxique :

```
static void impExp (Terme t) {
  switch (t.nature) {
  case ADD: impProd(t.a1); System.out.print ("+"); impExp(t.a2); break;
  case SUB: impProd(t.a1); System.out.print ("-"); impExp(t.a2); break;
  default: impProd(t); break;
  }
}

static void impProd (Terme t) {
  switch (t.nature) {
  case MUL: impFact(t.a1); System.out.print ("*"); impProd(t.a2); break;
  case DIV: impFact(t.a1); System.out.print ("/"); impProd(t.a2); break;
  default: impFact(t); break;
  }
}

static void impFact (Terme t) {
  switch (t.nature) {
  case CON: System.out.print (t.valeur); break;
  case VAR: System.out.print (t.nom); break;
```

```
    default: System.out.print ("("); impExp(t); System.out.print (")"); break;
    }
  }
```

Exercice 16 Faire le programme d'impression avec la règle d'association à gauche pour les opérateurs de même précédence.

2.12.3 Génération de code

Il s'agit de générer du code machine pour calculer des expressions arithmétiques sur un processeur simplifié. Notre machine a des registres R_i dans lesquels s'effectuent les opérations arithmétiques. Les registres peuvent recevoir une constante n (entière) ou être chargés à partir d'une mémoire x. Le jeu complet des instructions de notre processeur est le suivant :

$$\begin{array}{lll} R_i \leftarrow n & R_i \leftarrow R_i + R_j & R_i \leftarrow R_i \times R_j \\ R_i \leftarrow x & R_i \leftarrow R_i - R_j & R_i \leftarrow R_i \;/\; R_j \end{array}$$

Ainsi pour évaluer l'expression $(x+1)\times(3\times(y+2))$ (dont l'arbre de syntaxe abstraite est sur la figure 2.4), on peut générer par exemple un des deux codes suivants :

$$\begin{array}{ll} R_0 \leftarrow x & R_0 \leftarrow x \\ R_1 \leftarrow 1 & R_1 \leftarrow 1 \\ R_0 \leftarrow R_0 + R_1 & R_0 \leftarrow R_0 + R_1 \\ R_1 \leftarrow 3 & R_1 \leftarrow y \\ R_2 \leftarrow y & R_2 \leftarrow 2 \\ R_3 \leftarrow 2 & R_1 \leftarrow R_1 + R_2 \\ R_2 \leftarrow R_2 + R_3 & R_2 \leftarrow 3 \\ R_1 \leftarrow R_1 \times R_2 & R_2 \leftarrow R_2 \times R_1 \\ R_0 \leftarrow R_0 \times R_1 & R_0 \leftarrow R_0 \times R_2 \end{array}$$

On constate que le premier code utilise 4 registres (R_0, R_1, R_2 et R_3), alors que le deuxième n'en utilise que 3 (R_0, R_1 et R_2). Si les registres constituent une ressource rare, il vaut mieux utiliser la deuxième solution. Pour connaître le nombre de registres minimum à utiliser, une vieille remarque due à Ershov consiste à dire que pour évaluer une constante ou une variable, il suffit d'un seul registre, et que, pour évaluer une expression de la forme $e \oplus e'$, il faut un registre supplémentaire pour stocker le résultat de e ou de e', si e et e' ont besoin d'un même nombre de registres, et que sinon il suffit d'évaluer d'abord l'expression qui demande le plus grand nombre de registres, et de stocker son résultat dans un registre non nécessaire pour le calcul de la deuxième expression. Ainsi, le nombre de registres minimal $reg(e)$ pour évaluer e est défini récursivement par :

$$\begin{array}{l} reg(x) = reg(n) = 1 \\ reg(e \oplus e') = \begin{cases} 1 + reg(e) & \text{si } reg(e) = reg(e') \\ \max(reg(e), reg(e')) & \text{sinon} \end{cases} \end{array}$$

D'où le programme suivant pour générer un code utilisant un nombre optimal de registres :

```
static int reg (Terme t) {
  switch (t.nature) {
  case ADD: case SUB: case MUL: case DIV:
    int n1 = reg(t.a1), n2 = reg(t.a2);
    if (n1 == n2) return 1 + n1;
    else return Math.max(n1,n2);
  case CONST: case VAR: return 1;
  default: throw new Error ("Erreur dans evaluation");
  }
}

static int genCode (Terme t, int r0) {
  switch (t.nature) {
  case ADD: case SUB: case MUL: case DIV:
    int n1 = reg(t.a1), n2 = reg(t.a2);
    if (n1 >= n2) {
      int r1 = genCode(t.a1, r0), r2 = genCode(t.a2, r1);
      System.out.println ("R" + r1 + " <- R" + r1 + " + R" + r2);
      return r1;
    } else {
      int r2 = genCode(t.a2, r0), r1 = genCode(t.a1, r2);
      System.out.println ("R" + r2 + " <- R" + r2 + " + R" + r1);
      return r2;
    }
  case CONST:
    System.out.println ("R" + r0 " <- " + t.val);
    return r0;
  case VAR:
    System.out.println ("R" + r0 " <- " + t.nom);
    return r0;
  default: throw new Error ("Erreur dans evaluation");
  }
}
```

A nouveau, la fonction *genCode* est définie par induction structurelle sur la classe des termes. On comprend bien sur cette exemple la nécessité de la phase d'analyse syntaxique, qui a permis de bien isoler la structure sur laquelle s'effectue la génération de code, à savoir la structure de syntaxe abstraite. Bien sûr, il existe des techniques beaucoup plus sophistiquées pour générer du code. C'est l'objet de tout le domaine de la compilation. Par exemple, on peut tenir compte du partage, et ne pas recalculer des expressions déjà calculées ; ainsi, dans $(x + 1) \times (3 \times (x + 1))$, ce n'est pas la peine de générer deux fois le code de $x + 1$. Un autre exemple est de tenir compte de la non banalisation des registres, certaines opérations ne pouvant se faire que dans des registres pairs. Un troisième exemple consiste à tenir compte de la durée de vie des variables, en supposant que les déclarations de variables ont une certaine portée. Tout cela nous entrainerait trop loin, mais fait partie de la problématique standard de la compilation.

Une autre curiosité a trait aux nombres $reg(e)$ exprimant le nombre de registres minimaux pour le calcul de e. Ce sont des entités que l'on peut définir sur tout arbre binaire, même ceux qui ne correspondent pas à des arbres de syntaxe abstraite. Ces nombres de Horton-Strahler ont un correspondant en hydrologie et en botanique. Ils donnent l'épaisseur d'une rivière en fonction de l'épaisseur de ses affluents, ou d'une branche en fonction de sa structure de branchement.

2.13 Programmes en OCaml

```
type asa =
  Feuille of char
| Noeud of char * asa * asa;;

exception Erreur_de_syntaxe of int;;

let f = " (a + a * a ) ";;
let i = ref 0;;

let rec expression () =
  let a = terme () in
  if f.[!i] = '+' then begin
    incr i;
    Noeud ('+', a, expression ()) end
  else a

and terme () =
  let a = facteur () in
  if f.[!i] = '*' then begin
    incr i;
    Noeud ('*', a, terme ()) end
  else a

and facteur () =
  if f.[!i] = '(' then begin
    incr i;
    let a = expression () in
    if f.[!i] = ')' then begin
      incr i;
      a end
    else raise (Erreur_de_syntaxe !i) end
  else
  if f.[!i] = 'a'
  then begin
    incr i;
    Feuille 'a' end
  else raise (Erreur_de_syntaxe !i);;
```

```
(* Prédicat définissant les non-terminaux *)
let est_auxiliaire y = ... ;;

(* règle.(s).(i) est la ième règle
   dont le membre gauche est s *)
let règle =
  [|  [| s00; s01; ... |];
      [| s10; s11; ... |];
      [| si0; si1; ... |];
  ... |];;

(* nbrègle.(s) est le nombre de règles
   dont le membre gauche est s *)
let nbrègle = [| s0; ...; sn |];;

(* Fonction auxiliaire sur les chaînes
   de caractères : remplacer s pos char
   renvoit une copie de la chaîne s
   avec char en position pos *)
let remplacer s pos char =
```

```
  let s1 = String.make (String.length s) ' ' in
  String.blit s 0 s1 0 (String.length s);
  s1.[pos] <- char;
  s1;;

let analyse_descendante f u =
  let b = ref false in
  let pos = ref 0 in
  while f.[!pos] = u.[!pos] do incr pos done;
  if f[!pos] = '$' && u.[!pos] = '#'
  then
    true
  else
    let y = u.[!pos] in
    if est_auxiliaire y
    then begin
      let i = ref 0 in
      let ynum = int_of_char y - int_of_char 'A' in
      while not !b && !i <= nbrègle.(ynum) do
        b := analyse_récursive
              (remplacer (u, !pos, règle.(ynum).(!i)), f);
       else incr i
     done;
     !b end
    else false;;
```

```
(* Analyse LL(1), voir page 65 *)
let rec arbre_synt_pref () =
  if f.[!pos] = 'a'
  then begin
    incr pos;
    Feuille 'a' end
  else
  if f.[!pos] = '('
  && f.[!pos + 1] = '+'
  || f.[!pos + 1] = '*'
  then begin
    let x = f.[!pos + 1] in
    pos := !pos + 2;
    let a = arbre_synt_pref () in
    let b = arbre_synt_pref () in
    if f.[!pos] = ')'
    then Noeud (x, a, b)
    else erreur (!pos) end
  else erreur(!pos);;
```

```
(* Évaluation, voir page 67 *)
type expression =
| Constante of int
| Opération of char * expression * expression;;

let rec évaluer = function
| Constante i -> i
| Opération ('+', e1, e2) -> évaluer e1 + évaluer e2
| Opération ('*', e1, e2) -> évaluer e1 * évaluer e2;;
```

Chapitre 3

Modularité et Objets

JUSQU'À présent, nous n'avons considéré que l'écriture de petits programmes ou de fonctions suffisants pour décrire des structures de données et les algorithmes correspondants. Cependant, la partie la plus importante de l'écriture de vrais programmes consiste à définir les structures nécessaires pour présenter les programmes comme un assemblage de briques s'emboîtant naturellement. Ce problème, qui peut apparaître comme purement esthétique, se révèle fondamental dès que la taille des programmes devient conséquente. En effet, si on ne prend pas garde au bon découpage des programmes en modules indépendants, on se retrouve rapidement débordé par un grand nombre de variables et de fonctions, et il devient difficile de réaliser un programme correct.

Dans ce chapitre, nous considérerons également le style de programmation par objets, souvent confondu, à tort, avec la programmation modulaire. Ce style de programmation très populaire depuis l'apparition des langages Simula et Smalltalk est un principe de programmation incrémentale, dirigée par l'organisation des données ; ce genre de programmation est plutôt utilisé pour la définition de bibliothèques graphiques, comme AWT en Java, ou de bibliothèques de fonctions appelées à distance à travers un réseau.

Dans ce chapitre, il sera question de modules, d'interfaces, de compilation séparée, de reconstruction incrémentale de programmes, et de programmation par objets.

3.1 Un exemple : les files de caractères

Pour illustrer notre chapitre, nous considérons diverses implémentations des files de caractères. Ce type de données est utilisé, par exemple, pour gérer les entrées/sorties d'un terminal (*tty driver*) ou du réseau Ethernet. Commençons par des implémentations classiques avec un tampon circulaire et avec des listes chaînées.

```
public class FC {
  private int     debut, fin;
  private boolean vide, pleine;
  private char[ ]  contenu;

  public FC (int n) {
    debut = fin = 0; vide = true; pleine = n == 0;
    contenu = new char[n];
  }

  public static void ajouter (char x, FC f) {
    if (f.pleine)
        throw new Error ("File Pleine.");
    f.contenu[f.fin] = x;
    f.fin = (f.fin + 1) % f.contenu.length;
```

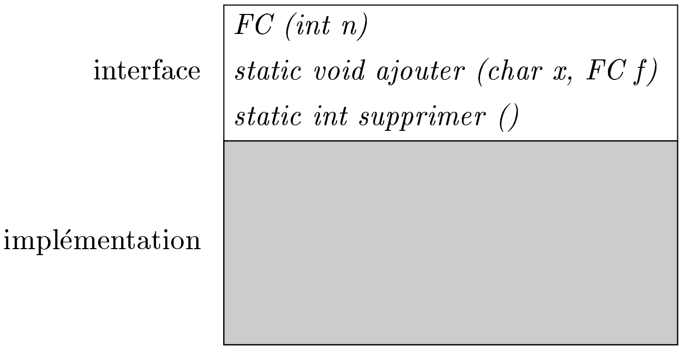


FIG. 3.1 – Interface d'un module

```
    f.vide = false; f.pleine = f.fin == f.debut;
  }

  public static char supprimer (FC f) {
    if (f.vide)
        throw new Error ("File Vide.");
    char res = f.contenu[f.debut];
    f.debut = (f.debut + 1) % f.contenu.length;
    f.vide = f.fin == f.debut; f.pleine = false;
    return res;
  }
```

Les champs *debut*, *fin*, *pleine*, *vide*, *contenu* sont privés ; le constructeur *FC* et les deux fonctions *ajouter* et *supprimer* sont publics. On peut utiliser des fonctions publiques depuis n'importe quelle classe. Seule la classe courante peut utiliser les champs et les fonctions privés. Ceci répond à un principe d'*abstraction*, puisqu'on n'accède à une classe que par ses champs et ses fonctions publics. Cette abstraction est déjà effectuée avec le regroupement des instructions en fonctions, puisqu'il est interdit d'accéder à une variable locale de l'extérieur d'une fonction. Mais avec les classes, le mécanisme dépasse la simple portée du nom des variables, puisqu'on peut aussi rendre privés des noms de champs ou de fonctions. C'est donc la tâche du programmeur de séparer le code de son programme en unités (classes en Java) dont on abstrait une interface (sa signature publique en Java). Graphiquement, comme sur la figure 3.1, le module *FC* (une classe) est divisée en une interface publique (le constructeur et les deux fonctions publiques avec leurs signatures), et une boîte noire destinée à implémenter cette interface (c'est-à-dire ses champs et fonctions privés et leur code, ainsi que le code de son interface publique).

Si on change la boîte noire (la partie implémentation) sans modifier l'interface, tous les programmes utilisant la classe *FC* n'ont pas besoin d'être modifiés. Par exemple, c'est le cas si on change l'implémentation de cette classe en utilisant des listes de caractères.

```
public class FC {
  private Liste debut, fin;
  public FC (int n) { debut = null; fin = null; }

  public static void ajouter (char x, FC f) {
    if (f.fin == null) f.debut = f.fin = new Liste (x);
    else {
      f.fin.suivant = new Liste (x);
```

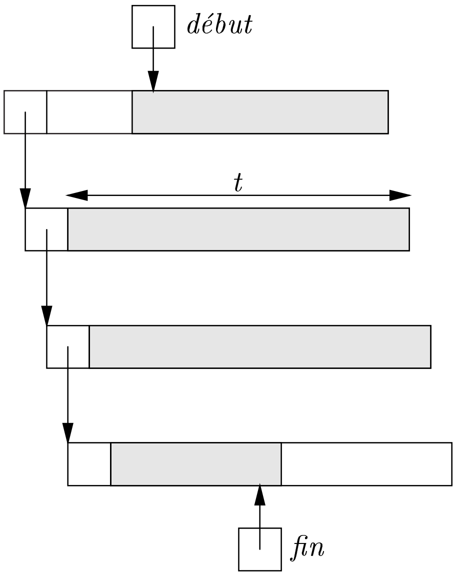


FIG. 3.2 – File de caractères

```
    f.fin = f.fin.suivant;
  }
}

public static char supprimer (FC f) {
  if (f.debut == null) throw new Error ("File Vide.");
  else {
    char res = f.debut.val;
    if (f.debut == f.fin) f.debut = f.fin = null;
    else f.debut = f.debut.suivant;
    return res;
  }
}
```

Remarquons que pour respecter la signature de l'interface publique, le constructeur *FC* est défini avec un paramètre, même s'il est inutile.

La représentation des files de caractères par des listes chaînées est coûteuse en espace mémoire. En effet, si un pointeur est représenté par une mémoire de 4 ou 8 octets (adresse mémoire sur 32 ou 64 bits), il faut 5 ou 9 octets par élément de la file, et donc $5N$ ou $9N$ octets pour une file de N caractères. La représentation par tampon circulaire est meilleure du point de vue de l'occupation mémoire. Toutefois, elle est plus statique puisque, pour chaque file, il faut réserver à l'avance la place nécessaire pour le tableau contenant le tampon circulaire.

Introduisons une troisième réalisation possible de ces files. Au lieu de représenter une file par la liste de tous les caractères la constituant, nous allons regrouper les caractères par blocs contigus de t caractères. Les premiers et derniers éléments de la liste pourront être incomplets (comme indiqué dans la figure 3.2). Ainsi, si $t = 12$, une file de N caractères utilise environ $(4 + t) \times N/t$ octets pour des adresses sur 32 bits, ce qui fait un incrément tout à fait acceptable de 1/3 d'octet par caractère. (En Java, on peut être amené à doubler la taille de chaque bloc, puisque les caractères prennent deux octets).

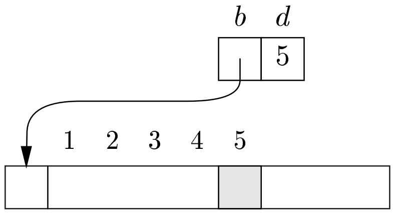


FIG. 3.3 – Adresse d'un caractère par base et déplacement

Une file de caractères sera alors décrite par un objet donnant le nombre d'éléments de la file, les bases et déplacements des premiers et derniers caractères de la file dans les premiers et derniers blocs les contenant. Par base et déplacement d'un caractère, nous entendons une référence vers un bloc de la liste contenant le caractère et son adresse relative dans ce bloc comme indiqué sur la figure 3.3. La déclaration de la classe *FC* d'une file de caractères s'effectue comme suit :

```
public class FC {
  private int cc;
  private Bloc debut_b, fin_b;
  private int debut_d, fin_d;
  public FC() { cc = 0; }
}

class Bloc {
  final static int TAILLE = 12;
  char[ ] contenu; Bloc suivant;
  Bloc () { contenu = new char[TAILLE]; suivant = null; }
}
```

Pour ajouter un caractère à la file, on doit tester si le caractère suivant est dans le même bloc ou s'il est nécessaire de prendre le bloc suivant dans la liste des blocs. On alloue donc un nouveau bloc dans le cas d'une file vide ou du franchissement d'un bloc.

```
public static void ajouter (char c, FC x) {
  Bloc b;
  if (x.cc == 0) {
    b = new Bloc();
    x.debut_b = b; x.debut_d = 0;
    x.fin_b = b; x.fin_d = -1;
  } else if (x.fin_d == Bloc.TAILLE - 1) {
    b = new Bloc();
    x.fin_b.suivant = b;
    x.fin_b = b; x.fin_d = -1;
  }
  x.fin_b.contenu[++x.fin_d] = c;
  ++x.cc;
}
```

Pour la suppression, il faut au contraire libérer un bloc si le caractère supprimé (rendu en résultat) libère un bloc.

```
public static char supprimer (FC x) {
  if (x.cc == 0)
    throw new Error ("File Vide.");
  else {
    char res = x.debut_b.contenu[x.debut_d];
    --x.cc;
    ++x.debut_d;
```

```
      if (x.debut_d >= Bloc.TAILLE) {
        x.debut_b = x.debut_b.suivant;
        x.debut_d = 0;
      }
      return res;
    }
  }
```

3.2 Interfaces et modules

Supposons qu'un programme de gestion de terminaux utilise des files de caractères, une pour chaque terminal. Grâce au principe d'abstraction, il ne mélange pas la gestion des files de caractères avec le reste de la logique du programme. Sa conception est donc structurée, les files de caractères sont dans un *module* à part. Le programme utilisant les files de caractères n'a pas besoin de connaître tous les détails de l'implémentation de ces files. Il ne connaît que la déclaration des types utiles dans la classe *FC* : le nom de la classe et les trois fonctions pour initialiser une file vide, ajouter un élément au bout de la file et retirer le premier élément. On se contente de l'*interface* suivante :

```
public class FC {

  public FC () {...}
  /* Retourne une file vide */

  public static void ajouter (char c, FC x) {...}
  /* Ajoute c au bout de la file x */

  public static char supprimer (FC x) {...}
  /* Supprime le premier caractère c de x et rend c comme résultat */
  /* Si x est vide, l'erreur "File Vide" est levée */
}
```

Les files de caractères ne seront manipulées qu'à travers cette interface. Pas question de connaître la structure interne de ces files, ni de savoir si elles sont organisées par de simples listes, des tableaux circulaires ou des blocs enchaînés. On dira que le programme utilisant des files de caractères à travers l'interface précédente *importe* cette interface. Les corps des fonctions sur les files sont dans la partie *implémentation* du *module* des files de caractères. Dans l'interface d'un module, il n'y a donc que les types, les champs ou les fonctions que l'on veut exporter ou rendre publiques. Il est bon d'y commenter la fonctionnalité de chaque élément pour en comprendre sa signification, par la simple lecture de l'interface. Dans un module, il y a donc toute une partie *cachée* comprenant les types et les corps des procédures ou des fonctions que l'on veut rendre privées. C'est ce qu'on appelle le principe d'encapsulation.

Comment y arriver en Java ? Comme déjà vu, le plus simple est de se servir des modificateurs d'accès dans la déclaration des variables ou des méthodes de la classe *FC*. Le qualificatif *private* d'un champ ou d'une fonction signifie que seuls les instructions à l'intérieur de la classe où ce champ ou cette fonction sont définis pourront les accéder. Au contraire *public* dit qu'un champ est accessible par toutes les classes. L'option par défaut est de rendre public les champs à l'intérieur du *paquetage* où il est défini (cf. plus loin), comme cela était le cas pour les champs de la classe *Bloc*)

On peut avoir à cacher non seulement le code des procédures réalisant l'interface, mais aussi des variables et des fonctions internes, en composant les modules pour en fabriquer d'autres. Supposons dans notre exemple, que, pour être efficace (ce qui peut être le cas pour piloter un périphérique), nous voulions avoir notre propre stratégie d'allocation pour les blocs. On construit alors une liste des blocs libres *listeLibre* à l'initialisation du chargement de la classe *FC* et on utilise des fonctions *nouveauBloc* et *libererBloc* comme suit :

```
class FC {
  private int cc;
  private Bloc debut_b, fin_b;
  private int debut_d, fin_d;
  public FC () { cc = 0; }

  public static void ajouter (char c, FC x) {
    Bloc b;
    if (x.cc == 0) {
      b = nouveauBloc();
      x.debut_b = b; x.debut_d = 0;
      x.fin_b = b; x.fin_d = -1;
    } else if (x.fin_d == Bloc.TAILLE - 1) {
      b = Bloc.allouer();
      x.fin_b.suivant = b;
      x.fin_b = b; x.fin_d = -1;
    }
    x.fin_b.contenu[++x.fin_d] = c;
    ++x.cc;
  }

  public static char supprimer (FC x) {
    if (x.cc == 0)
      throw new Error ("File Vide.");
    else {
      char res = x.debut_b.contenu[x.debut_d];
      --x.cc;
      ++x.debut_d;
      if (x.cc <= 0)
        Bloc.liberer (x.debut_b);
      else if (x.debut_d >= Bloc.TAILLE) {
        Bloc b = x.debut_b;
        x.debut_b = x.debut_b.suivant;
        x.debut_d = 0;
        Bloc.liberer (b);
      }
      return res;
    }
  }
}

class Bloc {
  final static int TAILLE = 12;
  char[ ] contenu; Bloc suivant;
  Bloc () { contenu = new char[TAILLE]; suivant = null; }

  private final static int NB_BLOCS = 1000;
  private static Bloc listeLibre;

  static {
    listeLibre = null;
    for (int i=0; i < NB_BLOCS; ++i) {
      Bloc b = new Bloc();
      b.suivant = listeLibre;
      listeLibre = b;
```

```
    }
  }

  private static Bloc allouer () {
    Bloc b = listeLibre;
    listeLibre = listeLibre.suivant;
    b.suivant = null;
    return b;
  }

  private static void liberer (Bloc b) {
    b.suivant = listeLibre;
    listeLibre = b;
  }
}
```

La variable *listeLibre* reste cachée, puisque cette variable n'est pas définie dans l'interface des files de caractères. Il en est de même pour les fonctions d'allocation et de libération des blocs. Faisons trois remarques. Premièrement, il est fréquent qu'un module nécessite une procédure d'initialisation. En Java, on le fait en mettant quelques instructions, précédées du mot-clé *static* dans la déclaration de la classe. Ces instructions ne seront exécutées qu'une seule fois au chargement de la classe. (Une classe est chargée lors du premier accès à un élément de cette classe). Deuxièmement, pour ne pas compliquer le programme, nous ne testons pas le cas où la liste des blocs libres devient vide. Alors l'allocation d'un nouveau bloc libre devient impossible. Troisièmement, il y a une belle structure hiérarchique des divers modules d'un programme : le *driver* se sert des files de caractères, lesquelles se servent d'un module d'allocation de blocs.

Pour résumer, un module contient deux parties : une interface exportée qui contient les constantes, les types, les variables et la signature des fonctions ou procédures que l'on veut rendre publiques, une partie implémentation qui contient la réalisation des objets de l'interface. L'interface est la seule porte ouverte vers l'extérieur. Dans la partie implémentation, on peut utiliser tout l'arsenal possible de la programmation. On ne veut pas que cette partie soit connue de son utilisateur. Si on arrive à ne laisser public que le strict nécessaire pour utiliser un module, on aura grandement simplifié la structure d'un programme. Il faut donc bien faire attention aux interfaces, car une bonne partie de la difficulté de la programmation réside dans le bon choix des interfaces.

Découper un programme en modules permet aussi la réutilisation des modules, la construction hiérarchique des programmes puisqu'un module peut lui-même être aussi composé de plusieurs modules, le développement indépendant de programmes par plusieurs personnes dans un même projet de programmation. Il facilite les modifications, si les interfaces restent inchangées. Ici, nous insistons sur la structuration des programmes, car tout le reste n'est que corollaire. Tout le problème de la modularité se résume à isoler des parties de programme comme des boîtes noires, dont les seules parties visibles à l'extérieur sont les interfaces. Bien définir un module assure la sécurité dans l'accès aux variables ou aux procédures, et est un bon moyen de structurer la logique d'un programme. Une deuxième partie de la programmation consiste à assembler les modules de façon claire.

Beaucoup de langages de programmation ont une notion explicite de modules et d'interfaces, par exemple Ada, Caml-light, Cedar, Clu, Mesa, Modula-2, Modula-3, et Oberon. Certains (comme SML/NJ et Ocaml) ont même des modules paramétriques très sophistiqués. En C ou C++, il n'y a aucune notion de module, et on utilise les fichiers *include* pour simuler les interfaces des modules, en faisant coïncider les notions de module et de compilation séparée.

En Java, la notion de modularité est plus dynamique et reportée dans le langage de programmation avec les modificateurs d'accès. Il n'y a pas de phase d'éditions de liens permettant de faire coïncider interfaces et implémentations. Seul le chargeur dynamique (le *ClassLoader*) ou l'interpréteur (la JVM, c'est-à-dire la machine virtuelle de Java) font quelques vérifications. Il existe une notion d'interfaces avec le mot-clé *Interface* et de classes implémentant ces interfaces, mais ces notions sont plutôt réservées à la construction de classes paramétrées ou à la gestion dynamique simultanée de plusieurs implémentations pour une même interface.

3.3 Structure de l'espace des noms, paquetages

Il existe aussi une autre forme de modularité en se servant de la restriction dans l'espace des noms. Nous avons vu que pour accéder à des variables ou à des fonctions, on pouvait qualifier leurs noms par le nom de la classe (ou d'un objet pour les méthodes dynamiques). Ainsi on peut écrire *Liste.ajouter*, *FC.TAILLE*, *args.length()*, *System.out.print*. Mais on peut aussi préfixer, les noms par un nom de paquetage. Un paquetage (*package*) permet de regrouper un ensemble de classes qui ont des fonctionnalités proches, par exemple les appels-système, les entrées/sorties, etc. Un paquetage est associé à un répertoire du système de fichiers, dont les classes sont des fichiers. On peut spécifier le paquetage où se trouve une classe en la précédent par l'instruction

`package` $id_1.id_2.\ldots.id_k$`;`

où id_1, id_2, $\ldots id_k$ sont des identificateurs quelconques. Pour accéder à une classe d'un tel paquetage, le nom de cette classe est précédée par le nom complet du paquetage dont elle fait partie. Par exemple, si on veut utiliser la classe *FC* du sous-paquetage *listes* de mon paquetage *ma_librairie*, on écrit :

```
ma_librairie.listes.FC f = new ma_librairie.listes.FC();
```

On peut revenir à une notation plus brève (qui évite de mettre le nom complet du paquetage) en utilisant l'instruction *import* en tête d'un programme.

```
import ma_librairie.listes.FC;
import java.io.*;
import java.lang.*;
import java.util.*;
```

Le dernier champ du paquetage importé est soit un nom de classe, soit un astérisque (`*`). Dans ce dernier cas, ce sont toutes les classes contenues dans le répertoire désigné qui sont importées.

L'emplacement des paquetages dépend de deux paramètres : le nom $id_1.id_2.\ldots.id_k$ qui désigne, dans le système Unix, l'emplacement $id_1/id_2/\ldots/id_k$ dans l'arborescence des répertoires du système de fichiers ; la variable d'environnement *CLASSPATH* qui donne une suite de racines possibles pour l'arborescence considérée. On considèrera alors le premier paquetage trouvé dans l'ordre de la liste des racines possibles. Les paquetages peuvent aussi se trouver dans une forme compressée des répertoires, très utilisée en Java, les fichiers *.jar* (*java archives*), qui contiennent en un seul fichier toute une arborescence de paquetages (cf. la commande Unix *jar*). Les fichiers *.zip* peuvent aussi être utilisés. Pour donner une valeur à *CLASSPATH*, on utilise, par exemple, la commande Unix :

```
% setenv CLASSPATH ".:$HOME/if431/DM1:/users/profs/info/chassignet/Jaxx"
```

Enfin, on peut aussi trouver le paquetage à l'endroit des paquetages standards (souvent regroupés dans un grand ficher *.jar* ou *.zip*). C'est le cas pour les paquetages *java.util* qui contient des classes standards pour les tables, les piles, les tableaux de taille variable, etc, pour *java.io* qui contient les classes d'entrées-sorties, pour *java.awt* qui contient les classes graphiques, *java.math* qui contient les classes mathématiques.

Le répertoire contenant un fichier source *.java* est aussi considéré comme un paquetage anonyme. Un fichier source ne commençant pas par une instruction *package* est considéré comme déclarant des classes dans le paquetage anonyme, c'est-à-dire dans le répertoire courant.

Les paquetages interviennent aussi dans les droits d'accès aux champs des classes. Plus exactement, trois types d'accès sont possibles pour les membres d'une classe :
– public en préfixant un champ par *public* pour permettre l'accès depuis toutes les classes.
– privé en préfixant un champ par *private* pour restreindre l'accès aux seules expressions ou fonctions dans la classe courante.
– amical en ne mettant aucun préfixe devant la déclaration d'un champ pour autoriser l'accès depuis toutes les classes du même paquetage (c'est donc l'accès par défaut).

Exercice 17 Montrer qu'on peut accéder facilement aux champs ou fonctions des classes compilées dans le même répertoire.

3.4 Compilation séparée et librairies

La compilation d'un programme consiste à fabriquer le binaire exécutable par le processeur de la machine. Pour des programmes de plusieurs milliers de lignes, il est bon de les découper en des fichiers compilés séparément. En Java, le code généré est indépendant de la machine qui l'exécute, ce qui permet de l'exécuter sur toutes les architectures à travers le réseau. Le code (*byte-code*) est interprété par un interpréteur dépendant lui de l'architecture sous-jacente, la machine virtuelle Java (encore appelée JVM pour en anglais *Java Virtual Machine*). Les fichiers de *byte-code* ont le suffixe *.class*, les fichiers sources ayant eux d'habitude le suffixe *.java*. Pour compiler un fichier source sous un système Unix, la commande :

```
% javac FC.java
```

permet d'obtenir le fichier de *byte-code* de nom *FC.class* qui peut être utilisé par d'autres modules compilés indépendamment. Supposons qu'un fichier *TTY.java* contenant un gestionnaire de terminaux utilise les fonctions sur les files de caractères. Ce programme contiendra des lignes du genre :

```
class TTY {
  FC in, out;
  TTY () { in = new FC(); out = new FC(); }

  static int Lire (FC in) { ··· }

  static void Imprimer (FC out) { ··· }
```

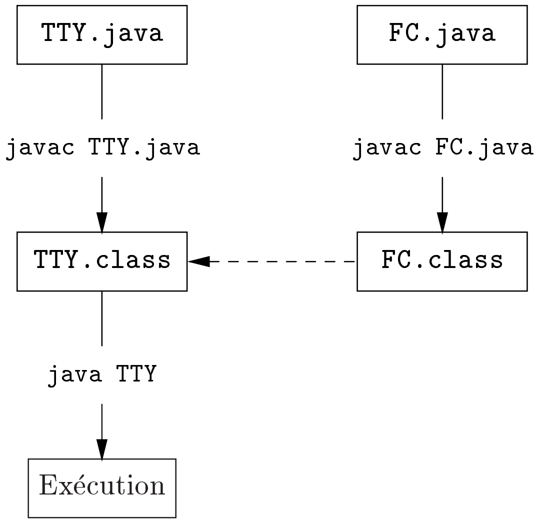


FIG. 3.4 – Compilation séparée

```
}
```

En Unix, on devra compiler séparément *TTY.java* et on lancera la machine Java sur *TTY.class* par les commandes :

```
% javac TTY.java
% java TTY
```

Remarquons que le suffixe *.class* ne doit curieusement pas apparaître dans la deuxième commande. La dernière commande cherche la fonction publique *main* dans la classe *TTY.class* et démarre la machine virtuelle Java sur cette fonction. Les diverses classes utilisées sont chargées au fur et à mesure de leur utilisation. Contrairement à beaucoup de langages de programmation, il n'y a pas (pour le meilleur et pour le pire) de phase d'édition de liens en Java. Tout se fait dynamiquement. Graphiquement, les phases de compilation sont représentées par la figure 3.4.

3.5 Dépendances entre modules

Lors de l'écriture d'un programme composé de plusieurs modules, il est commode de décrire les dépendances entre modules et la manière de reconstruire les binaires exécutables. Ainsi on peut recompiler le strict nécessaire en cas de modification d'un module. Dans l'exemple de notre gestionnaire de terminaux, nous voulons indiquer que les dépendances induites par la figure 3.5 pour reconstruire le code objet *TTY.class*. La description des dépendances varie selon le système. La commande *javac* fait l'analyse de dépendances et compile ce qui est nécessaire. De même, la commande *Run* ou *Compile* en CodeWarrior sur Mac. De manière plus générale sous le système Unix, on utilise des *Makefile* et la commande *make*.

Supposons pour un moment notre programme écrit en C, avec un fichier d'interface *fc.h*. Le fichier de dépendances serait ainsi écrit :

```
tty: tty.o fc.o
        cc -o tty tty.o fc.o
```

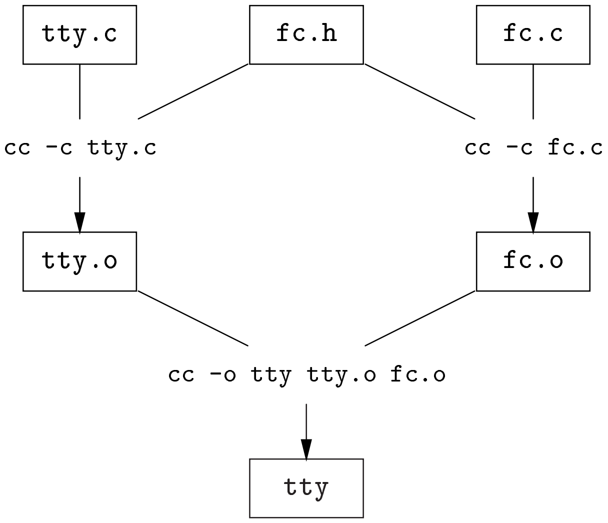


FIG. 3.5 – Dépendances dans un *Makefile*

```
tty.o: tty.c fc.h
       cc -c tty.c

fc.o: fc.c fc.h
       cc -c fc.c
```

Après “ :”, il y a la liste des fichiers dont dépend le but mentionné au début de la ligne. Dans les lignes suivantes, il y a la suite de commandes à effectuer pour obtenir le fichier but. La commande Unix *make* considère le graphe des dépendances et calcule les commandes nécessaires pour reconstituer le fichier but. Si les interdépendances entre fichiers sont représentés par les arcs d'un graphe dont les sommets sont les noms de fichier, on fait un tri topologique sur le graphe de dépendances pour trouver la suite des commandes à effectuer.

3.6 Un exemple de module en C

Comme en Java, le langage C n'a pas de système de modules. Nous reprenons l'exemple des files de caractères (en fait telles qu'elles se trouvaient dans le système Unix version 7 pour les gestionnaires de terminaux). C'est aussi l'occasion de constater comment la programmation en C permet certaines acrobaties, peu recommandables car on aurait pu suivre la technique d'adressage des caractères dans les blocs comme en Java. La structure des files est légèrement différente car on adresse directement les caractères dans un bloc au lieu du système base et déplacement. Le débordement de bloc est testé en regardant si on est sur un multiple de la taille d'un bloc, car on suppose le tableau des blocs aligné sur un multiple de cette taille. Le fichier interface *fc.h* est

```
#define NCLIST 80    /* max total clist size   */
#define CBSIZE 12    /* number of chars in a clist block */
#define CROUND 0xf   /* clist rounding : sizeof(int + CBSIZE-1 */

/*
 * A clist structure is the head
 * of a linked list queue of characters.
```

```
 * The characters are stored in 4-word
 * blocks containing a link and several characters.
 * The routines fc_get and fc_put
 * manipulate these structures.
 */
struct clist
{
    int   c_cc;            /* character count */
    char  *c_cf;           /* pointer to first char  */
    char  *c_cl;           /* pointer to last char   */
};

struct cblock {
    struct cblock *c_next;
    char          c_info[CBSIZE];
};

typedef struct clist *fc_type;

int  fc_put(char c, fc_type p);
int  fc_get(fc_type p);
void fc_init(void);
```

Dans la partie implémentation qui suit, on remarque l'emploi de la directive *static* (celle de C, et non de Java !) qui permet de cacher à l'édition de liens des variables, procédures ou fonctions privées qui ne seront pas considérées comme externes. Il est possible en C de cacher la représentation des files, en ne déclarant le type *fc_type* que comme un pointeur vers une structure *clist* non définie. Les fonctions retournent un résultat entier qui permet de retourner des valeurs erronées comme -1. Le fichier *fc.c* est

```
#include <stdlib.h>
#include <fc.h>

static struct cblock  cfree[NCLIST];
static struct cblock  *cfreelist;

int fc_put(char c, fc_type p)
{
    struct cblock *bp;
    char *cp;
    register s;

    if ((cp = p->c_cl) == NULL || p->c_cc < 0 ) {
        if ((bp = cfreelist) == NULL)
            return(-1);
        cfreelist = bp->c_next;
        bp->c_next = NULL;
        p->c_cf = cp = bp->c_info;
    } else if (((int)cp & CROUND) == 0) {
        bp = (struct cblock *)cp - 1;
        if ((bp->c_next = cfreelist) == NULL)
            return(-1);
        bp = bp->c_next;
        cfreelist = bp->c_next;
        bp->c_next = NULL;
        cp = bp->c_info;
    }
    *cp++ = c;
    p->c_cc++;
    p->c_cl = cp;
    return(0);
```

```
}

int fc_get(fc_type p)
{
    struct cblock *bp;
    int c, s;

    if (p->c_cc <= 0) {
        c = -1;
        p->c_cc = 0;
        p->c_cf = p->c_cl = NULL;
    } else {
        c = *p->c_cf++ & 0xff;
        if (--p->c_cc<=0) {
            bp = (struct cblock *)(p->c_cf-1);
            bp = (struct cblock *) ((int)bp & ~CROUND);
            p->c_cf = p->c_cl = NULL;
            bp->c_next = cfreelist;
            cfreelist = bp;
        } else if (((int)p->c_cf & CROUND) == 0){
            bp = (struct cblock *)(p->c_cf-1);
            p->c_cf = bp->c_next->c_info;
            bp->c_next = cfreelist;
            cfreelist = bp;
        }
    }
    return(c);
}

void fc_init()
{
    int ccp;
    struct cblock *cp;

    ccp = (int)cfree;
    ccp = (ccp+CROUND) & ~CROUND;
    for(cp=(struct cblock *)ccp; cp <= &cfree[NCLIST-1]; cp++) {
        cp->c_next = cfreelist;
        cfreelist = cp;
    }
}
```

3.7 Modules en OCaml

On construit pour tout module deux fichiers *fc.mli* (pour *ML interface*) et *fc.ml* (le fichier d'implémentation du module). Le premier est un fichier d'interface dans lequel on donne la signature de tous les types, variables ou fonctions exportées. Par exemple :

```
(* Files de caractères *)
type t
  (* Le type des files de caractères *)
exception FileVide
  (* Levée quand supprimer est appliquée à une file vide. *)
val vide: unit -> t
  (* Retourne une nouvelle file, initialement vide. *)
val ajouter: char -> t -> t
  (* ajouter c x ajoute le caractère c à la fin de la file x. *)
val supprimer: t -> char
  (* supprimer x enlève et retourne le premier élément de la file x
```

ou lève l'exception *FileVide* si la queue est vide. `*)`

Le deuxième contient l'implémentation des files de caractères, où on fournit les structures de données ou le code correspondant à chaque déclaration précédente :

```
type t = {mutable cc: int;
          mutable debut_b: liste_de_blocs; mutable debut_d: int;
          mutable fin_b: liste_de_blocs; mutable fin_d: int} and
     liste_de_blocs = Nil | Cons of bloc and
     bloc = {contenu: string; mutable suivant: liste_de_blocs};;

exception FileVide;;

let taille_bloc = 12;;

let vide() = {cc = 0; debut_b = Nil; debut_d = 0; fin_b = Nil; fin_d = 0};;

let ajouter c x =
    let nouveau_bloc() =
      Cons {contenu = String.make taille_bloc ' '; suivant = Nil} in
    if x.cc = 0 then begin
       let b = nouveau_bloc() in
       x.debut_b <- b; x.debut_d <- 0;
       x.fin_b <- b; x.fin_d <- -1;
    end else if x.fin_d = taille_bloc - 1 then begin
       let b = nouveau_bloc() in
       (match x.fin_b with
           Cons r -> r.suivant <- b
         | Nil -> ()
       );
       x.fin_b <- b; x.fin_d <- -1;
    end;
    x.fin_d <- x.fin_d + 1;
    (match x.fin_b with
       Cons r -> r.contenu.[x.fin_d] <- c
     | Nil -> ()
    );
    x.cc <- x.cc + 1;
    x;;

let supprimer x =
    if x.cc = 0 then
       raise FileVide
    else match x.debut_b with
      Nil -> failwith "Cas impossible"
    | Cons r -> let res = r.contenu.[x.debut_d] in
       x.cc <- x.cc - 1;
       x.debut_d <- x.debut_d + 1;
       if x.debut_d >= taille_bloc then begin
         x.debut_b <- r.suivant;
         x.debut_d <- 0;
       end;
       res;;
```

Dans un deuxième module, par exemple un gestionnaire de terminaux dont le code se trouve dans un fichier *tty.ml*, on peut utiliser les noms du module *fc.mli* en les qualifiant par le nom du module suivi du symbole (.) point.

```
let nouveau_tty =
  let x = Fc.vide() in
  ...
  let y = Fc.ajouter c x ....
```

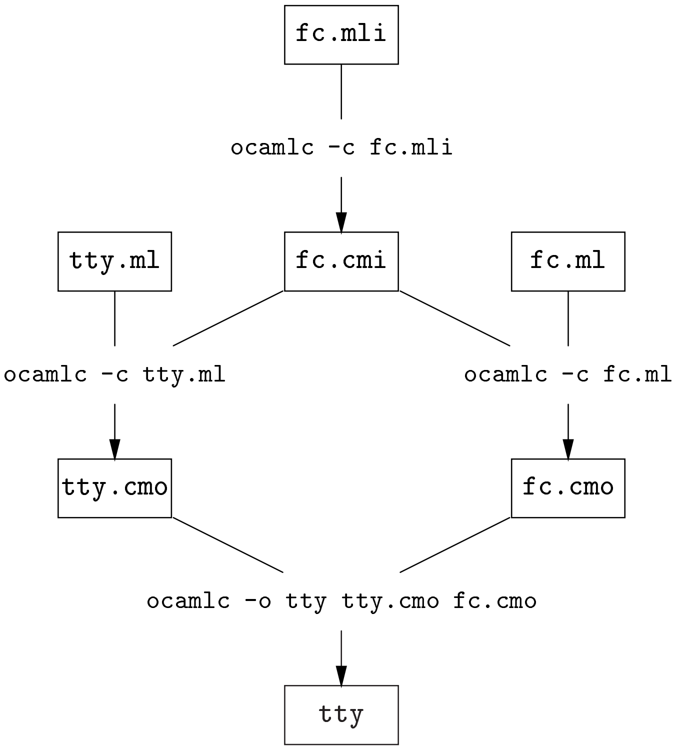


FIG. 3.6 – Dépendances entre modules Caml

Si on n'a pas envie d'utiliser cette notation longue, on supprime le préfixe avec la directive suivante en tête de *tty.ml*.

```
open Fc;;
```

Les dépendances entre modules se font comme dans le cas de C. On se sert de la commande *make* de Unix et du *makefile* suivant. Remarquons que les fichiers *.ml* se compilent en fichiers *.cmo*, les fichiers *.mli* en *.cmi*, et que l'on finit avec un fichier *tty* directement exécutable.

```
tty: tty.cmo fc.cmo
       ocamlc -o tty tty.cmo fc.cmo

tty.cmo: tty.ml fc.cmi
       ocamlc -c tty.ml

fc.cmo: fc.ml
       ocamlc -c fc.ml

fc.cmi: fc.mli
       ocamlc -c fc.mli
```

Enfin, en Caml, on n'est pas forcé de définir un fichier d'interface, auquel cas le compilateur générera automatiquement un fichier *.cmi* en supposant toutes les variables et fonctions publiques dans le module d'implémentation. Toutefois, c'est plus sûr de définir soi-même les fichiers d'interface.

	modification des données	modification du programme
programmation procédurale	changement global	changement local
programmation par objets	changement local	changement global

FIG. 3.7 – Programmation procédurale, programmation par objets

3.8 Programmation par objets

La programmation par objets est devenue très populaire. C'est maintenant un argument de vente pour les langages de programmation, puisque tous les langages modernes ont une extension objets. Pourtant, ce style de programmation n'est pas facile à définir. La théorie des types correspondante a fait couler beaucoup d'encre. Si on arrive à présent à mieux la comprendre, les problèmes pratiques associés ne sont pas encore vraiment résolus. En demandant à des informaticiens (chevronnés) ce qu'est la programmation par objets, on obtient des réponses variées : un objet est une boîte avec des boutons, un objet est un enregistrement avec des champs fonctionnels, un objet est un module, un objet a un état mémoire, un objet est réutilisable, un objet sert à faire de l'héritage, un objet permet de faire de la liaison retardée, etc. Un non-informaticien m'a une fois bien résumé la situation en remarquant qu'un objet était un peu tout et n'importe quoi ! Cependant, nous nous lançons dans une tentative d'explication.

La programmation classique a un caractère procédural. Les instructions d'un programme sont regroupées en fonctions (encore appelées procédures) ; à partir des fonctions, on construit d'autres fonctions. Dans la programmation par objets, la programmation est organisée autour des données. Une donnée contient ses valeurs et les fonctions (encore appelées méthodes) opérant sur elle. Dans le style procédural, si on rajoute quelques champs à un type de donnée, on doit faire un changement global dans toutes les fonctions qui agissent sur ce type de donnée. Dans le style par objets, on ne fait qu'un changement local en créant une sous-classe (avec les méthodes associées) de la classe correspondant à ce type de donnée. Mais dans la programmation procédurale, si on modifie une fonction, le changement est local à cette fonction, alors que dans la programmation par objets, le changement doit être effectué dans toutes les méthodes de même nom que cette fonction. Ceci est résumé sur la figure 3.7.

La programmation par objets a une deuxième caractéristique : c'est la notion de programmation *incrémentale*. Comme l'organisation du programme est centrée sur les données, on peut se contenter de faire suivre son évolution par spécialisation de ses différentes classes de données à partir de classes plus générales. On ne modifie pas un programme, on ne fait qu'ajouter des incréments (à une bibliothèque de fonctions préexistantes). L'exemple typique est celui d'une boîte à outils graphique, où les données (les fenêtres, boutons, barres de défilement) ont beaucoup de paramètres et où on ne code principalement que des incréments par rapport à des modèles de classes graphiques fournies en standard. Ainsi les fenêtres à coins arrondis sont une spécialisation des fenêtres rectangulaires, et beaucoup des fonctions valables sur les fenêtres rectangulaires restent valides sur les fenêtres à coins arrondis. Si nécessaire, on redéfinira quelques

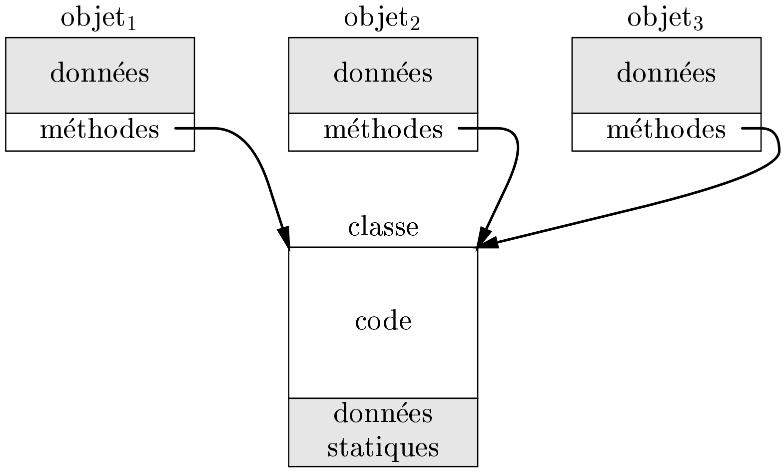


FIG. 3.8 – Trois objets d'une même classe

fonctions, telles que celle pour l'affichage d'une fenêtre ou l'interaction avec la souris. La programmation incrémentale fonctionne donc bien dans une univers où toutes les données sont proches, et où de petites modifications sont suffisantes pour définir de nouvelles données.

Troisièmement, on associe souvent la notion de *sous-typage* au style de programmation par objets. Il n'y a pas la place ici pour traiter du sous-typage, notion fine de la programmation. Disons simplement que, si T est un sous-type de T' (écrivons $T \leq T'$), alors si une variable ou une fonction est de type T, elle peut aussi être considérée de type T', et donc être utilisée dans tout contexte où une expression de type T' est attendue. En Java, par exemple, on a $byte \leq short \leq int$, $char \leq int$, etc. Nous savons aussi qu'on a $C \leq Object$ pour toute classe C. Tout le problème du sous-typage consiste à étudier son interaction avec les différentes constructions du langage de programmation, c'est-à-dire avec les fonctions et les classes en Java.

Enfin, avant d'aller plus loin dans la programmation par objets, rappelons qu'un objet est techniquement l'instance d'une classe, c'est-à-dire une entité de type classe. En Java, c'est la seule manière (avec les tableaux) de faire des données composites. Les objets ont un état (qui donne la valeur de leurs champs de données) et un ensemble de méthodes attachées. Il y a deux types de méthodes : les méthodes f, g, etc, statiques qui sont invoquées par les expressions de la forme $C.f$, $C.g$ si C est le nom de la classe où elles sont définies ; les méthodes f', g', etc, non statiques, qui sont appelées par les expressions $o.f'$, $o.g'$, si o est l'objet dont elles sont membres. A priori, ce deuxième type de méthodes parait superflu. Pourtant, on est là au cœur de la programmation par objets, car l'idée est qu'on ne connait pas *a priori* la définition de f', g', etc. Nous verrons plus tard la notion de spécialisation de méthodes, qui correspond à cette notation.

D'un point de vue implémentation, un objet a la représentation de la figure 3.8. Les champs de données sont différents pour les objets d'une même classe. Seuls les champs de données statiques sont uniquement représentés dans la classe. La partie code peut être regroupée au niveau de la classe pour les méthodes (statiques et non statiques), puisque toutes les méthodes partagent un même code. En fait, tout objet a un point d'entrée pour chaque méthode non statique, ce point d'entrée n'est qu'un pointeur vers la zone de code unique pour la classe. Pour résumer, un objet est un enregistrement avec des champs de données et des champs fonctionnels (représentés par un simple pointeur vers la zone de code) pour les méthodes.

3.9 Hiérarchie des classes et sous-typage

Considérons l'exemple classique des points colorés sur $\mathbf{R}^2$, où chaque point a deux coordonnées x et y. On considère deux fonctions statiques *translation* et *rotation* qui font les opérations correspondantes sur le point p pris en argument.

```
class Point {
  double x, y;
  Point () { }
  Point (double x0, double y0) { x = x0; y = y0; }

  static void translation (Point p, double dx, double dy) {
    p.x = p.x + dx; p.y = p.y + dy;
  }

  static void rotation (Point p, double theta) {
    double x = p.x * Math.cos(theta) - p.y * Math.sin(theta);
    double y = p.x * Math.sin(theta) + p.y * Math.cos(theta);
    p.x = x; p.y = y;
} }
```

Considérons la classe *PointC* des points colorés, définie comme une sous-classe de la classe *Point*, grâce au mot-clé *extends* :

```
class PointC extends Point {
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC (double x0, double y0, int col) { super(x0, y0); c = col;}

  public static void main (String[ ] args) {
    PointC p = new PointC(3, 4, JAUNE) ;
    rotation (p, Math.PI);
} }
```

Un point coloré p a un champ supplémentaire c pour sa couleur. Les fonctions de translation et de rotation, qui ne font pas intervenir la couleur, restent les mêmes. La sous-classe *PointC* hérite donc ces deux méthodes, ainsi que les deux champs x et y, et on peut appliquer *translation* et *rotation* à des points colorés, comme cela se produit dans le programme principal précédent, où il y a donc une conversion implicite du point coloré p en un point non coloré pour le passer en argument de la fonction *rotation*.

Au contraire, pour passer d'un point à un point coloré, il faut faire une conversion explicite. Considérons une autre définition de la fonction de rotation qui rend le point modifié comme résultat. Alors cette fonction et *main* sont modifiées, chacune dans sa classe, comme suit :

```
  static Point rotation (Point p, double theta) {
    double x = p.x * Math.cos(theta) - p.y * Math.sin(theta);
    double y = p.x * Math.sin(theta) + p.y * Math.cos(theta);
    p.x = x; p.y = y;
    return p;
  }
  ...
```

```
public static void main (String[ ] args) {
  PointC p = new PointC(3, 4, JAUNE) ;
  p = (PointC) rotation (p, Math.PI);
}
```

Comme la fonction *rotation* retourne un point pas forcément coloré, on doit tester dynamiquement si ce point est coloré, ce qui est fait par la conversion explicite de son résultat avant de l'affecter à la variable p de type point coloré. Ce test peut échouer et lever l'exception *ClassCastException*. Cela n'est pas le cas dans notre exemple.

Modifions encore notre programme, et supposons à présent que le point retourné par la fonction *rotation* est un nouveau point. Alors cette fonction ne fait plus d'effet de bord sur son argument (ce qui est bien en général). Son code et celui de *main* deviennent :

```
static Point rotation (Point p, double theta) {
  double x = p.x * Math.cos(theta) - p.y * Math.sin(theta);
  double y = p.x * Math.sin(theta) + p.y * Math.cos(theta);
  return new Point (x, y);
}
...
public static void main (String[ ] args) {
  PointC p = new PointC (3, 4, JAUNE) ;
  p = (PointC) rotation (p, Math.PI);
}
```

L'exécution de *main* lève maintenant l'exception *ClassCastException*, puisque la valeur retournée par la fonction *rotation* est un point non coloré. Sa conversion en point coloré est impossible, puisque son champ couleur n'existe pas.

En résumé, il y a conversion implicite d'un objet d'une sous-classe en objet de sa surclasse. Mais il faut convertir explicitement un objet d'une classe pour le spécialiser en un objet d'une sous-classe. Cette conversion n'est possible que s'il est déjà (à conversion implicite près) un objet de cette sous-classe. On peut le tester dynamiquement, car tout objet garde un indicateur de la classe dans laquelle il a été créé. Donc, en Java, si la classe C est une sous-classe de la classe D, alors C est un sous-type de D ($C \leq D$). Et si $C \to C'$ désigne le type des fonctions de C dans C', et $D \to D'$ le type des fonctions de D dans D', on a

$$D \leq C \quad \text{et} \quad C' \leq D' \quad \Rightarrow \quad C \to C' \leq D \to D'$$

Cette proposition, vraie dans tous les systèmes de sous-typage, indique que le sous-typage se conduit différemment dans les arguments d'une fonction (contravariance) et dans son résultat (covariance). Plus simplement, dans l'exemple précédent, on retrouve cette formule on considérant $C = PointC$, $D = Point$, $C' = D' = void$ (dans le premier cas) et $C' = D' = Point$ (dans le deuxième cas).

Quand une classe est une sous-classe d'une autre, on dit aussi qu'elle hérite de sa surclasse. En Java, la hiérarchie entre les classes a une structure arborescente. Toute classe n'a qu'une seule classe parente. On dit encore que l'héritage est simple. Il existe des langages de programmation où une classe peut hériter de plusieurs classes (héritage multiple), par exemple Smalltalk, C++, Ocaml, mais alors l'implémentation de ces langages est plus délicate, car, dans les langages à héritage simple, on connait l'emplacement de chacun de ses champs avant l'exécution du programme. La classe racine de la hiérarchie des classes est la classe *Object*. (En Java, les interfaces permettent de faire de l'héritage multiple ; cette partie du langage a une implémentation moins efficace).

Une remarque un peu plus subtile consiste à noter que le typage de Java nous force à faire une conversion explicite de type dans la deuxième définition de *rotation*. En fait, la fonction *rotation* a un type $\forall C \leq Point.\, C \times double \rightarrow C$, elle peut prendre un argument de type C arbitraire sous-type de *Point* et un réel *double* comme deuxième argument, puis retourner un objet du type C. Avec un tel système de type, on sait que la fonction retourne un point coloré si l'argument est de type point coloré, et la conversion explicite devient inutile dans la fonction *main*. Une autre solution pour typer *rotation* consiste à donner le type $\langle x : int,\, y : int,\, \rho\rangle \times double \rightarrow \langle x : int,\, y : int,\, \rho\rangle$ où ρ est une variable de type désignant la possibilité de rajouter des champs (type d'une rangée). Alors à nouveau sans conversion de type, on peut assurer que le retour de la fonction de rotation redonne bien un point coloré si on lui passe en argument un point coloré. Cette solution, se servant de sous-typage structurel, est utilisée en Ocaml. Enfin, on peut remarquer qu'on a plus de difficulté à typer la troisième version des points colorés (qui fait une rotation sans effet de bord). La raison est qu'on crée alors un nouvel objet et que celui a alors le type *Point*. Pourtant, on peut arriver à lui donner un des deux types précédents si on dispose d'une opération de clonage des objets, qui copie un objet existant avec son type. On retrouve alors les deux solutions précédentes pour typer la troisième version de notre fonction de rotation. Cette opération de clonage n'existe pas en Java. Chacune de ces solutions pour résoudre le problème du sous-typage pose des problèmes pour calculer automatiquement le type (inférence de type en Ocaml).

3.10 Notation objet et liaison tardive

Reprenons l'exemple des points colorés en utilisant des méthodes non statiques et en utilisant la notation objet pour l'invocation des fonctions de translation et de rotation :

```
class Point {
  double x, y;
  Point () { }
  Point (double x0, double y0) { x = x0; y = 0; }

  Point translation (double dx, double dy) {
    x = x + dx; y = y + dy; return this;
  }

  Point rotation (double theta) {
    double x1 = x * Math.cos(theta) - y * Math.sin(theta);
    double y1 = x * Math.sin(theta) + y * Math.cos(theta);
    x = x1; y = y1;
    return this;
} }

class PointC extends Point {
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC (double x0, double y0, int col) { x = x0; y = y0; c = col;}

  public static void main (String[ ] args) {
    PointC p = new PointC(3, 4, JAUNE) ;
    p = (PointC) p.rotation(Math.PI);
} }
```

Pour les points colorés, la notation objet n'est pas vraiment nécessaire. Certains aiment l'utiliser pour son aspect compact. Mais l'intérêt de cette écriture réside dans la notion de liaison *tardive*. En effet, lorsqu'on écrit $o.f$ pour appeler la méthode f de l'objet o, on ne fixe pas vraiment la classe où f est défini (contrairement à la notation utilisée pour les méthodes statiques). La règle est d'invoquer la méthode f de la classe la plus spécifique (c'est-à-dire la plus petite) dont l'objet o est une instance. Prenons par exemple le cas de la méthode *toString* de transformation en chaîne de caractères, utilisée dans les fonctions d'impression. En Java, par convention,

```
System.out.println (p);
```

est toujours équivalent à

```
System.out.println (p.toString());
```

sauf si p est déjà une chaîne. Il y a donc un appel de la méthode *toString* pour effectuer l'impression de p. Cette méthode est déjà définie dans la classe *Object*. Mais, on veut la redéfinir dans toutes les classes pour obtenir un format d'impression compréhensible, et notamment dans notre cas des points possiblement colorés :

```
class Point {
  double x, y;
  ...
  public String toString() { return "(" + x + ", " + y + ")"; }
}

class PointC extends Point {
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  ...
  public String toString() { return "(" + x + ", " + y + ", " + c + ")"; }

  public static void main (String[ ] args) {
    PointC p = new PointC(3, 4, JAUNE) ;
    System.out.println (p);
    Point q = new Point (1, 8) ;
    System.out.println (q);
} }
```

Les implémenteurs de Java ont écrit le code de la fonction *System.out.println* sans connaître la définition de la méthode *toString* pour les points, éventuellement colorés. La notation objet permet de faire cette liaison tardive. Les deux appels de *println* invoquent deux méthodes *toString* différentes, puisque, pour p, on se sert de la méthode des points colorés et pour q de celle des points non colorés. Remarquons qu'on pourait se dispenser de la liaison tardive en passant en argument de *println* la fonction *toString*, mais cela est impossible dans un langage non fonctionnel comme Java. L'écriture compacte de la liaison tardive a ses adeptes, mais il faut remarquer que cela entraine des difficultés de lisibilité du code, par exemple pour retrouver la fonction véritablement invoquée au milieu d'une grande hiérarchie de classes.

En résumé, la programmation par objets implique une forme de modularité, puisqu'on est amené à regrouper dans un objet les données et les fonctions dont les sens sont proches. Mais la programmation objet repose aussi sur une notion de programmation incrémentale, organisée autour de la représentation des données. Une sous-classe contient au moins tous les champs de sa classe parente, et un certain nombre de nouveaux champs. Une sous-classe peut aussi redéfinir (spécialiser) certains champs de sa classe parente. La notation objet pour l'invocation des méthodes utilise la notion d'héritage entre les classes pour retrouver la fonction véritablement appelée, puisqu'on applique toujours la fonction la plus spécifique dans la hiérarchie des classes. La notation objet permet d'effectuer une liaison tardive entre l'invocation d'une méthode et sa réalisation effective, puisqu'on peut l'invoquer sans connaitre sa définition.

3.11 Droits d'accès, polymorphisme, surcharge et héritage

La hiérarchie des classes intervient aussi dans les droits d'accès aux champs des classes. Un quatrième type d'accès, l'accès protégé, permet de restreindre l'accès aux seules sous-classes. Globalement, en Java, il y a donc quatre types d'accès pour les membres d'une classe :

- public en préfixant un champ par *public* pour permettre l'accès depuis toutes les classes,
- privé en préfixant un champ par *private* pour restreindre l'accès aux seules expressions ou fonctions dans la classe courante,
- amical en ne mettant aucun préfixe devant la déclaration d'un champ pour autoriser l'accès depuis toutes les classes du même paquetage (c'est donc l'accès par défaut),
- protégé en préfixant un champ par *protected* pour permettre l'accès aux seules expressions ou fonctions dans les sous-classes de la classe courante.

Les méthodes redéfinies dans les sous-classes doivent avoir la même signature (arguments et résultat) que celles des fonctions redéfinies. Les méthodes redéfinies doivent également fournir au moins les mêmes droits d'accès. Ainsi une méthode publique (par exemple *toString* de la classe *Object*) ne peut être redéfinie en méthode privée dans la classe *Point*.

Exercice 18 Donner un moyen de fournir un accès public aux objets d'une classe, tout en contrôlant leurs créations, c'est-à-dire en interdisant les accès directs aux constructeurs.

Le polymorphisme est la propriété de pouvoir appliquer la même fonction a plusieurs types de données. Le code reste le même, seuls les arguments et résultats peuvent changer. L'exemple typique est celui de la fonction *append* sur les listes, qui concatène deux listes. Le code est le même quel que soit le type de chacun des éléments de liste. En Java, il n'y a pas de polymorphisme. Il est donc impossible d'écrire *append* sans conversions explicites.

La surcharge est la propriété de donner un même nom à des fonctions différentes (donc de code différent). Selon le type des arguments, on appelera l'une de ces fonctions. En Java, la surcharge est partout présente, par exemple pour avoir plusieurs constructeurs dans une même classe. La surcharge est résolue *statiquement*, à la compilation du programme.

L'héritage est la propriété de trouver *dynamiquement* la méthode s'appliquant à un objet (par exemple pour *toString*). Cette opération demande un temps constant quand il y a héritage simple. Le mélange de la surcharge et de l'héritage peut arriver à des situations compliquées.

Exercice 19 Dans l'exemple suivant, quelle est la valeur imprimée ?

```
class C {
  void f() { g(); }
  void g() { System.out.println(1); }
}
```

```
class D extends C {
  void g() { System.out.println(2); }

  public static void main (String[ ] args) {
    D o = new D();
    o.f();
} }
```

Un autre exemple est le suivant, où on a rajouté deux méthodes *equals* pour tester l'égalité des points et des points colorés.

```
class Point {
  ...
  public boolean equals (Point p) {
    return x == p.x && y == p.y;
  }
}

class PointC extends Point {
  ...
  public boolean equals (PointC p) {
    return x == p.x && y == p.y && c == p.c;
  }

  static boolean f (Point p, Point q) {
    return p.equals (q);
  }

  public static void main (String[ ] args) {
    PointC p = new PointC (3, 4, JAUNE) ;
    PointC q = new PointC (3, 4, ROUGE) ;
    System.out.println(f(p, q));
  }
}
```

Le résultat est vrai. En effet, comme la surcharge de la méthode *equals* est résolue statiquement dans le code de f, on n'a comme seule information disponible sur q qu'il est un point normal. On cherche donc une méthode *equals* prenant un argument de type *Point* dans les sous-classes possibles pour l'argument p de f (On sait qu'une telle méthode existera toujours puisqu'il y a déjà cette méthode dans la classe *Point* de p). Or, lors de l'appel de f dans *main*, alors p est un point coloré. Dans la classe *PointC* de p, il n'existe pas de méthode *equals* prenant un argument de type *Point* ; on appelle donc la méthode de sa surclasse *Point*. Le résultat est donc vrai, puisque cette méthode ne teste pas l'argument couleur. Certains peuvent trouver ce raisonnement logique ; d'autres peuvent s'inquiéter de son aspect peu intuitif.

En fait, si on veut que *p.equals(q)* spécialise simultanément p et q pour appeler la méthode *equals* des points colorés lorsque p et q sont deux points colorés, on doit résoudre le problème non trivial dit des méthodes binaires. C'est impossible en Java ; dans d'autres langages, on y arrive en utilisant un argument de type désignant l'objet lui-même. En Java, il n'y a pas de type spécifique pour *this*. Cette notion fine de la programmation étend le pouvoir expressif d'un langage de programmation (par exemple c'est le cas en Ocaml). En Java, le problème disparait, mais la puissance d'expression du langage est moins grande.

3.12 Classes abstraites et types disjonctifs

Un classe abstraite de Java est une classe dont toutes les méthodes ne sont pas définies. Pour celles-ci, on ne précise que leur signature. Les classes abstraites s'appellent aussi classes virtuelles dans d'autres langages de programmation.

Les types disjonctifs sont un cas d'utilisation des classes abstraites dans la programmation par objets. Un type disjonctif est la somme disjointe de plusieurs types. Prenons l'exemple des arbres de syntaxe abstraite (ASA) pour les expressions arithmétiques, définis dans la section 2.6 du chapitre précédent. Ces ASA peuvent être définis récursivement (avec la notation BNF) comme l'ensemble T suivant :

$$\begin{array}{lcl} T & := & x \mid n \mid T + T \mid T - T \mid T \times T \mid T \;/\; T \\ x & := & \text{variable} \\ n & := & \text{constante entière} \end{array}$$

C'est la somme (disjointe) de 6 cas : constante, variable, addition, soustraction, multiplication et division. Dans la classe *Terme* correspondante (cf. le chapitre précédent), un champ *nature* permettait de distinguer entre les 6 cas. En style orienté objets, on peut définir 6 sous-classes d'une classe générique de la manière suivante :

```
abstract class Terme {
  abstract int eval (Environnement e);
  abstract public String toString ();
}

class Var extends Terme {
  String nom;
  Var (String s) { nom = s; }
  int eval (Environnement e) { return Environnement.assoc(nom, e); }
  public String toString () { return nom; }
}

class Const extends Terme {
  int valeur;
  Const (int n) { valeur = n; }
  int eval (Environnement e) { return valeur; }
  public String toString () { return valeur + ""; }
}

class Add extends Terme {
  Terme a1, a2;
  Add (Terme x, Terme y) {a1 = x; a2 = y; }
  int eval (Environnement e) { return a1.eval(e) + a2.eval(e); }
  public String toString () { return "(" + a1 + " + " + a2 + ")"; }
}

class Soust extends Terme {
  Terme a1, a2;
  Soust (Terme x, Terme y) {a1 = x; a2 = y; }
  int eval (Environnement e) { return a1.eval(e) - a2.eval(e); }
  public String toString () { return "(" + a1 + " - " + a2 + ")"; }
}

class Mul extends Terme {
  Terme a1, a2;
  Mul (Terme x, Terme y) {a1 = x; a2 = y; }
  int eval (Environnement e) { return a1.eval(e) * a2.eval(e); }
```

```
    public String toString () { return "(" + a1 + " * " + a2 + ")"; }
  }

  class Div extends Terme {
    Terme a1, a2;
    Div (Terme x, Terme y) {a1 = x; a2 = y; }
    int eval (Environnement e) { return a1.eval(e) / a2.eval(e); }
    public String toString () { return "(" + a1 + " / " + a2 + ")"; }
  }

  class Test {
      public static void main (String[ ] args) {
      Terme t = ... ;
      System.out.println (t);
      Environnement e = ... ;
      System.out.println (t.eval(e));
    }
  }
```

où la classe *Environnement* est la classe de la section 2.12.1 manipulant des listes d'association entre les noms et valeurs des variables. La classe générique *Terme* est abstraite, car tous ses champs ne sont pas définis. Comme les champs *eval* et *toString* sont abstraits, chacune des 6 sous-classes (non abstraites) devra les définir. Tout terme t aura donc deux méthodes *toString* et *eval*, pour l'imprimer et pour l'évaluer dans un environnement donné.

Considérons l'expression $(x + 1) \times (3 \times y + 1)$ et son arbre de syntaxe abstraite t. Celui-ci est construit par :

```
Terme t = new Mul (
            new Add (new Var ("x"), new Const (1)),
            new Add(
              new Mul (new Const (3), new Var ("y")),
              new Const (1)));
```

Alors $t.eval(e)$ retourne la valeur 2002 dans l'environnement e tel que $x = 90$ et $y = 7$. En fait l'évaluation commence par appliquer la méthode *eval* de la classe *Mul*, qui est la plus petite classe contenant t ; celle-ci appelle la méthode *eval* sur son premier argument, puis sur son deuxième argument et multiplie les deux résultats. A nouveau l'appel d'*eval* sur le premier argument engendre un appel de la méthode de la sous-classe *Add* puisque c'est la plus petite contenant contenant $t.a1$, etc. C'est donc la hiérarchie des classes qui contrôle les appels récursifs de la fonction *eval*. Les cas de base de la récursion correspondent aux sous-classes *Var* et *Const*, où on retourne la valeur trouvée dans l'environnement ou la valeur stockée pour la constante dans le champ *valeur*. A la différence de la fonction définie dans le style procédural à la section 2.12.1, il n'y a pas besoin d'un champ (*nature*) indiquant le sous-cas dans la somme disjointe des types. Ici, on utilise les seuls champs nécessaires dans chacun des sous-cas, plutôt que la réunion de tous les champs comme dans la version procédurale.

Le cas de la méthode *toString* a été déjà vu dans l'exemple des points colorés. Ici, chaque sous-classe définit sa fonction de transformation en chaîne de caractères, en concaténant les résultats obtenus récursivement pour les sous-termes. Il n'est pas besoin de faire l'appel explicite à *a1.toString* ou *a2.toString*, car, comme pour les fonctions d'impression, cet appel est généré implicitement par l'opérateur de concaténation `+`.

L'exemple des ASA permet de bien comprendre la différence entre programmation procédurale et programmation par objets. La version orientée objet est complètement organisée autour du type des données. Les fonctions d'évaluation et d'impression se retrouvent dispersées dans les données. Si on veut, par exemple, rajouter l'opération moins unaire qui calcule l'opposé de tout terme, il suffira de rajouter une nouvelle classe avec ses propres méthodes sans rien changer au reste, alors que, dans la version procédurale de la section 2.12.1, il faudra faire de multiples changements. Inversement, si on rajoute une nouvelle fonction, dans la version orientée-objet, il faudra effectuer un changement dans chacune des sous-classes, alors que ce changement ne se fera qu'à un seul endroit dans la version procédurale. Choisir entre les deux styles de programmation est souvent affaire de goût.

Il faut enfin mentionner que certains langages de programmation existent avec des opérateurs explicites de somme disjointe sur types. Ces opérateurs peuvent être plus ou moins sophistiqués, de Pascal à ML. Mais il n'évitent pas le choix entre style procédural et style par objets.

Exercice 20 Ecrire la méthode *equals* qui teste l'égalité (structurelle) de deux termes.

Exercice 21 Calculer la dérivée d'un terme arithmétique en orienté-objet.

Exercice 22 Ecrire la méthode de belle-impression de termes arithmétiques en orienté-objet.

Chapitre 4
Exploration

Souvent, on recherche un ensemble d'éléments satisfaisant une certaine contrainte parmi un ensemble fini donné. Par exemple, on cherche parmi les sommets d'un graphe les sommets figurant sur un chemin reliant deux sommets donnés, ou on cherche parmi tous les mouvements possibles d'un cavalier sur un échiquier ceux qui permettent de parcourir tout l'échiquier à partir d'une position donneé, ou on cherche parmi tous les pavages possibles ceux qui sont légaux avec des dominos de Wang (carrés à bord colorés dont les couleurs de bords adjacents doivent coïncider) ou avec des polyminos (pentaminos, hexaminos, etc) donnés. Parfois, on complique le problème en introduisant une notion de coût, certaines solutions étant meilleures que d'autres, et on recherche alors la solution de meilleur coût. D'un point de vue abstrait, on peut énoncer le problème sous la forme suivante :

> Soit E un ensemble fini. A chaque élément e de E, on affecte une valeur $v(e)$ (en général, un entier positif), on se donne de plus un prédicat (une fonction à valeurs booléennes) C sur l'ensemble des parties de E. Le problème consiste à construire un sous ensemble F de E tel que :
> - $C(F)$ est satisfait
> - $\sum_{e \in F} v(e)$ est maximal (ou minimal, dans certains cas)

Les méthodes développées pour résoudre ces problèmes sont de natures très diverses. Certaines font intervenir la spécificité du problème (par exemple la théorie des graphes pour trouver des chemins reliant deux sommets donnés dans un graphe). Ici nous ne considérerons que des méthodes générales, consistant à explorer tous les sous-ensembles de E, parfois avec des heuristiques particulières. Nous les rangerons en trois catégories : les méthodes gloutonnes, la programmation dynamique et les techniques d'exploration en force brute, encore appelée *exploration arborescente* (ou *backtracking* en anglais). L"exploration peut aussi être considérée comme un cas particulier de l'optimisation, quand la notion de coût intervient. Alors on est souvent face à un problème très célèbre et très étudié de l'informatique : la *NP-complétude*, que nous ne ferons qu'aborder dans ce chapitre.

4.1 Algorithmes gloutons

La technique gloutonne est une des plus rapides méthodes d'exploration. Elle consiste à choisir la solution parmi l'espace de celles possibles en fonction de critères purement locaux, sans besoin d'aucun retour en arrière (*backtracking*). Cela ne produit pas toujours la solution optimale. Mais, dans les trois cas considérés dans cette section, cela fonctionne très bien.

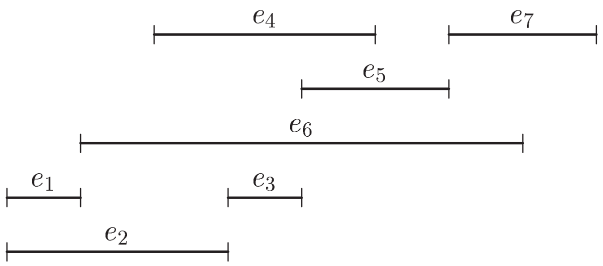


FIG. 4.1 – Un planning de réservation pour les locations d'une voiture

4.1.1 Affectation d'une ressource

Le problème décrit précisément ci-dessous peut être résolu par l'algorithme glouton (mais, comme on le verra, l'algorithme glouton ne donne pas la solution optimale pour une autre formulation du problème, pourtant proche de celle-ci). Il s'agit d'affecter une ressource unique, non partageable, successivement à un certain nombre d'utilisateurs qui en font la demande en précisant la période exacte pendant laquelle ils souhaitent en disposer.

On peut matérialiser ceci en prenant pour illustration la location d'une seule voiture. Des clients formulent un ensemble de demandes de location et, pour chaque demande, sont donnés le jour du début de la location et le jour de restitution du véhicule. Le but est d'affecter le véhicule de façon à satisfaire le maximum de clients (et non pas de maximiser la somme des durées de location). On peut formuler ce problème en utilisant le cadre général considéré plus haut. L'ensemble E est celui des demandes de location, pour chaque élément e de E, on note $d(e)$ et $f(e)$ les dates de début et de fin de location ($d(e) < f(e)$). La valeur $v(e)$ de tout élément e de E est égale à 1 et la contrainte à respecter pour le sous-ensemble F à construire est la suivante :

$$\forall e_1, e_2 \in F \quad d(e_1) \leq d(e_2) \Rightarrow f(e_1) \leq d(e_2)$$

puisque, disposant d'un seul véhicule, on ne peut le louer qu'à un seul client à la fois. Sur la figure 4.1, la solution optimale consiste à prendre les locations e_1, e_3, e_5, e_7. L'algorithme glouton suivant résoud ce problème :

- *Etape 1 :* Trier les éléments de E par ordre des dates de fins. Les éléments de E forment une suite $\langle e_1, e_2, \ldots e_n \rangle$ telle que $f(e_1) \leq f(e_2), \ldots \leq f(e_n)$.
- *Etape 2 :* Initialiser $F = \emptyset$.
- *Etape 3 :* Pour i variant de 1 à n, ajouter la demande e_i à F si celle-ci ne chevauche pas la dernière demande appartenant à F.

Montrons que l'on obtient bien ainsi une solution optimale. Soit $F = \{x_1, x_2, \ldots x_p\}$ la solution obtenue par l'algorithme glouton et soit $G = \{y_1, y_2, \ldots y_q\}$ ($q \leq p$) une solution optimale. Dans les deux cas, nous supposons que les demandes sont classées par dates de fins croissantes. Montrons que $p = q$. Prenons le premier x_k différent de y_k, c'est-à-dire $x_i = y_i$ pour $1 \leq i < k$ et $x_k \neq y_k$. Alors, par construction de F, on a $f(x_k) \leq f(y_k)$. Et, on peut remplacer G par $G' = \{y_1, y_2, \ldots y_{k-1}, x_k, y_{k+1}, \ldots y_q\}$ tout en satisfaisant à la contrainte de non chevauchement des demandes. Ainsi G', de même cardinalité que G, est aussi une solution optimale, ayant plus d'éléments en commun avec F que n'en avait G. En répétant cette opération, on trouve un ensemble H optimal qui contient F. Comme H est optimal, on a $H = F$, et donc $p = q$.

Le programme correspondant est donné ci-dessous. Les tableaux d et f sont donnés en arguments de la fonction *location*. Pour chaque indice i, ils produisent les dates $d(e_i)$ et $f(e_i)$. On supposera que f est déjà trié en ordre croissant. Le résultat est un tableau de booléens indiquant pour tout i si e_i fait partie de la solution optimale.

```
static boolean[ ] location (int[ ] d, int[ ] f) {
  int n = d.length;
  boolean[ ] r = new boolean[n];
  int dernier = -1;
  for (int i = 0; i < n; ++i) {
    r[i] = dernier == -1 || f[dernier] <= d[i];
    dernier = r[i] ? i : dernier;
  }
  return r;
}
```

Si le but est de maximiser la durée totale de location du véhicule, l'algorithme glouton ne donne pas l'optimum. En particulier, il ne considérera pas comme prioritaire une demande de location de durée très importante. L'idée est alors de classer les demandes par durées décroissantes et d'appliquer l'algorithme glouton, malheureusement cette technique ne donne pas non plus le bon résultat (il suffit de considérer une demande de location de 3 jours et deux demandes qui ne se chevauchent pas mais qui sont incompatibles avec la première chacune de durée égale à 2 jours). Le problème de la maximisation de cette durée totale est NP-complet, il est illusoire de penser en trouver un algorithme simple et efficace.

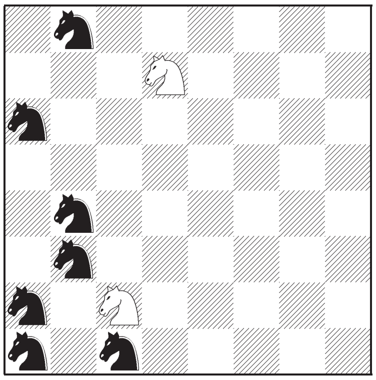
Exercice 23 Montrer que, si on dispose de deux voitures à louer, l'algorithme glouton précédent, triant sur les dates de fin, ne donne pas la solution optimale.

4.1.2 La marche du cavalier

Dans le jeu d'échecs, les mouvements du cheval sont un peu bizarres. Un problème classique consiste à parcourir toutes les cases d'un échiquier avec un cheval à partir d'une position initiale donnée sans repasser deux fois par une même case. On peut montrer que c'est toujours possible (sur un échiquier standard de 64 cases). Une technique gloutonne permet d'écrire un programme marchant sur toutes les cases de départ, sauf une.

On démarre sur la position de départ donnée (`c2` sur la figure 4.2 marquée par un cheval blanc). A toute étape, on choisit d'aller sur la case où, le coup suivant, on a le moins de coups possibles (un coup n'est possible que s'il ne passe pas par une case déjà rencontrée). Sur la figure, les 9 premiers coups sont montrés sur la partie gauche ; la suite de coups est indiquée sur la partie droite, le coup 0 est en `c2`, le coup 1 en `a1`, le coup 2 en `b3`, etc. Au 10ème coup, on choisit d'aller en `f8` où moins de coups sont possibles qu'en `f6`. Dans cette marche du cavalier, le choix du mouvement suivant ne prend en compte que le nombre de coups réalisables pour le coup qui suit immédiatement ; c'est donc une technique gloutonne.

Ceci aboutit au programme suivant. La fonction *marche* prend en paramètre la position de départ indiquée par ses coordonnées cartésiennes i et j ($0 \le i < 8$ et $0 \le j < 8$). La fonction modifie le tableau m qui donne la suite des mouvements (comme sur la partie droite de la figure 4.2). On suppose ce tableau initialisé à la valeur négative *LIBRE*. Pour calculer les nombres de coups possibles, on utilise une fonction



22	7	58	61	24	9	28	53
59	62	23	8	51	54	25	10
6	21	60	57	40	27	52	29
63	36	39	50	55	48	11	26
20	5	56	41	38	45	30	47
35	2	37	44	49	42	15	12
4	19	0	33	14	17	46	31
1	34	3	18	43	32	13	16

FIG. 4.2 – Marche du cavalier sur un échiquier

auxiliaire *jouable* testant si une position donnée par ses coordonnées cartésiennes est une position libre à l'intérieur de l'échiquier.

```
final static int LIBRE = -1;
final static int[ ] x = {2, 1, -1, -2, -2, -1, 1, 2};
final static int[ ] y = {1, 2, 2, 1, -1, -2, -2, -1};

static void marche (int[ ][ ] m, int i, int j) {
  int coup = -1; int i0, j0;
  do {
    m[i][j] = ++coup;
    i0 = i; j0 = j;
    int min = Integer.MAX_VALUE;
    for (int d = 0; d < x.length; ++d) {
      int n = nbDeCoups (m, i0+x[d], j0+y[d]);
      if (n < min) {
        i = i0+x[d]; j = j0+y[d];
        min = n;
      }
    }
  } while (i != i0 || j != j0);
}

static boolean jouable (int[ ][ ] m, int i, int j) {
  return 0 <= i && i < m.length && 0 <= j && j < m[0].length
         && m[i][j] == LIBRE;
}

static int nbDeCoups (int[ ][ ] m, int i, int j) {
  if ( !jouable (m, i, j) )
    return Integer.MAX_VALUE;
  else {
    int r = 0;
    for (int d = 0; d < x.length; ++d)
      if ( jouable (m, i+x[d], j+y[d]) )
        ++r;
    return r;
  }
}
```

Exercice 24 Montrer que ce programme fonctionne pour les 64 valeurs de la position de départ, sauf une. Trouver la position de départ problématique. Modifier le programme pour qu'il fonctionne aussi avec cette position de départ.

Cet algorithme ne marche plus pour de grands échiquiers (76×76). D'autres algorithmes fonctionnent alors par décomposition de l'échiquier en échiquiers plus petits.

4.1.3 Arbre recouvrant de poids minimal

Un exemple classique d'utilisation de l'algorithme glouton est la recherche d'un arbre recouvrant de poids minimal dans un graphe non dirigé. Plusieurs algorithmes existent : Kruskal, Prim ; nous considérons le premier. Ce problème se pose dans la construction de réseaux hydrauliques, électriques ou informatiques. Le graphe correspond alors aux plans d'une ville, d'une maison ou d'un système de relais. Il s'agit de relier des maisons, des pièces ou des sites avec un réseau de coût minimal.

Formellement, on se donne un graphe valué $G = (X, A, p)$ non-orienté et connexe. Pour chaque arc a de A, une fonction de valuation à valeur dans les entiers naturels donne son poids $p(a)$. Comme auparavant, nous représentons le graphe non-orienté par un graphe orienté et symétrique (pour tout $a \in A$, il existe un arc opposé $\bar{a}$ dont l'origine est l'extrémité de a et dont l'extrémité est l'origine de a). La paire $(a, \bar{a})$ est une arête. Chaque arête a un poids, et donc a et $\bar{a}$ ont même poids, c'est-à-dire $p(a) = p(\bar{a})$ pour $a \in A$. On cherche un arbre recouvrant de poids minimal, c'est à dire un arbre dont la somme des poids des arcs est minimale.

On peut facilement formuler le problème dans le cadre général donné en début de chapitre. On prend pour E l'ensemble des arcs du graphe ; la condition C à satisfaire par F est de former un graphe connexe ; enfin il faut minimiser la somme des poids des éléments de F. Ce problème peut être résolu très efficacement par l'algorithme glouton suivant :

- *Etape 1 :* Classer les arcs par ordre de poids croissants. Ils forment alors une suite $e_1, e_2, \dots e_n$ telle que $p(e_1) \leq p(e_2), \dots \leq p(e_n)$.
- *Etape 2 :* Initialiser $F = \emptyset$
- *Etape 3 :* Pour i variant de 1 à n, ajouter l'arête e_i à F si celle-ci ne crée pas de circuit avec celles appartenant à F.

On montre que l'algorithme glouton donne l'arbre de poids minimal en utilisant la propriété suivante des arbres recouvrants d'un graphe.

Proposition 4.1 *Soient T et U deux arbres recouvrants distincts d'un graphe G et soit a une arête de U qui n'est pas dans T. Alors il existe une arête b de T telle que $U \setminus \{a\} \cup \{b\}$ soit aussi un arbre recouvrant de G.*

Plus généralement on montre que l'algorithme glouton donne le résultat si et seulement si la propriété suivante est vérifiée par les sous ensembles F de E satisfaisant C :

Proposition 4.2 *Si F et G sont deux ensembles qui satisfont la condition C et si x est un élément qui est dans F et qui n'est pas dans G, alors il existe un élément de G tel que $F \setminus \{x\} \cup \{y\}$ satisfasse C.*

Un exemple d'arbre recouvrant de poids minimal est donné sur la figure 4.3.

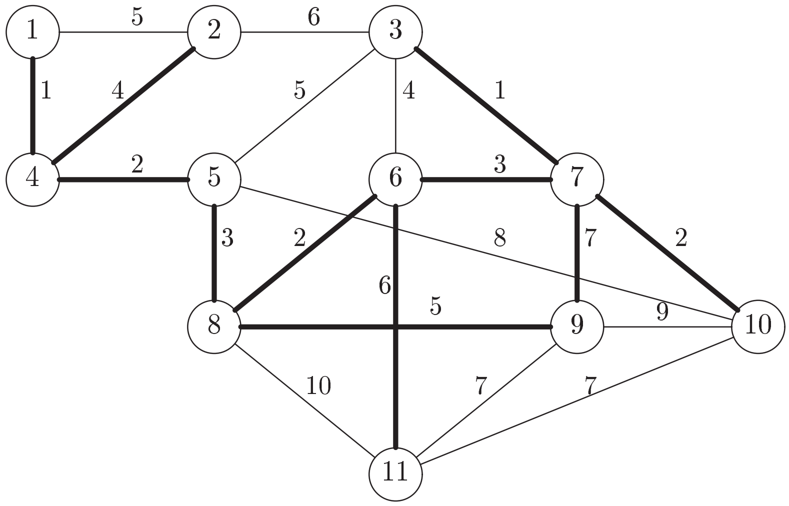


FIG. 4.3 – Un arbre recouvrant de poids minimum

4.2 Exploration arborescente

De très nombreux problèmes d'optimisation ou de recherche de configurations particulières donnent lieu à un algorithme qui consiste à faire une recherche exhaustive des solutions. Ces algorithmes paraissent simples puisqu'il s'agit de parcourir systématiquement un ensemble de solutions, mais bien que leur principe ne soit pas particulièrement ingénieux, la programmation nécessite un certain soin.

4.2.1 Sac à dos

Soit E un ensemble d'objets e_i ayant chacun un certain poids $p(e_i)$ ($p(e_i) \in \mathbf{R}^+$). Soit M la charge maximum que l'on peut emporter dans le sac à dos ($M \in \mathbf{N}$). On veut trouver un ensemble d'objets dont la somme des poids soit la plus voisine possible de M tout en lui étant inférieure ou égale. Le problème est ici formulé dans les termes généraux du début du chapitre, la condition C portant sur le poids du sac à ne pas dépasser. Souvent la formulation du problème du sac à dos fait intervenir deux fonctions : le poids et la valeur, ou encore le volume et le poids. Il s'agit alors d'avoir la valeur maximale avec une borne supérieure sur le poids, ou de minimiser le poids avec une borne supérieure sur le volume. Mais notre présentation simplifiée comporte le même degré de difficulté. Il faut toutefois bien faire attention à définir les valeurs des poids comme non entiers, car sinon un algorithme simple de programmation dynamique peut être réalisé (à condition que M ne soit pas trop grand).

Le problème du sac à dos est un exemple typique classique de problème (NP-complet) pour lequel aucun algorithme efficace n'est connu et où il faut explorer toutes les possibilités pour obtenir la meilleure solution.

Notons n le nombre d'éléments de E, le tableau de booléens *sac* indique pour tout i ($0 \leq i < n$) si l'objet i est dans le sac. On énumère donc toutes les valeurs possibles de *sac* et on note le meilleur poids total possible ne dépassant pas la borne M. Le flottant *meilleur* mémorise la plus petite valeur trouvée pour la différence entre la capacité du sac et la somme des poids des objets qui s'y trouvent. Le tableau *sacMax* garde en mémoire le contenu du sac qui réalise ce minimum. Les arguments de *calcul* sont l'objet i à partir duquel on doit prendre des décisions, et la capacité u disponible restante.

```
static float meilleur = Float.MAX_VALUE;
static boolean sac = new boolean[n];
static boolean sacMax = new boolean[n];

static void calcul (int i, float u) {
  if (i >= n) {
    if (u < meilleur) {
      for (int j = 0; j < n; ++j)
        sacMax[j] = sac[j];
      meilleur = u;
    }
  } else {
    if  (p[i] <= u) {
      sac[i] = true;
      calcul(i + 1, u - p[i]);
      sac[i] = false;
    }
    calcul(i + 1, u);
  }
}
```

On vérifie sur des exemples que cette fonction donne des résultats assez rapidement pour $n \leq 20$. Pour des valeurs plus grandes le temps mis est bien plus long car il croît comme 2^n.

4.2.2 Les huit reines

Gauss en 1850 a inventé le problème suivant : placer huit reines sur un échiquier sans qu'aucune d'entre elles ne soit en prise par une autre. On peut en fait définir ce problème pour n reines sur un échiquier de taille $n \times n$. Quand on essaie de le résoudre à la main, cela n'a rien d'évident. Deux solutions se trouvent sur la figure 4.4. On peut démontrer qu'il y a toujours une solution pour $n > 3$. Un algorithme simple pour en trouver au moins une fonctionne par recherche exhaustive en parcourant l'ensemble des configurations possibles. Pour les valeurs successives de i ($0 \leq i < 8$), on place une reine sur la ligne i et sur une colonne j ($0 \leq i < 8$) en vérifiant bien qu'elle n'est pas en prise. Le tableau *pos* contient les positions des reines sur chaque ligne. Si une reine figure sur la case de coordonnées i et j, on a $pos[i] = j$. Pour tester si deux reines placées sur les cases i_1, j_1 et i_2, j_2 sont en prise, on regarde si elles sont sur une même ligne ($i_1 = i_2$), ou sur une même colonne ($j_1 = j_2$), ou sur une même diagonale ($|i_1 - i_2| = |j_1 - j_2|$).

La fonction qui trouve les solutions au problème des n reines est alors la suivante :

```
static int[ ] pos = new int[8];

static void reines (int i, int n) {
  if (i < n)
    for (int j = 0; j < n; ++j)
      if (compatible (i, j)) {
        pos[i] = j;
        reines (i+1, n);
      }
  else
    imprimerLaSolution(pos);
}
```

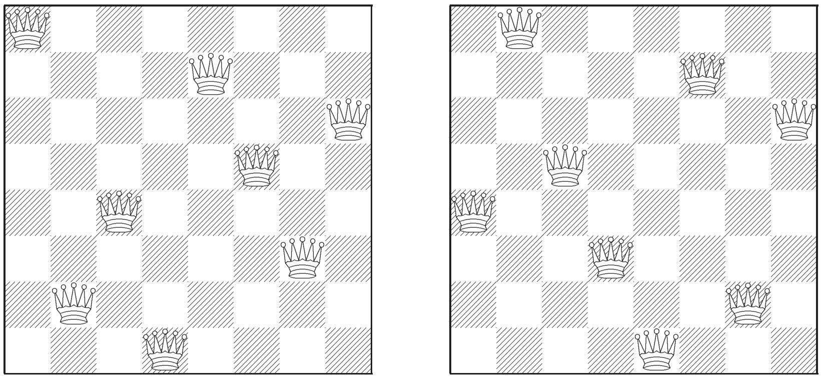


FIG. 4.4 – Huit reines sur un échiquier

```
static boolean compatible (int i1, int j1) {
  for (int i2 = 0; i2 < i1; ++i2) {
    int j2 = pos[i2];
    if (i1 == i2 || j1 == j2 || Math.abs (i1 - i2) == Math.abs (j1 - j2))
      return false;
  }
  return true;
}
```

La boucle à l'intérieur de la fonction *reines* parcourt toutes les positions sur la ligne i compatibles avec les reines déjà placées sur les lignes 0, 1, $\ldots i-1$. Les appels successifs de *reines* modifient la valeur de *pos*. Quand on a parcouru toutes les lignes, on imprime la solution indiquée par les valeurs du tableau *pos*, et on continue pour chercher les solutions suivantes. La fonction *reines* affiche donc toutes les solutions possibles. On peut facilement la modifier pour s'arrêter dès que l'on a trouvé une solution. En lançant *reines(0, 8)*, on trouve ainsi toutes les solutions pour un échiquier 8×8. Sur un ordinateur classique, cela prend quelques minutes. Avec des langages plus sophistiqués comme GNU-Prolog, on arrive à trouver toutes les solutions pour $n = 200$. En fait, il existe une formule analytique donnant une solution pour tout n ($n > 3$).

Exercice 25 Trouver le nombre de solutions au problème des huit reines pour un échiquier de taille 8×8.

Dans les deux exemples donnés plus haut, toute la difficulté réside dans le parcours de toutes les solutions possibles, sans en oublier et sans revenir plusieurs fois sur la même. On peut noter que l'ensemble de ces solutions peut être vu comme les sommets d'une arborescence qu'il faut parcourir. La différence avec les algorithmes décrits au chapitre 5 est que l'on ne représente pas cette arborescence en totalité en mémoire mais simplement la partie sur laquelle on se trouve.

4.3 Programmation dynamique

La programmation dynamique consiste à tabuler des résultats intermédiaires pour éviter de recalculer un résultat déjà rencontré. En outre, elle est souvent associée à des problèmes d'optimisation, comme l'avait énoncé son promoteur R. Bellman. Pour résoudre un problème avec de la programmation dynamique, souvent on le généralise, et il devient un des résultats intermédiaires nécessaires à son propre calcul. La tabulation permet d'en accélérer son calcul. Par exemple, considérons le cas de la fonction de Fibonacci. Récursivement elle est définie par

```
int fib (int n) {
  if (n < 2) return n; else return fib (n-2) + fib (n-1);
}
```

On supprime la duplication des calculs intermédiaires en écrivant :

```
int fib (int n) {
  int[ ] tab = new int[n+1];
  tab[0] = 0; tab[1] = 1;
  for (int i = 2; i <= n; ++i)
    tab[i] = tab[i-2] + tab[i-1];
  return t[n];
}
```

Des esprits chagrins remarqueront que deux valeurs entières suffisent plutot que les n valeurs stockés dans le tableau *tab*, pour gagner 18 cases de la mémoire dans le calcul de *fib(20)* qui fait déjà peiner notre ordinateur ! La grosse différence vient de la vitesse en $O(n)$ au lieu de $O(2^n)$. Le principe de la programmation dynamique est là : les résultats intermédiaires ont été tabulés, en nombre pas trop élevé, mais suffisamment pour obtenir le résultat final. Dans les techniques d'exploration, beaucoup de cas relèvent de la programmation dynamique.

4.3.1 Plus courts chemins dans un graphe

On considère un graphe valué $G = (X, A, \ell)$ ayant X comme ensemble de sommets et A comme ensemble d'arcs. On se donne une application ℓ de A dans les entiers naturels. La valeur $\ell(a)$ est la *longueur* de l'arc a. La longueur d'un chemin est égale à la somme des longueurs des arcs qui le composent. Le problème consiste à déterminer, pour chaque couple (x_i, x_j) de sommets, le plus court chemin, s'il existe, qui joint x_i à x_j.

Nous commençons par trouver les longueurs des plus courts chemins notées $\delta(x_i, x_j)$; on convient de noter $\delta(x_i, x_j) = \infty$ s'il n'existe pas de chemin entre x_i et x_j. La construction effective des chemins sera examinée ensuite. On suppose qu'entre deux sommets il y a au plus un arc. Les algorithmes de recherche de chemins les plus courts reposent sur l'observation suivante :

Proposition 4.3 *Si f est un chemin de longueur minimale joignant x à y et qui passe par z, alors il se décompose en deux chemins de longueur minimale l'un qui joint x à z et l'autre qui joint z à y.*

Dans la suite, on suppose les sommets numérotés $x_1, x_2, \dots x_n$ et, pour tout $k > 0$ on considère la propriété P_k suivante pour un chemin :

> $(P_k(f))$ Tous les sommets de f, autres que son origine et son extrémité, ont un indice strictement inférieur à k.

On peut remarquer qu'un chemin vérifie P_1 si et seulement s'il se compose d'un unique arc, d'autre part la condition P_{n+1} est satisfaite par tous les chemins du graphe. Notons $\delta_k(x_i, x_j)$ la longueur du plus court chemin qui vérifie P_k et qui a pour origine x_i et pour extrémité x_j. Cette valeur est ∞ si aucun tel chemin n'existe. Ainsi $\delta_1(x_i, x_j) = \infty$ s'il n'y a pas d'arc entre x_i et x_j et vaut $\ell(a)$ si a est cet arc. D'autre part $\delta_{n+1} = \delta$.

Le lemme suivant permet de calculer les δ_{k+1} connaissant les $\delta_k(x_i, x_j)$. On en déduira un algorithme itératif.

Proposition 4.4 *Les relations suivantes sont satisfaites par les δ_k :*

$$\delta_{k+1}(x_i, x_j) = min(\delta_k(x_i, x_j), \delta_k(x_i, x_k) + \delta_k(x_k, x_j))$$

Démonstration Soit un chemin de longueur minimale satisfaisant P_{k+1}, ou bien il ne passe pas par x_k et on a $\delta_{k+1}(x_i, x_j) = \delta_k(x_i, x_j)$ ou bien il passe par x_k et, d'après la remarque préliminaire, il est composé d'un chemin de longueur minimale joignant x_i à x_k et satisfaisant P_k et d'un autre minimal aussi joignant x_k à x_j. (On ne peut passer plus d'une fois par x_k puisque les longueurs sont toutes positives). Il a donc pour longueur : $\delta_k(x_i, x_k) + \delta_k(x_k, x_j)$. $\square$

L'algorithme suivant pour la recherche du plus court chemin met à jour une matrice d, valant initialement les longueurs des arcs et ∞ (*Integer.MAX_VALUE*) s'il n'y a pas d'arc entre x_i et x_j. A chaque itération de la boucle externe, on fait croître l'indice k du δ_k calculé.

```
static void distancesMinimales (GrapheMatVal g) {
  int n = g.d.length;
  for (k = 0; k < n; ++k)
    for (i = 0; i < n; ++i)
      for (j = 1; j < n; ++j)
        g.d[i][j] = Math.min(g.d[i][j], g.d[i][k] + g.d[k][j]);
}
```

La classe *GrapheMatVal* ressemble à la classe *GrapheMat* considérée dans le chapitre sur les graphes pour la représentation avec des matrices d'adjacence. Dans cette nouvelle classe, la matrice est à présent une matrice de distances $d = \delta_1$. Dans le programme précédent, on remarque la similitude avec l'algorithme de Warshall pour trouver la fermeture transitive d'un graphe. Sur l'exemple du graphe donné sur la figure 4.5, la matrice de départ δ_1 et le résultat final δ sont :

$$\delta_1 = \begin{pmatrix} 0 & 1 & \infty & 4 & \infty & \infty & \infty \\ \infty & 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & 0 & 2 & \infty & 6 \\ \infty & 3 & \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 2 & \infty & 0 & 1 \\ 4 & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix} \qquad \delta = \begin{pmatrix} 0 & 1 & 4 & 3 & 5 & 6 & 7 \\ 10 & 0 & 3 & 2 & 4 & 5 & 6 \\ 8 & 5 & 0 & 5 & 2 & 3 & 4 \\ 8 & 5 & 8 & 0 & 2 & 3 & 4 \\ 6 & 3 & 6 & 3 & 0 & 1 & 2 \\ 5 & 6 & 9 & 2 & 4 & 0 & 1 \\ 4 & 5 & 8 & 7 & 9 & 10 & 0 \end{pmatrix}$$

Pour produire les chemins les plus courts entre les sommets i et j, on utilise une matrice S qui donne pour tout i et j, le sommet qui suit i dans le chemin le plus court de i à j. Au début, la valeur S_1 de cette matrice contient les éléments $S_1(i, j)$ valant j s'il existe un arc de i à j ou si $i = j$. Sinon ses éléments prennent la valeur indéfinie $\perp$.

Sur l'exemple précédent, on trouve :

$$S_1 = \begin{pmatrix} 0 & 1 & \perp & 3 & \perp & \perp & \perp \\ \perp & 1 & 2 & 3 & \perp & \perp & \perp \\ \perp & \perp & 2 & \perp & 4 & \perp & \perp \\ \perp & \perp & \perp & 3 & 4 & \perp & 6 \\ \perp & 1 & \perp & \perp & 4 & 5 & \perp \\ \perp & \perp & \perp & 3 & \perp & 5 & 6 \\ 0 & \perp & \perp & \perp & \perp & \perp & 6 \end{pmatrix} \qquad S = \begin{pmatrix} 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 2 & 3 & 4 & 4 & 4 & 4 \\ 5 & 5 & 3 & 5 & 5 & 5 & 5 \\ 5 & 5 & 5 & 4 & 5 & 5 & 5 \\ 6 & 2 & 2 & 6 & 5 & 6 & 6 \\ 7 & 7 & 7 & 4 & 4 & 6 & 7 \\ 1 & 1 & 1 & 1 & 1 & 1 & 7 \end{pmatrix}$$

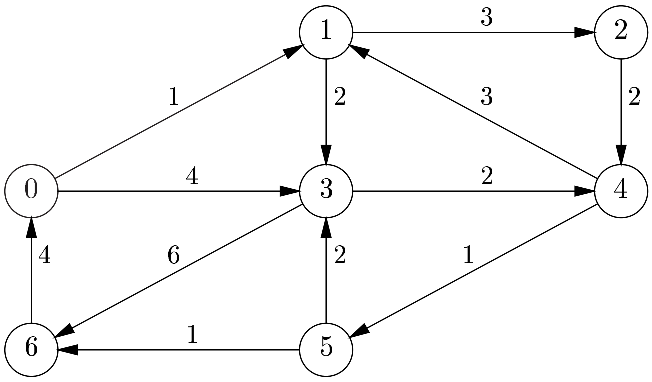


FIG. 4.5 – Un graphe aux arcs valués

Le calcul de la matrice S et donc du chemin le plus court entre deux sommets i et j se font par les fonctions suivantes :

```
final static int INDEFINI = -1;

static int[ ][ ] sommetSuivant (GrapheMatVal g) {
  int n = g.d.length;
  int[ ][ ] s = new int[n][n];
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      s[i][j] = (g.d[i][j] != Integer.MAX_VALUE) ? j : INDEFINI;
  for (k = 0; k < n; ++k)
    for (i = 0; i < n; ++i)
      for (j = 0; j < n; ++j)
        if (g.d[i][j] > (g.d[i][k] + g.d[k][j]) {
          g.d[i][j] = g.d[i][k] + g.d[k][j];
          s[i][j] = s[i][k];
        }
  return s;
}

static void plusCourtChemin (GrapheMatVal g, int i, int j) {
  distancesMinimales (g);
  int[ ][ ] s = sommetSuivant (g);
  for (int k = i; k != j; k = s[k][j])
    if (k == INDEFINI)
      throw new Error ("Chemin inexistant");
    System.out.print (k + " ");
  System.out.println (j);
}
```

4.3.2 Sous-séquences communes

Le problème est de retrouver la plus grande sous-séquence commune à deux chaînes de caractères, ou encore à deux mots sur un alphabet. Ce problème se pose, par exemple, dans la commande *diff* du système Unix, qui affiche les différences « minimales » entre deux fichiers. Dans *diff*, on fait la différence sur les lignes, mais le coeur de la commande utilise un algorithme analogue à celui que nous voyons dans cette section. De même, pour comparer des chaînes d'ADN, on doit aussi retrouver les plus longues séquences communes.

	ϵ	a	a	a	c
ϵ					
b		$\leftarrow$	$\leftarrow$	$\leftarrow$	$\leftarrow$
a		$\nwarrow$	$\nwarrow$	$\nwarrow$	$\leftarrow$
c		$\uparrow$	$\leftarrow$	$\leftarrow$	$\nwarrow$
b		$\uparrow$	$\leftarrow$	$\leftarrow$	$\uparrow$

pred

	ϵ	a	a	a	c
ϵ	0	0	0	0	0
b	0	0	0	0	0
a	0	1	1	1	1
c	0	1	1	1	2
b	0	1	1	1	2

longueur

FIG. 4.6 – Plus longue sous-séquence commune entre *bacb* et *aaac*

Une *séquence* (ou un *mot*) est une suite finie de symboles (ou *lettres*) pris dans un ensemble fini (ou *alphabet*). La longueur de la séquence u est notée $|u|$, et ϵ dénote la séquence vide (de longueur nulle). Une séquence $v = b_1 b_2 \cdots b_m$ est une *sous-séquence* de $u = a_1 a_2 \cdots a_n$ s'il existe des entiers $i_1,i_2, \ldots i_m$ $(0 \leq i_1 < i_2 < \cdots i_m \leq n)$ tels que $a_{i_k} = b_k$ $(1 \leq k \leq m)$. Une séquence w est une *sous-séquence commune* aux séquences u et v si w est sous-séquence de u et de v. Une sous-séquence commune est *maximale* si elle est de longueur maximale.

La plus grande sous-séquence commune $ssc(u,v)$ entre deux séquences u et v est la séquence donnée par la définition récursive suivante :

$$\begin{aligned} ssc(u,\epsilon) &= ssc(\epsilon,u) = \epsilon \\ ssc(ua,va) &= ssc(u,v)a \\ ssc(ua,vb) &= \begin{cases} ssc(ua,v) & \text{si } |ssc(ua,v)| \geq |ssc(u,vb)| \\ ssc(u,vb) & \text{sinon} \end{cases} \end{aligned}$$

En effet, soit w une sous-séquence, de longueur maximale, commune à $a_1 a_2 \cdots a_{i-1}$ et $b_1 b_2 \cdots b_{j-1}$. Si $a_i = b_j$, alors wa_i est une sous-séquence commune maximale à $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$. Si $a_i \neq b_j$, une sous-séquence commune à $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$ est ou bien commune à $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_{j-1}$, ou bien commune à $a_1 a_2 \cdots a_{i-1}$ et $b_1 b_2 \cdots b_j$, en choisissant le cas donnant la sous-séquence de plus grande longueur.

Sur l'exemple de la figure 4.6, cela est illustré dans le cas où $u = bacb$ et $v = aaac$. Le premier mot est représenté verticalement, le deuxième horizontalement. Pour chaque caractère de l'un (à la position i) et de l'autre (à la position j), on obtient la plus longue sous-séquence commune à $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$ en suivant les flèches de la table *pred*. La flèche diagonale correspond à la deuxième ligne des équations vérifiées par *ssc* ; la flèche vers la gauche correspond à la troisième ligne, la flèche vers le haut à la quatrième.

La longueur de la plus grande sous-séquence commune à $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$ est donnée récursivement par :

$$L(i,j) = \begin{cases} 1 + L(i-1,j-1) & \text{si } a_i = b_j \\ \max(L(i,j-1), L(i-1,j)) & \text{sinon} \end{cases}$$

On en déduit les fonctions suivantes pour le calcul de la plus longue sous-séquence commune. Le résultat final se fait en remontant le tableau *pred* à partir de l'extrémité en bas et à droite.

```
final static int GAUCHE = 1, HAUT = 2, DIAG = 3;
```

```
static int longueurSSC (String u, String v, int[ ][ ] pred) {
  int m = u.length(), n = v.length();
  int[ ][ ] longueur = new int[m][n];
  for (int i = 1; i < m + 1; ++i)
    for (int j = 1; j < n + 1; ++j)
      if (u.charAt(i-1) == v.charAt(j-1)) {
        pred[i][j] = DIAG;
        longueur[i][j] = 1 + longueur[i-1][j-1];
      } else if longueur[i][j-1] > longueur[i-1][j] {
        pred[i][j] = GAUCHE;
        longueur[i][j] = longueur[i][j-1];
      } else {
        pred[i][j] = HAUT;
        longueur[i][j] = longueur[i-1][j];
      }
  return longueur[m][n];
}

static String ssc (String u, String v) {
  int m = u.length(), m = v.length();
  int[ ][ ] pred = new int[m][n];
  int lg = longueurSSC (u, v, pred);
  StringBuffer r = new StringBuffer(lg);
  int i = m, j = n;
  for (int k = lg-1; k >= 0; )
      switch (pred[i][j]) {
      case DIAG: r.setCharAt(k, u.charAt(i-1));
         --i; --j; --k; break;
      case GAUCHE: --j; break;
      case HAUT: --i; break;
      }
  return new String(r);
}
```

4.4 Programmes en OCaml

```
(* les n reines, voir page 105 *)

let nReines n =
  let pos = Array.make n 0 in

  let conflit i1 j1 i2 j2 =
    i1 = i2 || j1 = j2 ||
    abs(i1 - i2) = abs (j1 - j2) in

  let compatible i j =
    try
      for k = 0 to i-1 do
        if conflit i j k pos.(k) then
           raise Exit
      done;
      true
    with Exit -> false in

  let rec reines i =
    if i >= n then
      imprimerSolution pos
    else
      for j = 0 to n-1 do
```

```
        if compatible i j then begin
          pos.(i) <- j;
          reines (i+1);
        end
      done in

  reines 0;;
```

```
(* les n reines (suite) *)
open Printf;;

let imprimerSolution pos =
  let n = Array.length(pos) in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if j = pos.(i) then
        printf "* "
      else
        printf "  "
     done;
     printf "\n";
   done;
   printf "----------------------\n";;
```

```
(* les sous séquences communes *)
let longueur_ssc u v =
   let n = String.length u and
       m = String.length v in
   let longueur = Array.make_matrix (n+1) (m+1) 0 and
       provient = Array.make_matrix (n+1) (m+1) 0 in
   for i=1 to n do
     for j=1 to m do
       if u.[i-1] = v.[j-1] then begin
         longueur.(i).(j) <- 1 + longueur.(i-1).(j-1);
         provient.(i).(j) <- 1;
       end else
       if longueur.(i).(j-1) > longueur.(i-1).(j) then begin
         longueur.(i).(j) <- longueur.(i).(j-1);
         provient.(i).(j) <- 2;
       end else begin
         longueur.(i).(j) <- longueur.(i-1).(j);
         provient.(i).(j) <- 3;
       end
     done
   done;
   longueur, provient;;

let ssc u v =
  let n = String.length u and
      m = String.length v in
  let longueur, provient = longueur_ssc u v in
  let lg = longueur.(n).(m) in
  let res = String.create lg and
      i = ref n and
      j = ref m and
      k = ref (lg-1) in
  while !k >= 0 do
    match provient.(!i).(!j) with
      1 -> res.[!k] <- u.[!i-1]; decr i; decr j; decr k
```

```
    | 2 -> decr j
    | _ -> decr i
  done;
  res;;
```

Chapitre 5

Correction

La programmation est une activité complexe, peut-être même une des plus complexes jamais imaginées par l'homme. En mathématiques standards, les variables sont souvent rangées en vecteurs, tableaux ou tenseurs. En informatique, le nombre de variables manipulées par un programme est souvent très grand, les variables sont plutôt indépendantes en dépit de la structuration des données et du regroupement des intructions en fonctions ou en modules. En outre, un programme s'exécute sur un grand nombre de données (ce nombre peut même être infini). Au total, il est délicat de savoir si un programme est correct. La validation des programmes a fait l'objet de nombreuses études et reste un domaine important de la recherche actuelle. On peut chercher à trouver un jeu de tests représentatifs, mais cette méthode n'est pratiquement jamais exhaustive. On peut aussi démontrer mathématiquement que le résultat d'un programme vérifie certaines propriétés. Malheureusement, les démonstrations sur les programmes sont souvent longues et peu intéressantes, à la grande différence des démonstrations des mathématiques, plutôt courtes et conceptuelles. Néanmoins les preuves de programmes sont importantes pour garantir le bon fonctionnement d'un logiciel. C'est pourquoi on cherche à faire les preuves de correction de programmes rigoureusement, c'est-à-dire en utilisant un langage spécial, celui de la logique mathématique, définie par Hilbert, Frege, Gödel et Herbrand. Il n'est pas question ici d'étudier la logique mathématique, qui fait l'objet de longs développements à elle seule. Nous nous contenterons de décrire la méthode des assertions de Floyd et de Hoare.

Dans ce chapitre, nous considérerons la correction de programmes itératifs simples manipulant des scalaires, puis la validation de programmes itératifs manipulant des tableaux, et celle de programmes récursifs avec le principe d'induction. Il sera aussi question de la terminaison des programmes.

5.1 Correction de programmes itératifs scalaires

La suite de Fibonacci définie par $u_0 = 0$, $u_1 = 1$, $u_n = u_{n-2} + u_{n-1}$ pour $n > 1$ est une suite linéaire récurrente d'ordre 2. On peut calculer chacun de ses termes en temps constant, en calculant les racines de l'équation caractéristique associée $r^2 - r - 1 = 0$. Ici nous considérons des calculs moins sophistiqués pour chaque terme u_n qui prennent un temps en $O(n)$.

```
int fib (int n) {
  int[ ] tab = new int[n+1];
  tab[0] = 0; tab[1] = 1;
  for (int i = 2; i <= n; ++i)
    tab[i] = tab[i-2] + tab[i-1];
  return t[n];
}
```

Comme déjà discuté pour la programmation dynamique dans le chapitre 4, le tableau de $n+1$ entiers est inutile, car seulement les deux dernières valeurs x et y suffisent. On peut se contenter de faire le calcul de $\mathit{fib}(n)$ par la fonction suivante :

```
static int fibonacci (int n) {
  int x = 0;
  if (n != 0) {
    x = 1; int y = 0;
    for (int k = 1; k != n; ++k) {
      int t = y;
      y = x;
      x = x + t;
    }
  }
  return x;
}
```

Même si cette optimisation est souvent inutile, car les $n-1$ cases mémoire gagnées sont dérisoires devant la taille des mémoires des ordinateurs modernes, le programme résultant est intéressant, car il a un degré de complexité supplémentaire. Par complexité nous signifions ici la complexité dans son sens naturel, c'est-à-dire la non-simplicité de la solution. En effet, les variables x et y doivent être initialisées, puis modifiées dans le bon ordre de manière à ce qu'aucune d'entre elles ne soit écrasée avant l'utilisation de sa valeur précédente. Nous nous efforçons de montrer rigoureusement que ce programme est correct.

En commentaires, nous plaçons une assertion logique d'entrée $\mathcal{P}$ au début de la fonction et une assertion de fin $\mathcal{R}$ sur le résultat final. Pour que le programme soit correct, il s'agit de montrer que si $\mathcal{P}$ est vérifiée, il en est de même pour l'assertion $\mathcal{Q}$.

```
static int fibonacci (int n) {
  // 𝒫 = {n ≥ 0}
  int x = 0;
  if (n != 0) {
    x = 1; int y = 0;
    for (int k = 1; k != n; ++k) {
      int t = y;
      y = x;
      x = x + t;
    }
  }
  // 𝒬 = {x = fib(n)}
  return x;
}
```

Pour cela, nous allons considérer une troisième assertion $\mathcal{I}$, dite assertion de boucle ou encore *invariant de boucle*, qui sera vraie au début de chaque itération. Pour que l'emplacement de cet invariant dans le programme soit plus clair, l'instruction *for* est décomposée en une instruction d'affectation pour l'initialisation de la variable de contrôle et une instruction d'itération *while*. Graphiquement, l'emplacement de ces assertions est aussi représenté sur l'organigramme de la figure 5.1.

```
static int fibonacci (int n) {
  // 𝒫 = {n ≥ 0}
  int x = 0;
  if (n != 0) {
    x = 1; int y = 0;
    int k = 1;
    // ℐ = {k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
```

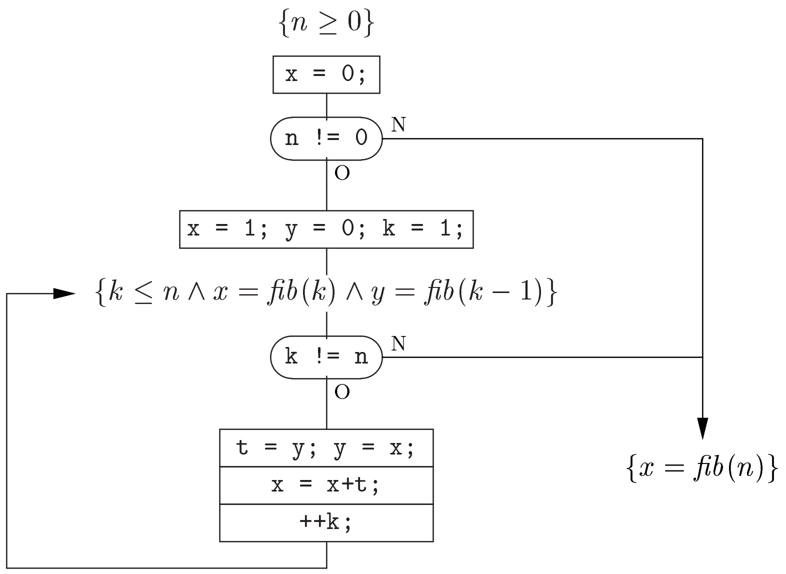


FIG. 5.1 – Organigramme d'un programme avec ses invariants

```
    while (k != n) {
      int t = y;
      y = x;
      x = x + t;
      ++k;
    }
  }
  // Q = {x = fib(n)}
  return x;
}
```

Pour simplifier, nous supprimons les symboles de commentaires et laissons les assertions entre accolades, sans qu'il n'y ait d'ambiguité avec celles utilisées en Java :

```
static int fibonacci (int n) {
  {n ≥ 0}
  int x = 0;
  if (n != 0) {
    x = 1; int y = 0;
    int k = 1;
    {k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
    while (k != n) {
      int t = y;
      y = x;
      x = x + t;
      ++k;
    }
  }
  {x = fib(n)}
  return x;
}
```

Il ne reste plus à montrer que de $\mathcal{P}$ on peut dériver $\mathcal{I}$, que $\mathcal{I}$ est invariant à chaque itération, que de $\mathcal{I}$ on dérive $\mathcal{Q}$ quand on sort de la boucle, et que de $\mathcal{P}$ on dérive $\mathcal{Q}$ quand on ne passe pas dans la boucle. Pour cela, on introduit des assertions intermédiaires à vérifier après chaque instruction élémentaire, et il suffit de montrer que chacune d'entre elles est vraie dès que l'assertion précédant l'instruction en question est vraie.

```
1   static int fibonacci (int n) {
2     {n ≥ 0}
3     int x = 0;
4     {n ≥ 0 ∧ x = 0}
5     if (n != 0) {
6       x = 1; int y = 0;
7      {n > 0 ∧ x = fib(1) ∧ y = fib(0)}
8      int k = 1;
9      {0 < k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
10     while (k != n) {
11       {0 < k < n ∧ x = fib(k) ∧ y = fib(k − 1)}
12       int t = y;
13       {0 < k < n ∧ x = fib(k) ∧ y = fib(k − 1) ∧ t = fib(k − 1)}
14       y = x;
15       {0 < k < n ∧ x = fib(k) ∧ y = fib(k) ∧ t = fib(k − 1)}
16       x = x + t;
17       {0 < k < n ∧ x = fib(k + 1) ∧ y = fib(k) ∧ t = fib(k − 1)}
18       ++k;
19     }
20     {k = n ∧ x = fib(k) ∧ y = fib(k − 1)}
21    }
22    {x = fib(n)}
23    return x;
24  }
```

Notons $\mathcal{P}_n$ l'assertion de la ligne n et $\mathcal{P}_m \vdash_f \mathcal{P}_n$ quand de l'assertion $\mathcal{P}_m$ on peut dériver $\mathcal{P}_n$ dans le code de la fonction f (Ici $f = \mathit{fibonacci}$). On écrira aussi $\mathcal{P}_m \vdash \mathcal{P}_n$ quand f est clair d'après le contexte. Donc, dans le code de la fonction *fibonacci*, on a trivialement $\mathcal{P}_2 \vdash \mathcal{P}_4$. De même $\mathcal{P}_4 \vdash P_7$, puisqu'alors $x = 1 = \mathit{fib}(1)$ et $y = 0 = \mathit{fib}(0)$. Par ailleurs, $n \neq 0$ puisque nous venons de franchir le test de la ligne 5, et $n > 0$ puisque $n \geq 0$ à cause de $\mathcal{P}_4$. Maintenant $\mathcal{P}_7 \vdash \mathcal{P}_9$, puisque, sur la ligne 9, on a alors $k = 1$. Montrons que $P_9 \vdash P_{20}$ quand on sort de la boucle. Alors le test de la ligne 10 est faux, et donc $k = n$, et on a $\mathcal{P}_{20}$. On en déduit $\mathcal{P}_{22}$ puisque $\mathcal{P}_{20} \Rightarrow \mathcal{P}_{22}$. Il reste à montrer que l'invariant de boucle $\mathcal{I} = P_9$ est bien préservé à chaque itération.

D'abord $\mathcal{P}_9 \vdash \mathcal{P}_{11}$, puisque le franchissement du test de la ligne 10 indique $k \neq n$. De même, on a facilement que $\mathcal{P}_{11} \vdash \mathcal{P}_{13}$ et $\mathcal{P}_{13} \vdash \mathcal{P}_{15}$. La preuve que $\mathcal{P}_{15} \vdash \mathcal{P}_{17}$ est un peu plus complexe. En effet, l'exécution de l'instruction de la ligne 16 ne modifie que la valeur de x . Après cette instruction, on a donc $x = \mathit{fib}(k) + \mathit{fib}(k-1)$, puisqu'avant cette instruction, d'après $\mathcal{P}_{15}$, on a $x = \mathit{fib}(k)$ et $t = \mathit{fib}(k-1)$. Comme $0 < k$, on a $\mathit{fib}(k+1) = \mathit{fib}(k) + \mathit{fib}(k-1)$. Donc $x = \mathit{fib}(k+1)$. Montrons finalement que $\mathcal{P}_{17}$ implique l'invariant de boucle $\mathcal{P}_9$. Entre les deux, on ne fait qu'incrémenter la valeur de k. Il suffit donc de remplacer $k+1$ par k dans la formule de $\mathcal{P}_{17}$ pour obtenir $\mathcal{P}_9$. Ceci est obtenu facilement puisque $x = fib(k+1)$ devient $x = \mathit{fib}(k)$, et $y = fib(k+1-1)$ devient $y = \mathit{fib}(k-1)$. De même $0 < k+1-1 < n$ devient $0 < k-1 < n$, c'est-à-dire $0 < k \leq n$.

On peut considérer l'organigramme de la figure 5.1 (complété avec toutes les assertions intermédiaires) comme un graphe dont les sommets sont les assertions et les arcs les instructions. On considère tous les chemins menant de l'assertion d'entrée à l'assertion de sortie. Le nombre de ces chemins peut être infini. Mais, il s'expriment tous par composition de trois chemins simples (c'est-à-dire qui ne passent pas deux fois par un même sommet intermédiaire) : les chemins de l'assertion d'entrée $\mathcal{P}$ à l'invariant de boucle $\mathcal{I}$, de $\mathcal{I}$ à $\mathcal{I}$ et de $\mathcal{I}$ à $\mathcal{Q}$. C'est ce shéma que nous avons suivi précédemment pour démontrer que $\mathcal{P} \vdash \mathcal{Q}$.

Considérons un deuxième exemple en posant des assertions sur le calcul du PGCD.

```
static int pgcd (int a, int b) {
  int x = a, y = b;
  while (x != y)
    if (x > y)
      x = x - y;
    else
      y = y - x;
  return x;
}
```

Il s'agit de montrer que si a et b sont deux entiers strictement positifs, le résultat final est bien le PGCD. On décore le programme précédent avec toutes les assertions intermédiaires.

```
1    static int pgcd (int a, int b) {
2      {a > 0 ∧ b > 0}
3      int x = a, y = b;
4      {x > 0 ∧ y > 0 ∧ pgcd(x, y) = pgcd(a, b)}
5      while (x != y) {
6        {x > 0 ∧ y > 0 ∧ x ≠ y ∧ pgcd(x, y) = pgcd(a, b)}
7        if (x > y) {
8          {x > 0 ∧ y > 0 ∧ x > y ∧ pgcd(x − y, y) = pgcd(a, b)}
9          x = x - y;
10         {x > 0 ∧ y > 0 ∧ pgcd(x, y) = pgcd(a, b)}
11       } else {
12         {x > 0 ∧ y > 0 ∧ x < y ∧ pgcd(x, y − x) = pgcd(a, b)}
13         y = y - x;
14         {x > 0 ∧ y > 0 ∧ pgcd(x, y) = pgcd(a, b)}
15       }
16     }
17     {x > 0 ∧ x = y = pgcd(x, y) = pgcd(a, b)}
18     return x;
19   }
```

A nouveau, $\mathcal{P}_n$ désigne l'assertion de la ligne n. Les preuves de $\mathcal{P}_6 \vdash \mathcal{P}_8$ et de $\mathcal{P}_6 \vdash \mathcal{P}_{12}$ reposent sur des propriétés arithmétiques du PGCD. Pour $\mathcal{P}_8 \vdash \mathcal{P}_{10}$, on remplace simplement $x - y$ par x. De même pour $\mathcal{P}_{12} \vdash \mathcal{P}_{14}$, c'est $y - x$ qui est remplacé par y.

Exercice 26 Montrer que le raisonnement n'est plus valide avec $\{a \geq 0 \wedge b \geq 0\}$ comme assertion d'entrée. Comment corriger le programme ?

Exercice 27 Montrer que l'algorithme (d'Euclide) suivant calcule aussi le PGCD :

```
static int pgcd (int a, int b) {
  int x = a, y = b;
  while (y != 0) {
    int r = x % y;
    x = y;
    y = r;
  }
  return x;
}
```

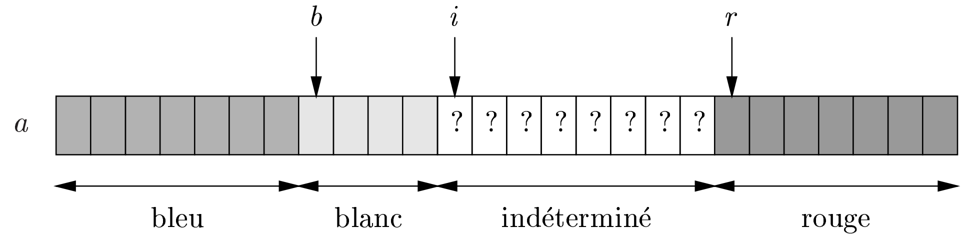


FIG. 5.2 – Le drapeau hollandais

5.2 Correction de programmes itératifs avec des tableaux

La méthode des assertions marche aussi avec des structures de données composites, comme les tableaux. Nous considérons le programme du drapeau hollandais défini par Dijkstra, qui consiste à faire en place un tri à trois valeurs. Au début, un tableau ne contient que des éléments bleus, blancs et rouges. On veut trier le tableau en regroupant d'abord les éléments bleus, puis les blancs et enfin les rouges. Tout cela doit se faire en une seule passe sans mémoire auxiliaire comme indiqué sur la figure 5.2. Pour tout élément a_i, on regarde sa couleur et on le permute avec a_b ou a_r selon les cas. Ce qui donne le programme suivant :

```
static void drapeauHollandais (int[ ] a) {
  int b = 0, i = 0, r = a.length;
  while (i < r) {
    switch (a[i]) {
    case BLEU:
      int t = a[b]; a[b] = a[i]; a[i] = t;
      ++b; ++i;
      break;
    case BLANC:
      ++i;
      break;
    case ROUGE:
      --r;
      int u = a[r]; a[r] = a[i]; a[i] = u;
      break;
    }
  }
}
```

L'invariant de boucle spécifie les zones où la couleur reste constamment bleu, blanc ou rouge, grâce au prédicat ϕ défini par

$$\phi(i, j, c) \ = \ 0 \leq i \leq j \leq n \ \wedge \ \forall k.\, i \leq k < j \Rightarrow a[k] = c$$

Avec ses assertions, le programme est donc comme suit :

```
static void drapeauHollandais (int[ ] a) {
  int b = 0, i = 0, r = a.length, n = r;
  {φ(0, b, BLEU) ∧ φ(b, i, BLANC) ∧ φ(r, n, ROUGE)}
  while (i < r) {
    switch (a[i]) {
    case BLEU:
      {φ(0, b, BLEU) ∧ φ(b, i, BLANC) ∧ φ(r, n, ROUGE) ∧ a[i] = BLEU}
      int t = a[b]; a[b] = a[i]; a[i] = t;
      {φ(0, b + 1, BLEU) ∧ φ(b + 1, i + 1, BLANC) ∧ φ(r, n, ROUGE)}
      ++b; ++i; break;
```

```
      case BLANC:
        {φ(0, b, BLEU) ∧ φ(b, i + 1, BLANC) ∧ φ(r, n, ROUGE)}
        ++i; break;
      case ROUGE:
        {φ(0, b, BLEU) ∧ φ(b, i, BLANC) ∧ φ(r, n, ROUGE) ∧ a[i] = ROUGE}
        --r;
        {φ(0, b, BLEU) ∧ φ(b, i, BLANC) ∧ φ(r + 1, n, ROUGE) ∧ a[i] = ROUGE}
        int u = a[r]; a[r] = a[i]; a[i] = u;
        break;
    } }
    {φ(0, b, BLEU) ∧ φ(b, r, BLANC) ∧ φ(r, n, ROUGE)}
  }
```

Manipuler des tableaux ne produit pas de difficultés supplémentaires dans la méthode des assertions. Il faut raisonner sur les valeurs et les indices des tableaux. On peut aussi remarquer que le programme fonctionne aussi pour le tableau vide ($n = 0$).

Exercice 28 Montrer la correction du programme de tri par sélection.

5.3 Correction partielle et logique de Hoare

On peut tirer un certain nombre de principes généraux des preuves de correction vues sur les exemples précédents. Ces principes seront énoncés de manière intuitive ici, mais ils peuvent être complètement formalisés pour définir les axiomes et les règles d'inférence d'une logique de programmes souvent appelée logique de Hoare.

D'abord, les assertions n'ont de sens que placées par rapport à un programme, puisqu'une assertion peut-être vraie à un endroit du programme, mais fausse à un autre endroit. Les termes de notre logique sont de la forme $\{P\}S\{Q\}$, où P et Q sont deux formules logiques et S un (bout de) programme. Les formules P et Q sont des *pré-conditions* et *post-conditions* de l'instruction S.

Une première remarque consiste à énoncer que pour montrer la post-condition $P(x)$ après une instruction d'affection de x par une expression E, il suffit d'avoir $P(E)$ valide comme pré-condition. Donc un premier axiome de notre logique est :

$$\{P(E)\}\ \texttt{x = E;}\ \{P(x)\}$$

De même, si on veut montrer

$$\{P\}\ \texttt{if (E) S else S'}\ \{Q\}$$

il faut montrer

$$\{P \wedge E\}\ \texttt{S}\ \{Q\} \qquad \text{et} \qquad \{P \wedge \neg E\}\ \texttt{S'}\ \{Q\}$$

Enfin, pour montrer

$$\{P\}\ \texttt{while (E) S}\ \{Q\}$$

il faut deviner l'invariant I, et montrer $P \Rightarrow I$ et $I \wedge \neg E \Rightarrow Q$ et

$$\{I \wedge E\}\ \texttt{S}\ \{I\}$$

Trouver l'invariant de boucle fait partie de la difficulté des preuves de correction ; mais c'est toujours un bon effort pour arriver à la correction du programme ou tout au moins pour le rendre plus robuste.

$$\frac{}{\{P(E)\}\ \texttt{x = E;}\ \{P(x)\}} \qquad \frac{\{P \wedge E\}\ \texttt{S}\ \{Q\} \qquad \{P \wedge \neg E\}\ \texttt{S'}\{Q\}}{\{P\}\ \ \texttt{if (E) S else S'}\ \ \{Q\}}$$

$$\frac{\{P\}\ \texttt{S}\ \{Q\} \quad \{Q\}\ \texttt{S'}\{R\}}{\{P\}\ \ \texttt{S S'}\{R\}} \qquad \frac{\{P\}\ \texttt{S}\ \{Q\}}{\{P\}\ \ \texttt{\{S\}}\ \{Q\}}$$

$$\frac{\{P \wedge E\}\ \texttt{S}\ \{P\}}{\{P\}\ \texttt{while (E) S}\ \{P \wedge \neg E\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ \texttt{S}\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ \texttt{S}\ \{Q\}}$$

FIG. 5.3 – Logique de Floyd-Hoare

D'autres règles existent pour les constructions restantes du langage de programmation. Nous remarquons finalement que la règle pour l'instruction *while* peut être simplifiée, si on ajoute une règle de transitivité. En effet, si P implique P' et si Q' implique Q, alors pour obtenir P et Q comme pré et post-conditions valides de S, il suffit de montrer que P' et Q' sont des pré et post-conditions valides de S. Autrement dit, pour obtenir

$$\{P\}\ \texttt{S}\ \{Q\}$$

il suffit de montrer

$$P \Rightarrow P' \qquad \{P'\}\ \texttt{S}\ \{Q'\} \qquad Q' \Rightarrow Q$$

Traditionnellement les règles de la logique sont présentées avec les prémisses et les conclusions de chaque règle au-dessus et en-dessous d'une ligne horizontale. Pour un axiome, il n'y a aucun prémisse ; l'axiome figure en dessous de la ligne horizontale. Pour une règle d'inférence, les hypothèses figurent au-dessus, la conclusion figure en-dessous. Ainsi, la mini-logique de Hoare que nous venons de voir est résumée par 6 règles sur la figure 5.3 avec concision et précision.

La logique de Hoare manipule des assertions dans les programmes. Ces assertions sont censées être vérifiées à chaque fois que le programme passe par leurs emplacements. Mais rien ne garantit que le programme passe à un emplacement donné. Ainsi quand on écrit $\{P\}S\{Q\}$, on sait que si on démarre l'instruction S avec l'assertion P vraie, alors l'assertion Q est vérifiée si on termine S. Mais on n'a aucune garantie sur la terminaison de S. Donc la correction d'un programme par la méthode des assertions, vue jusqu'à présent, dit que l'assertion de fin est vraie si on démarre avec l'assertion de départ également vraie ; mais il n'est pas sûr que le programme termine. Par exemple

$$\{\mathit{true}\}\ \ \texttt{while (true) S;}\ \{P\}$$

est vraie pour tout P et tout S, puisque jamais P ne sera atteint. Comme on n'a aucune indication sur la terminaison des programmes, on dit que la correction obtenue est une *correction partielle*.

Enfin, il faut comprendre que les assertions de correction d'un programme sont distinctes de ses spécifications. Les spécifications, parfois aussi appelées le « le cahier des charges », forment un concept souvent informel. L'adéquation des assertions de correction d'un programme avec ses spécifications n'est pas toujours claire. Il se peut qu'on prouve correcte une assertion ne répondant pas aux spécifications. La correspondance entre spécifications et programmes peut se faire progressivement par couches successives, par rafinements successifs. Les spécifications relèvent parfois de méthodologies quelque peu miraculeuses. Il existe aussi des systèmes formels pour spécifier les programmes. Cela est particulièrement important quand on veut faire du zéro-fautes, dans des programmes critiques, tels que les systèmes embarqués pour les transports ou le guidage de missiles, ou les protocoles de sécurité dans les transactions commerciales, ou encore les programmes de conception des circuits intégrés.

5.4 Implémentation des assertions

Les langages modernes ont tous un système d'assertions. En Java 1.4, il existe une instruction spéciale `assert`, qui lève une erreur si l'assertion n'est pas vérifiée. On retrouve aussi cette instruction en Caml, C, C++. En Java 1.1.8, il n'y a pas de système d'assertions, mais il est toujours possible d'écrire sa propre classe.

```
class AssertionError extends Error { }

public class Assertion {
  public static void check (boolean e) {
    if (!e) throw new AssertionError();
  }
}
```

On peut passer en argument de `Assertion.check` une expression booléenne quelconque correspondant à l'assertion à tester. Bien sûr, pour les assertions avec des quantificateurs universels ou existentiels sur des espaces infinis, c'est plus délicat (quelquesoit le langage de programmation !).

Ecrire des assertions dans ses programmes est toujours recommandé. Les systèmes d'exécution peuvent les désarmer sans modifier les programmes en mode de fonctionnement opérationnel. Mais le surcoût engendré par leur vérification est souvent faible devant le gain en fiabilité obtenu en gardant les assertions actives.

5.5 Fonctions et récursion

Le découpage des programmes en fonctions n'induit pas de difficultés supplémentaires dans la méthode des assertions. Il suffit d'avoir une assertion d'entrée et de sortie pour chaque fonction. Pour les programmes récursifs, il en va de même. Prenons l'exemple du calcul récursif de factorielle.

```
static int fact (int n) {
  int r;
  if (n == 0)
    r = 1;
  else
    r = n * fact(n-1);
  return r;
}
```

On veut montrer la formule : $\forall k.\ k \geq 0 \Rightarrow fact(k) = k!$. Pour cela, on suppose, comme pour les invariants de boucles, que les appels récursifs utilisés vérifient déjà cette formule, et on montre qu'elle reste invariante dans le corps de la fonction. On a donc les deux assertions suivantes d'entrée et de sortie dans le corps de la fonction :

```
static int fact (int n) {
  int r;
  {$n \geq 0 \wedge \forall k.\ k \geq 0 \Rightarrow fact(k) = k!$}
  if (n == 0)
    r = 1;
  else
    r = n * fact(n-1);
  {$r = n!$}
  return r;
}
```

et en complétant avec les assertions intermédiaires

```
1        static int fact (int n) {
2          int r;
3         {$n \geq 0 \wedge \forall k.\ k \geq 0 \Rightarrow fact(k) = k!$ }
4         if (n == 0) {
5           {$n = 0$ }
6           r = 1;
7           {$n = 0 \wedge r = 1 = 0!$}
8         } else {
9           {$n - 1 \geq 0 \wedge fact(n-1) = (n-1)!$ }
10          r = n * fact(n-1);
11          {$r = n(n-1)! = n!$ }
12        }
13        {$r = n!$}
14        return r;
15     }
```

Remarquons que $\mathcal{P}_3 \vdash \mathcal{P}_9$ par instanciation. En effet, à la ligne 9, on a $\mathcal{P}_3 \wedge n \neq 0$, et donc $n > 0$. En utilisant la partie quantifiée universellement de P_3, on a $fact(n-1) = (n-1)!$ sur la ligne 9, et donc $\mathcal{P}_9$ est vrai. Le reste se dérive aisément. A nouveau, il ne s'agit que de correction partielle. Un autre argument doit être utilisé pour démontrer la terminaison. Remarquons aussi qu'on n'a pas fait de récurrence sur les entiers, mais simplement supposé l'invariant de sortie vrai pour l'appel récursif. Ce principe de raisonnement est aussi valide pour plusieurs fonctions s'appelant entre elles.

Exercice 29 Montrer que, pour tout P, on a $\{true\}$`x=f(n);`$\{P\}$ quand f est définie par : `static int f(int n) { return f(n); }`.

Un deuxième exemple de correction de programme récursif fait intervenir des listes : c'est le programme élémentaire d'insertion d'un élément dans une liste triée

```
static Liste inserer (int x, Liste a) {
  int r;
  if (a == null || x < a.val) {
    r = new Liste(x, a);
  } else if (a.val < x) {
    r = new Liste(a.val, inserer(x, a.suiv));
  } else
    r = a;
  return r;
}
```

On cherche à montrer $\forall a. \forall x.\ ord(a) \Rightarrow ord(inserer(x, a))$ où $ord(a)$ signifie que la liste a est ordonnée, c'est-à-dire que si a est la liste contenant la suite $\langle a_1, a_2, \ldots a_n \rangle$ ($n \geq 0$), on a $a_1 \leq a_2 \leq \ldots a_n$. On peut aussi définir récursivement ce prédicat par :

$$ord(a) = (a \neq null \wedge a.suiv \neq null \ \Rightarrow \ a.val \leq a.suiv.val \wedge ord(a.suiv))$$

Nous avons aussi besoin de l'ensemble des éléments d'une liste a défini par :

$$ens(a) = \begin{cases} \emptyset & \text{si } a = null \\ \{\, a.val\ \} \cup ens(a.suiv) & \text{sinon} \end{cases}$$

Et on montre que l'on a aussi :

$$\forall a. \forall x.\ ens(inserer(a, x)) \subset \{x\} \cup ens(a)$$

Toutes les assertions intermédiaires figurent sur le programme suivant :

```
1   static Liste inserer (int x, Liste a) {
2     int r;
3     {ord(a) ∧ ∀a.∀x. ord(a) ⇒ ord(inserer(x,a)) ∧
                       ∀a.∀x. ens(inserer(a,x)) ⊂ {x} ∪ ens(a)}
4     if (a == null || x < a.val) {
5       {ord(a) ∧ (a ≠ null ⇒ x < a.val)}
6       r = new Liste(x, a);
7       {ord(r) ∧ ens(r) = {x} ∪ ens(a)}
8     } else if (a.val < x) {
9       {ord(a) ∧ a ≠ null ∧ a.val < x ∧ ord(inserer(x, a.suiv)) ∧
                        ens(inserer(x, a.suiv)) ⊂ {x} ∪ ens(a.suiv)}
10      r =  new Liste(a.val, inserer(x,  a.suiv));
11      {ord(r) ∧ ens(r) ⊂ {x} ∪ ens(a)}
12    } else {
13      {ord(a)}
14      r = a;
15      {ord(r) ∧ ens(r) = ens(a)}
16    }
17    {ord(r) ∧ ens(r) ⊂ {x} ∪ ens(a)}
18    return r;
19  }
```

Les raisonnements non triviaux font passer de $\mathcal{P}_5$ à $\mathcal{P}_7$, et de P_9 à $\mathcal{P}_{11}$. Pour le premier cas, on remarque que si $a = null$, alors r est une liste singleton et donc $ord(r)$. Si $a \neq null$, alors $x < a.val$ et donc, comme r est la liste contenant x puis continuant par la liste ordonnée a, on a aussi $ord(r)$. Pour le second cas, si $\mathcal{P}_9$ est vrai, on sait, sur la ligne 11, que r est la liste contenant $a.val$ puis continuant par la liste ordonnée $inserer(x, a.suiv)$. Or comme $a.val < x$ et comme $ord(a)$ est vrai, on a $a.val < y$ pour tout $y \in \{x\} \cup ens(a.suiv)$. Comme $ens(inserer(x, a.suiv)) \subset \{x\} \cup ens(a.suiv)$, on a aussi $a.val < y$ pour tout $y \in ens(inserer(x, a.suiv))$. D'où $ord(r)$. Par ailleurs, par $\mathcal{P}_9$, on a $ens(inserer(x, a.suiv)) \subset \{x\} \cup ens(a.suiv)$. D'où on déduit pour le résultat r que $ens(r) \subset \{a.val\} \cup \{x\} \cup ens(a)$. Comme $a \neq null$, on obtient $ens(r) \subset \{x\} \cup ens(a)$.

Exercice 30 Démontrer la correction de ce programme sans utiliser d'ensembles. Indication : on peut se servir du prédicat $prem(x, a) = (a \neq null \Rightarrow inserer(x, a).val = x \vee inserer(x, a).val = a.val)$.

5.6 Terminaison et correction totale

La méthode des assertions ne montre que la correction partielle d'un programme. Elle ne traite pas de la terminaison. Pour montrer qu'un programme termine, il faut alors considérer des ordinaux sur les points du programme et vérifier qu'ils baissent à chaque passage dans un espace bien-fondé, c'est-à-dire ne contenant pas de chaîne infinie décroissante. Reprenons l'exemple de la fonction de Fibonacci, on considère alors les assertions et ordinaux suivants :

```
static int fibonacci (int n) {
  {n ≥ 0}
  int x = 0;
  if (n != 0) {
    x = 1; int y = 0;
    int k = 1;
    {Ω(n,k) = n − k ∧ n ≥ k}
    while (k != n) {
      int t = y;
      y = x;
      x = x + t;
      ++k;
    }
  }
  {x = fib(n)}
  return x;
}
```

A chaque itération, on considère l'ordinal $\Omega(n,k) = n-k$ toujours positif puisqu'on peut vérifier qu'on a toujours $n \geq k$ à ce point du programme. En un tour de boucle, la paire (n,k) devient $(n,k+1)$ et on a donc $\Omega(n,k) = n-k > n-k-1 = \Omega(n,k+1)$. Ce programme termine puisque l'ordinal finit par atteindre 0 après un nombre fini d'itérations.

Considérons le programme du PGCD. Un raisonnement analogue consiste à définir l'ordinal $\Omega(x,y) = \max(x,y)$. Ici encore, en un tour de boucle, si $x > y$, on a $\Omega(x,y) = \max(x,y) > \max(x-y,y) = \Omega(x-y,y)$. De même, si $x < y$, on a $\Omega(x,y) = \max(x,y) > \max(x,y-x) = \Omega(x,y-x)$. Par ailleurs $\Omega(x,y)$ est toujours positif, puisque l'invariant $x > 0 \wedge y > 0$ est maintenu à ce point du programme. Le programme s'arrête donc. On peut remarquer que ce raisonnement s'appuie sur le fait que x et y sont strictement positifs.

```
static int pgcd (int a, int b) {
  {a > 0 ∧ b > 0}
  int x = a, y = b;
  {x > 0 ∧ y > 0 ∧ Ω(x,y) = max(x,y)}
  while (x != y)
    if (x > y)
      x = x - y;
    else
      y = x - y;
  return x;
}
```

Exercice 31 Montrer la terminaison de l'algorithme d'Euclide.

Mais les relations d'ordre considérées peuvent dépasser le simple cas de l'inégalité sur les entiers naturels. Plus généralement, si une *relation d'ordre* (strict) $\prec$ est toute relation binaire, irréflexive et transitive, c'est-à-dire vérifiant

$$\begin{array}{ll} (i) & \neg(x \prec x) \\ (ii) & x \prec y \prec z \Rightarrow x \prec z \end{array}$$

On dit qu'elle est *bien-fondée* s'il n'existe pas de chaîne infinie descendante $x_0 \succ x_1 \succ x_2 \succ \cdots x_n \succ \cdots$. Quelques exemples d'ordres bien-fondés sont les entiers naturels avec l'ordre sur les entiers $\langle \mathbf{N}, < \rangle$, le produit cartésien d'entiers avec l'ordre lexicographique $\langle \mathbf{N} \times \mathbf{N}, <_\ell \rangle$, ou le produit cartésien d'entiers avec l'ordre sur les multi-ensembles.

$\langle \mathbf{N} \times \mathbf{N}, <_m \rangle$ où :

$$\begin{array}{ll} (x,y) <_\ell (x',y') & \text{si} \quad x < x' \ \vee \ (x = x' \wedge y < y') \\ (x,y) <_m (x',y') & \text{si} \quad (x_1,y_1) <_\ell (x'_1,y'_1) \end{array}$$

où (x_1, y_1) et (x'_1, y'_1) sont (x, y) et (x', y') en ordre décroissant ($x_1 \geq y_1$, $x'_1 \geq y'_1$). Par exemple, on a $(4,3) >_\ell (4,2) >_\ell (3,15) >_\ell (3,6) >_\ell (3,4) >_\ell (3,2) >_\ell (2,21) \ldots$ et $(2,9) >_m (8,7) >_m (8,6) >_m (7,7) >_m (4,6) >_m (5,4) >_m (5,2) \ldots$.

Ainsi on peut démontrer la terminaison de la fonction d'Ackermann (ci-dessous). En effet, on considère la paire (m, n) des arguments munie de l'ordre lexicographique $<_\ell$. Alors pour tout m et n ($m \geq 0$, $n \geq 0$), la paire (m, n) des arguments décroit strictement dans les appels récursifs. En effet $(m,n) >_\ell (m, n-1)$ et $(m,n) >_\ell (m-1, p)$ pour tout p. Voici un exemple d'ordinaux engendrés par *ack(2,3)*, montrant que l'ordre lexicographique est une méthode assez puissante de démonstration de terminaison : $(2,3)$, $(2,2)$, $(2,1)$, $(2,0)$, $(1,1)$, $(1,0)$, $(0,1)$, $(0,2)$, $(1,3)$, $(1,2)$, $(1,1)$, $(1,0)$, $(0,1)$, $(0,2)$, $(0,3)$, $(0,4)$, $(1,5)$, $(1,4)$, $(1,3)$, $(1,2)$, $(1,1)$, $(1,0)$, $(0,1)$, $(0,2)$, $(0,3)$, $(0,4)$, $(0,5)$, $(0,6)$, $(1,7)$, $(1,6)$, $(1,5)$, $(1,4)$, $(1,3)$, $(1,2)$, $(1,1)$, $(1,0)$, $(0,1)$, $(0,2)$, $(0,3)$, $(0,4)$, $(0,5)$, $(0,6)$, $(0,7)$, $(0,8)$.

```
static int ack (int m, int n) {
  {m ≥ 0 ∧ n ≥ 0 ∧ Ω(m,n) = (m,n)}
  if (m == 0)
    return n + 1;
  else
    if (n == 0)
      return ack (m - 1, 1);
    else
      return ack (m - 1, ack (m, n - 1));
}
```

Exercice 32 Prouver la terminaison de la fonction 91 de McCarthy définie par :

```
static int f (int x) {
  if (x > 100)
    return x-10;
  else
    return f(f(x+11));
}
```

Chapitre 6

Concurrence

TRÈS souvent, on doit exécuter plusieurs instructions simultanément, ceci pour plusieurs raisons. D'abord, il existe des machines avec plusieurs processeurs de calcul, et il faut bien les programmer. Ensuite, de nombreux programmes contrôlent des automatismes réagissant à des événements asynchrones (programmation réactive). C'est le cas des jeux dont les programmes réagissent à plusieurs boutons que l'on peut actionner dans des ordres imprévisibles. De même, dans les systèmes embarqués, on réagit aux informations fournies régulièrement par des capteurs. Une autre forte raison à la programmation concurrente est la recherche de la diminution des temps de latence. Par exemple, si un programme controle des claviers ou des périphériques lents, il ne faut pas bloquer tout un système dans l'attente d'événements très peu fréquents. Plus généralement, si on croit que les ordinateurs simulent les humains, on remarque que ces derniers ont un comportement concurrent. Par exemple, on n'arrête pas le fonctionnement de son cœur pour lire ces notes de cours. Une autre raison pour s'intéresser au comportement concurrent des programmes est la multiplicité des utilisateurs d'un système informatique. De même, sur un réseau informatique, un grand nombre d'activités ou d'utilisateurs co-existent. Si on écrit un programme de gestion du réseau, on doit pouvoir traiter l'exécution simultanée de plusieurs tâches. Une dernière raison de s'intéresser aux programmes parallèles est qu'on peut obtenir un facteur d'accélération, puisqu'avec plus de puissance de calcul on peut résoudre plus rapidement un problème. Si cet argument est vrai dans le principe et fait l'objet de l'algorithmique parallèle, il faut remarquer que le facteur d'accélération ne dépasse jamais le nombre de processeurs exécutant ce programme. Souvent il est moindre, car une certaine synchronisation est nécessaire entre les diverses tâches s'exécutant en parallèle.

Toutes ces raisons attestent du besoin d'exprimer la concurrence par programme. Dans ce chapitre, nous introduisons les notions de base de la concurrence : processus, sémaphores, conditions, et problèmes d'ordonnancement.

6.1 Processus

Un *processus* (*thread*) est un programme qui s'exécute. Un programme est le texte décrivant une suite d'instructions. Un processus correspond à l'action de l'exécuter. Jusqu'à présent aucune distinction n'était faite, car on ne considérait qu'un seul processus. Dans ce chapitre, comme il sera toujours question de plusieurs processus, la distinction devient importante et on remarque, par exemple, que plusieurs processus peuvent exécuter un même programme.

La partie concurrente de Java correspond à l'interface standard des bibliothèques de processus POSIX (the *Portable Operating System Interface*), qui existent actuellement

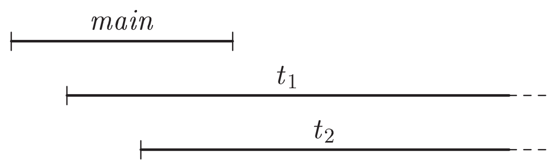


FIG. 6.1 – Exécution concurrente de trois processus

sur pratiquement tous les systèmes et toutes les architectures matérielles. Certaines caractéristiques comme le style en programmation par objets sont plus spécifiques à Java.

Voici un programme qui démarre comme d'habitude par la fonction *main*, puis qui crée deux processus t_1 et t_2 exécutant en parallèle la méthode *run* de la classe *Code*. Cette méthode imprime continuellement le numéro du processus courant, puis laisse un autre processus s'exécuter, puis recommence l'impression du processus courant, etc.

```
class Code implements Runnable {
  public void run () {
    while (true) {
      System.out.println ("Bonjour de " + Thread.currentThread());
      Thread.yield();
    }
  }

  public static void main (String[ ] args) {
    Code p = new Code();
    Thread t1 = new Thread(p);
    Thread t2 = new Thread(p);
    t1.start(); t2.start();
  }
}
```

L'interface *Runnable* ne contient que la méthode *run* qui est appelée au lancement d'un processus. Le constructeur des processus (*Thread*) prend un objet de la classe *Runnable* et le transforme en processus. Le lancement des processus t_1 et t_2 se fait en appelant la méthode *start*. Les deux processus t_1 et t_2 exécutent le même code p. La fonction statique *currentThread* donne la valeur du processus courant. On peut aussi ré-écrire le programme en style programmation par objets, style plus simple, mais qui donne moins de latitude sur la hiérarchie des classes utilisées :

```
class Code1 extends Thread {
  public void run () {
    while (true) {
      System.out.println ("Bonjour de " + this);
      Thread.yield();
    }
  }

  public static void main (String[ ] args) {
    Thread t1 = new Code1();
    Thread t2 = new Code1();
    t1.start(); t2.start();
  }
}
```

Remarquer le remplacement de *currentThread* en *this*, puisque *this* est alors le processus en cours d'exécution.

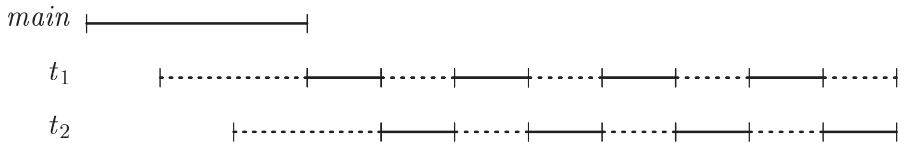


FIG. 6.2 – Exécution concurrente de trois processus sur un processeur

Le programme précédent comporte trois processus : le processus correspondant au programme principal démarrant par la fonction *main*, les deux processus créés t_1 et t_2. Un processus s'exécute sur un *processeur*. Un processeur est une entité matérielle de calcul. Tout ordinateur contient au moins un processeur. Pour que trois processus puissent s'exécuter concurremment, il faudrait au moins trois processeurs. Or bien souvent les machines actuelles n'ont qu'un seul processeur. L'environnement d'exécution simule alors l'existence des trois processeurs demandés, en attribuant par exemple périodiquement une tranche de temps à chacun des trois processus, comme illustré sur la figure 6.2. Nous reviendrons plus tard sur les stratégies d'allocation de temps pour les processus. Pour l'instant, nous ne considérons que la fonction *yield* qui cède le processeur sur lequel le processus tourne et donne plus de chance aux autres processus de s'exécuter immédiatement.

Dans la terminologie officielle des systèmes d'exploitation, nos processus (*threads*) sont plutôt appelés des *processus légers* (*light-weight processes*). Les processus (lourds) sont les processus gérés directement par le système d'exploitation (Unix, Linux, Windows). Dans le système Unix, on obtient leur liste par la commande `ps` ou `ps aux`. Un processus lourds peut générer plusieurs processus légers, qui sont donc gérés par un sous-système, par exemple par l'interpréteur de programmes Java (la *Java Virtual Machine* — JVM). Les processus sont dits légers car basculer d'un processus à un autre est censé être une opération peu coûteuse. On dit alors que le changement de contexte est rapide. Du point de vue de la programmation, la principale différence entre processus lourds et légers concerne la mémoire : les processus légers partagent la zone mémoire, les processus lourds possèdent leur propre mémoire. De toutes façons, ici, comme nous ne nous soucions pas du système d'exploitation sous-jascent, les processus légers sont simplement appelés des processus.

6.2 Terminaison

Un processus peut se terminer spontanément. On peut aussi vouloir l'arrêter depuis un autre processus. Dans l'exemple précédent, le processus principal s'arrêtait juste après avoir lancé le processus t_2. Les processus t_1 et t_2 eux ne s'arrêtaient pas et continuaient leur exécution indéfiniment. Cela n'est pas trop grave, puisqu'on peut toujours continuer à faire d'autres tâches en parallèle. Pourtant, en utilisant la fonction *interrupt* qui lève l'exception *InterruptedException*, on peut interrompre l'exécution d'un processus comme indiqué dans le programme suivant :

```
class Exemple1 implements Runnable {
  public void run () {
    try {
      while (true) {
        System.out.println ("Bonjour de " + Thread.currentThread());
```

```
        Thread.sleep (1000); // Une seconde de pause
      }
    } catch (InterruptedException e) { }
  }

  public static void main (String[ ] args) {
    Thread t = new Thread (new Exemple1());
    t.start();
    t.interrupt();
  }
}
```

La fonction *sleep* arrête l'exécution d'un processus pendant le nombre de millisecondes données en argument. On l'utilise ici pour faire une pause entre deux impressions. Sur notre exemple, on constate que le processus t n'est pas arrêté brutalement depuis un autre processus puisqu'une exception lui est simplement envoyée. C'est le processus t qui gère sa propre terminaison. Ainsi les invariants logiques de t sont préservés, et le processus t peut se terminer dans un état cohérent.

Un processus peut aussi tester régulièrement s'il n'est pas interrompu, au lieu d'attendre une exception. Pour cela, il fait une attente active sur la fonction booléenne *isInterrupted* de la bibliothèque standard de Java :

```
class Exemple2 extends Thread {
  public void run () {
    while (!isInterrupted()) {
      System.out.println ("Bonjour de " + this);
    }
  }

  public static void main (String[ ] args) {
    Thread t = new Exemple2();
    t.start();
    t.interrupt();
  }
}
```

Un processus attend souvent la fin d'un autre processus. Par exemple, on lance un processus t et on attend qu'il finisse. Cela peut se réaliser en utilisant la fonction *join* qui prend en argument un délai maximum d'attente en millisecondes (0 si ce délai est infini). Ainsi :

```
class Exemple3 extends Thread {
  public void run () {
    System.out.println ("Bonjour de " + this);
  }

  public static void main (String[ ] args)
    throws InterruptedException {
    Thread t = new Exemple3();
    t.start();
    t.join (0);
  }
}
```

La méthode *join* lance une exception si un autre processus a interrompu le processus courant (qui est en attente).

Nous avons donc considéré la création, le lancement et l'arrêt des processus. Ceci constitue l'essentiel si les processus ne communiquent pas entre eux. Mais c'est rarement le cas, puisque si plusieurs activités sont menées en parallèle, c'est souvent pour calculer un résultat commun ou partager certaines informations. Le reste du chapitre décrit la synchronisation nécessaire pour communiquer entre processus. Plusieurs modèles standards existent pour cette communication. Ici nous allons considérer le modèle simple de la mémoire partagée.

6.3 Variables partagées

Dans le programme suivant, la variable globale x de la classe P peut être modifiée par les deux processus t_1 et t_2. Comme ces deux processus s'exécutent en parallèle, il est dûr de prédire la valeur de x imprimée dans *main* : 0, 23, 7 ou autre chose ?

```
class P {
  static int x = 0;

  public static void main (String args[ ])
    throws InterruptedException {
    Thread t1 = new P1(), t2 = new P2();
    t1.start(); t2.start();
    while (true) {
      System.out.println (x);
      Thread.sleep (1000);
} } }

class P1 extends Thread {
  public void run () {
    while (true) {
      P.x = 23;
      Thread.yield();
} } }

class P2 extends Thread {
  public void run () {
    while (true) {
      P.x = 7;
      Thread.yield();
} } }
```

Pour répondre à cette question, il faut comprendre l'*atomicité* des instructions et l'ordonnancement des processus. Une instruction est atomique si elle ne peut être interrompue pour laisser l'exécution à un autre processus. Très peu d'instructions sont atomiques au niveau du langage de programmation puisque chaque instruction correspond à plusieurs instructions machine exécutées par le processeur. Toutefois on peut dire que la lecture ou de l'écriture d'un mot (de 32 bits) dans la mémoire d'un ordinateur est atomique. Au niveau de Java, on peut donc dire que l'écriture d'un entier dans une variable et que la lecture de la valeur d'une variable entière sont atomiques. Cependant, cela n'est plus vrai pour les entiers longs (de 64 bits) ou pour les flottants en double précision (*double*), sauf si on dispose d'un processeur 64 bits.

Par ailleurs, on ne peut prédire l'ordre dans lequel t_1 ou t_2 modifient la variable partagée x. Cela dépend de l'ordonnancement des processus t_1, t_2 et de celui exécutant *main*. Cet ordre peut varier entre deux exécutions du même programme. La seule conclusion à retenir est que les programmes concurrents sont souvent *non déterministes*. Ils peuvent donner des résultats différents sur les mêmes données dans deux exécutions distinctes. Ce phénomène rend particulièrement délicate la programmation de processus concurrents.

6.4 Sections critiques

Une *section critique* rend atomique un ensemble d'instructions, par exemple pour modifier plusieurs données ou une donnée composite, tout en maintenant des invariants. Prenons l'exemple classique du transfert d'argent entre deux comptes bancaires, on veut que la somme des deux comptes reste constante, même si plusieurs appels de la fonction *transfertVersB* sont faits simultanément dans des processus distincts. Pour cela, il suffit interdire l'exécution de cette fonction pendant qu'un autre processus l'exécute. C'est ce que fait le qualificatif *synchronized* qui contrôle l'accès à une section critique :

```
class ComptesBancaires {
  private int compteA, compteB;

  synchronized void transfertVersB (int s) {
    compteA = compteA - s;
    compteB = compteB + s;
  }
}
```

Le corps de la fonction *transfertVersB* est une section critique qui doit s'exécuter en *exclusion mutuelle*. Les exécutions de cette fonction sont donc sérialisées, c'est-à-dire faites séquentiellement, puisqu'on garantit qu'un seul appel de cette fonction s'exécute au plus à un moment donné. Pour cela, le processus, qui appelle une méthode avec le qualificatif *synchronized*, doit obtenir un verrou sur cette fonction. Si le verrou est déjà pris par ce processus ou par un autre, l'appel est suspendu jusqu'à ce que le verrou soit disponible. En Java, comme tout est objet, les verrous sont associés à des objets : une méthode synchronisée doit prendre le verrou sur l'objet dont elle est une méthode. Il n'y a qu'un verrou par objet. Si deux méthodes d'un même objet sont synchronisées, elles s'excluent donc mutuellement. Par conséquent, une méthode synchronisée au plus peut être appelée simultanément sur un même objet. Mais une même méthode peut être appelée simultanément sur deux objets différents, car les deux appels prennent alors un verrou différent sur chaque objet. Pour les méthodes statiques, une méthode statique prend un verrou associée à toute la classe si elle est synchronisée.

Finalement, remarquons que seules les méthodes peuvent avoir le mot-clé *synchronized*, les champs de données ne peuvent avoir ce qualificatif. Il est toutefois possible de rendre atomique un ensemble d'instructions, plutôt que le corps d'une fonction. Alors il faut donner en argument l'objet sur lequel on veut prendre le verrou. Cette forme de section critique permet de faire une section critique plus finement que sur des appels de méthodes.

```
class ComptesBancaires {
  private int compteA, compteB;

  void transfertVersB (int a) {
    ...
    synchronized (this) {
      compteA = compteA - s;
      compteB = compteB + s;
    }
    ...
  }
}
```

Il est possible de prendre plusieurs verrous pour exécuter un bout de programme. Mais créer des sections critiques demande un peu d'attention, puisque des *interblocages* (*deadlocks*) sont possibles. Ainsi dans :

```
class P1 extends Thread {
  public void run () {
    synchronized (a) {
      synchronized (b) {
        ...
} } } }

class P2 extends Thread {
  public void run () {
    synchronized (b) {
      synchronized (a) {
        ...
} } } }
```

Si deux processus t_1 et t_2 exécutent les codes de P_1 et P_2 en parallèle, le processus t_1 peut prendre le verrou sur a ; le processus t_2 peut alors prendre le verrou sur b, et il y a un interblocage, puisque les deux verrous sont pris et que chaque processus attend la libération d'un verrou pris par l'autre processus. Détecter les interblocages peut être très complexe, c'est un des problèmes standards des systèmes concurrents. Souvent un tel système s'arrête à cause d'un tel interblocage.

Exercice 33 Faire un exemple d'interblocage avec des appels sur des méthodes synchronisées.

6.5 Conditions

Un paradigme de la programmation concurrente est celui de la file d'attente concurrente. Pour simplifier, supposons que la longueur maximale de la file vaut 1. Alors la file ne peut qu'osciller entre les deux états : vide ou plein. Les deux méthodes *ajouter* et *supprimer* peuvent s'exécuter de manière concurrente. On pourrait être tenté d'écrire :

```
class FIFO {
  boolean  pleine;
  int      contenu;
  FIFO () { pleine = false; }

  synchronized void ajouter (int x) {
    if (pleine)
      // Attendre que la file se vide
    contenu = x;
    pleine = true;
  }

  synchronized int supprimer () {
    if (!pleine)
      // Attendre que la file devienne pleine
    pleine = false;
    return contenu;
  }
}
```

La fonction *ajouter* attend que la file se vide, la fonction *supprimer* attend que la file se remplisse. Ces deux opérations s'excluent mutuellement pour que les variables internes de la file (*contenu*, *pleine*) restent dans un état cohérent. Mais ce code ne fonctionne pas, puisqu'un ajout dans une file pleine prend le verrou et interdit à quiconque de vider la file. Dans chacun des cas d'attente, la fonction devrait relâcher le verrou pour que l'autre processus puisse s'exécuter. Il faut donc ressortir de chaque méthode sans se bloquer et revenir tester la file. C'est alors une attente active (*busy wait*) qui coûte cher en temps. Il est préférable d'utiliser le concept de *condition* dû à Hoare. Une condition est associée à un verrou. Quand on attend sur une condition, on relâche atomiquement le verrou et on attend un signal sur la condition. Quand il y a notification d'un signal sur la condition, on reprend le verrou (si possible) et on redémarre à l'endroit où on s'était mis en attente.

En Java, une condition est une simple référence à un objet (son adresse). Il n'y a qu'une seule condition par objet (ce qui est une solide restriction). On peut attendre sur la condition d'un objet dont on a déjà pris le verrou. L'exemple précédent s'écrit donc précisément :

```
synchronized void ajouter (int x)
  throws InterruptedException {
  while (pleine)
    wait();
  contenu = x;
  pleine = true;
  notify();
}

synchronized int supprimer ()
  throws InterruptedException {
  while (!pleine)
    wait();
  pleine = false;
  notify();
  return contenu;
}
```

Les fonctions *wait* et *notifiy* sont deux méthodes de la classe *Object*. Tous les objets possèdent ces deux méthodes puisque toute classe est une sous-classe de *Object*. La méthode *wait* attend sur la condition et relâche le verrou de l'objet correspondant à la condition. Si ce verrou n'est pas pris, l'exception *IllegalMonitorStateException* est levée. (Cette exception est une exception du système d'exécution, c'est-à-dire sous-classe de *RuntimeException* ; il n'est pas nécessaire de la mentionner dans la signature des fonctions *ajouter* et *supprimer*). La méthode *notify* met dans l'état prêt à l'exécution un des processus en attente sur cette condition. Quand celui-ci repart, il reprend automatiquement le verrou. Remarquons que l'on itère sur l'attente de la condition avec une instruction *while*, plutôt que de faire une simple instruction conditionnelle *if*. En effet, entre le moment où le processus est réveillé et celui où il s'exécute, rien n'empêche qu'un autre processus ne prenne le verrou n'invalide le test sur l'état de la file. Il faut donc toujours utiliser une telle instruction *while*. Si on veut réveiller plusieurs processus, on utilise la méthode *notifyAll*, qui réveille *tous* les processus en attente sur la condition.

Enfin, on remarque que les les méthodes *wait* et *notify* peuvent générer l'exception *InterruptedException* en cas d'interruption des processus les appelant. Au total, le programme contenant le lancement des deux processus t_1 et t_2 qui appellent les fonctions *ajouter* et *supprimer* s'écrit comme suit.

```
class FIFO {
  ... // définitions de la classe, d'ajouter et de supprimer

  public static void main (String[ ] args)
    throws InterruptedException {
```

```
    FIFO f = new FIFO();
    Thread t1 = new P1(f), t2 = new P2(f);
    t1.start(); t2.start();
    Thread.sleep (200);
    t1.interrupt(); t2.interrupt();
} }

class P1 extends Thread {
  static int i = -1;
  FIFO f;
  P1 (FIFO x) {f = x; }

  public void run () {
    try {
      while (true) f.ajouter(++i);
    } catch (InterruptedException s) { }
} }

class P2 extends Thread {
  FIFO f;
  P2 (FIFO x) {f = x; }

  public void run () {
    try { while (true)
        System.out.println(f.supprimer());
    } catch (InterruptedException s) { }
} }
```

Exercice 34 Expliquer ce qu'il se passe si on n'interrompt pas le processus t_2.

Exercice 35 Modifier le programme pour que la file puisse contenir n éléments ($n > 1$).

Exercice 36 Ecrire un programme de gestion d'une pile concurrente avec deux méthodes *ajouter* et *supprimer*.

Pour bien comprendre le sens de *wait* et de *notify*, on remarque que cette dernière fonction n'a pas de mémoire. Elle ne fait qu'envoyer un signal à un des processus en attente. Rien n'est enregistré. C'est pourquoi la méthode *wait* atomiquement relâche le verrou et attend sur la condition. Sinon, un processus pourrait prendre le verrou et envoyer un signal de reprise avant que l'on attende sur la condition et ce signal serait perdu.

Exercice 37 Expliquer ce qu'il se passe si *notify* est fait après avoir relâché le verrou.

6.6 Etats d'un processus et ordonnancement

Un processus peut être dans plusieurs états comme indiqué sur la figure 6.3. Dans l'état *prêt*, un processus est prêt à être exécuté. Si un processeur devient disponible, il passe dans l'état *exécution*. Alors le processus s'exécute. En Java, il donne alors sa valeur à la variable globale *currentThread* qui indique à tout moment le processus courant. Un processus peut quitter l'état d'exécution pour plusieurs raisons. Par exemple, il termine en passant à l'état *mort*. Il peut aussi laisser sa place à un autre processus en repassant à l'état prêt. Il peut enfin passer à l'état *bloqué*, car en attente d'une condition ou d'un verrou. Si ceux-ci se libèrent, il passe dans l'état prêt.

Il peut donc y avoir plusieurs processus prêts à l'exécution. Si le nombre de processeurs est suffisant, ils peuvent tous passer à l'état d'exécution. Si ce n'est pas le cas

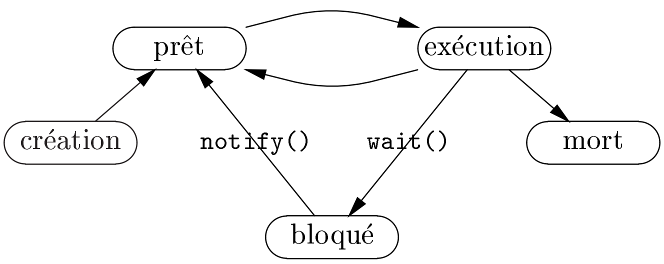


FIG. 6.3 – Les états d'un processus

(comme cela l'est bien souvent), le système doit choisir un des processus pour l'exécuter. En Java, c'est la JVM qui doit faire ce choix. Il lui faut donc une politique d'ordonnancement des processus. Les politiques d'ordonnancement se rangent en deux grandes catégories.

La stratégie d'allocation des processeurs peut être non préemptive. Cela signifie qu'un processus ne relâche pas un processeur tant qu'il ne l'a pas fait explicitement, par exemple avec la fonction *Thread.yield*. On dit aussi parfois que les processus fonctionnent alors en *co-routines*. Le désavantage d'une telle technique est que rien ne peut empêcher un processus gourmand de monopoliser un processeur. Le gros avantage de cette technique est sa simplicité, car c'est le programme qui se charge de l'ordonnancement.

La deuxième technique d'allocation des processeurs est *préemptive*. Cela signifie que le système d'exécution des processus peut arrêter à tout moment un processus pour laisser la place à un autre. Souvent cela fonctionne avec un quantum de temps que l'exécution de tout processus ne doit pas dépasser avant d'abandonner le processeur.

Au niveau de Java, les transitions entre les divers états d'un processus sont donnés par :
- le constructeur de la classe *Thread* pour arriver dans l'état création,
- la méthode *start* pour passer de création à prêt,
- l'ordonnanceur de la JVM pour passer de prêt à exécution,
- la méthode *Thread.yield* ou la fin d'un quantum de temps (pour les systèmes préemptifs) pour passer d'exécution à prêt,
- la méthode *wait* ou l'appel à une méthode synchronisée pour passer d'exécution à bloqué,
- les méthodes *notify*, *notifyAll* ou la fin d'une fonction synchronisée d'un autre processus pour passer de bloqué à prêt,
- la fin de la méthode *run* pour passer d'exécution à mort.

Tous les processus ne sont pas égaux. On peut établir des priorités entre eux dans l'accès à l'état d'exécution. Les priorités sont simplement des nombres entiers associés à chaque processus. Les priorités statiques peuvent être manipulées et affectées par le programmeur avec les fonctions *getPriority* et *setPriority*. On peut utiliser les priorités suivantes prédéfinies : *MIN_PRIORITY*, *NORM_PRIORITY*, *MAX_PRIORITY*. L'ordonnanceur peut aussi affecter des priorités dynamiques pour gérer sa politique d'allocation du (des) processeur(s). La spécification de Java dit qu'en général les processus de forte priorité vont s'exécuter avant les processus de basse priorité.

Cependant, il est important d'écrire du code portable, donc de comprendre ce que la JVM garantit sur tous les systèmes. Ecrire du code contrôlant l'accès à des variables partagées en faisant des hypothèses sur la priorité des processus ou sur leur vitesse relative (*race condition*) est *très dangereux*, et doit donc être banni.

A titre de curiosité, nous considérons deux exemples de politique d'ordonnancement : la priorité sur l'age et la priorité dans le système Unix 4.3 BSD. Dans la stratégie de la priorité sur l'age, tout processus a initialement une priorité $p = p_0$ dite priorité statique. Si un processus prêt attend le processeur pendant k secondes, on incrémente p. Le processus qui s'exécute est choisi parmi les processus prêts de plus forte priorité. Quand il s'exécute, on refait $p = p_0$. Cette politique garantit qu'un processus en attente d'exécution depuis longtemps va prendre le processeur. Après exécution, on oublie son histoire et on revient à la priorité statique initiale. Cette stratégie donne donc la priorité aux vieux.

Dans la politique d'ordonnancement des processus du système Unix, au début $p = p_{nice}$ est la priorité statique initiale, où on a $-20 \leq p_{nice} \leq 20$ (-20 pour le plus prioritaire, et 20 pour le moins prioritaire). Nous écrivons p_{nice} au lieu de p_0 pour rappeler que c'est la commande *nice* d'Unix qui fixe cette valeur. Toutes les 40ms, la priorité p d'un processus est recalculée par :

$$p = PUSER + \left\lceil \frac{p_{cpu}}{4} \right\rceil + 2\, p_{nice}$$

où $PUSER$ est une même constante pour tous les processus utilisateurs (ainsi distingués des processus du système), et p_{cpu} est incrémentée toutes les 10 ms quand le processus s'exécute. Ainsi la valeur de p augmente si on utilise longtemps le processeur, le processus devient donc moins prioritaire. Les processus correspondants à des programmes interactifs, peu consommateurs du processeur, sont donc avantagés. Cependant, toutes les secondes, p_{cpu} est aussi modifiée par le filtre suivant :

$$p_{cpu} = \frac{2\ load}{2\ load + 1}\, p_{cpu} + p_{nice}$$

qui fait baisser sa valeur en fonction de la charge globale du système *load* (la charge est le nombre moyen de processus prêts sur la dernière minute). Si la charge est grande, la valeur de p_{cpu} est peut altérée, et les processus gros consommateurs de processeur sont toujours pénalisés. Au contraire, si la charge est faible, la valeur de p_{cpu} diminue et on essaie de faire disparaitre aussi vite que possible un processus qui s'exécute en lui donnant plus longtemps le processeur. (A nouveau, dans le système Unix, la priorité croit dans l'ordre inverse de sa valeur : $p < 0$ est très prioritaire ; $p = 127$ correspond à un processus peu prioritaire).

Finallement, voici un exemple démontrant le danger de l'interaction entre la synchronisation et les priorités, c'est le phénomène d'inversion de priorités suivant : soient trois processus t_b, t_m, t_h s'exécutant en priorité basse, moyenne, et haute. Supposons que t_b prenne un verrou qui bloque t_h. Si maintenant t_m s'exécute, il empêche t_b (de priorité plus faible) de s'exécuter. Supposons que t_m ne relâche pas le processeur, alors t_b ne peut progresser, et le processus de plus faible priorité t_m empêche le processus t_h de s'exécuter. On peut s'en sortir en affectant des priorités sur les verrous en notant qu'un processus de plus haute priorité attendant sur ce verrou, et on fait monter la priorité d'un processus ayant pris un verrou bloquant un processus de forte priorité (cela fonctionne en Windows). Pour conclure, un bon conseil est d'éviter le mélange de priorités et de synchronisations.

6.7 Les lecteurs et les écrivains

L'exemple de la file d'attente avec accès concurrents ne fait pas de distinction entre les phases de lecture et d'écriture. Pourtant il est très courant d'avoir des fonctions d'accès différenciés entre celles ne consistant qu'à lire la structure de donnée partagée et celles la modifiant. La particularité de la lecture est qu'elle ne change pas la valeur de la donnée lue. Plusieurs lectures concurrentes peuvent donc s'effectuer simultanément. Pour l'écriture, au contraire, il faut s'assurer qu'elle se fait en exclusion mutuelle de toutes les autres fonctions d'accès à la donnée, puisque, si on lit une donnée en cours de modification, on peut obtenir un résultat incohérent. C'est le problème dit des lecteurs et des écrivains. Une donnée partagée peut autoriser la lecture par n ($n > 0$) processus simultanément, mais ne permet l'écriture que par un seul processus. Tous les systèmes de réservation de places en ligne (train, avion, théatre) sont des exemples de ce problème, puisqu'il ne s'agit pas de signaler une place disponible au moment même où un autre client la prend.

La lecture commence par exécuter la fonction *accesPartage* qui vérifie que l'on peut obtenir un accès partagé à la donnée que l'on veut lire. Elle se finit par une fonction *retourPartage* qui remet en place les verrous pris pour effectuer la lecture. De même, l'écriture commence par l'exécution de la fonction *accesExclusif* et se finit par une autre fonction *retourExclusif*. Nous supposons qu'une variable globale *nLecteurs* donne à tout moment le nombre de lecteurs. Par convention, $nLecteurs = -1$ si un écrivain est en action.

```
void lecture() {
  accesPartage();
  // lire la donnée partagée
  retourPartage();
}

void ecriture() {
  accesExclusif();
  // modifier la donnée partagée
  retourExclusif();
}

synchronized void accesPartage() {
  while (nLecteurs < 0)
    wait();
  ++nLecteurs;
}

synchronized void retourPartage() {
  -- nLecteurs;
  if (nLecteurs == 0)
    notify();
}

synchronized void accesExclusif() {
  while (nLecteurs != 0)
```

```
      wait();
    nLecteurs = -1;
  }

  synchronized void retourExclusif() {
    nLecteurs = 0;
    notifyAll();
  }
```

NotifyAll est pratique, il évite de gérer l'ensemble des lecteurs en attente. Pourtant il peut réveiller trop de processus qui se retrouvent immédiatement bloqués. Pour arriver à un contrôle plus fin, on peut considérer deux conditions séparées en lecture et en écriture, et attendre sur l'une ou sur l'autre. Ceci est impossible en Java, car on ne peut disposer de conditions différentes sur un même verrou, mais cela est possible dans toutes les bibliothèques de processus Posix (un standard assez répandu). Pour exposer notre solution, nous supposons disposer de telles conditions Posix : *wait* est une méthode prenant en argument l'objet dont la condition libère ou reprend le verrou, *notify*, *notifyAll* ont leur sens classique. Avec deux conditions (*lecture* et *écriture*), et une variable globale *nLecteursEnAttente* qui compte le nombre de lecteurs en attente. on obtient le contrôle plus fin suivant :

```
  synchronized void accesPartage() {
    ++ nLecteursEnAttente;
    while (nLecteurs < 0)
      lecture.wait(this);
    -- nLecteursEnAttente;
    ++ nLecteurs;
  }

  synchronized void retourPartage(this) {
    -- nLecteurs;
    if (nLecteurs == 0)
      ecriture.notify();
  }

  synchronized void accesExclusif() {
    while (nLecteurs != 0)
      ecriture.wait(this);
    nLecteurs = -1;
  }

  synchronized void retourExclusif() {
    nLecteurs = 0;
    if (nLecteursEnAttente > 0)
      lecture.notifyAll();
    else
      ecriture.notify();
  }
```

Exécuter *notify* à l'intérieur d'une section critique n'est pas très efficace. Avec un seul processeur, ce n'est pas un problème car les processus réveillés passent dans l'état prêt attendant la disponibilité du processeur. Avec plusieurs processeurs, le processus réveillé peut retomber rapidement dans l'état bloqué, tant que le verrou n'est pas relaché. Le mieux est de faire *notify* à l'extérieur de la section critique.

```
  void retourPartage() {
    boolean faireNotify;
    synchronized (this) {
      -- nLecteurs;
```

```
      faireNotify = nLecteurs == 0;
    }
    if (faireNotify)
      ecriture.notify();
  }
```

Des blocages inutiles sont possibles (même avec plusieurs processeurs) sur le *notifyAll* de fin d'écriture. Comme avant, on peut le sortir de la section critique. Si plusieurs lecteurs sont réveillés, un seul prend le verrou. Mieux vaut faire *notify* en fin d'écriture, puis refaire *notify* en fin d'accès partagé pour relancer les autres lecteurs.

```
  void accesPartage() {
    synchronized (this) {
      ++ nLecteursEnAttente;
      while (i < 0)
        lecture.wait(this);
      -- nLecteursEnAttente;
      ++ nLecteurs;
    }
    lecture.notify();
  }
```

Une famine est possible pour un écrivain en attente de fin de lecture. La politique d'ordonnancement des processus peut aider. On peut aussi logiquement imposer le passage d'un écrivain.

```
  void accesPartage() {
    synchronized () {
      ++ nLecteursEnAttente;
      if (nEcrivainsEnAttente > 0)
        lecture.wait(this);
      while (i < 0)
        lecture.wait(this);
      -- nLecteursEnAttente;
      ++ nLecteurs;
    }
    lecture.notify();
  }

  synchronized void accesExclusif() {
    ++ nEcrivains;
    while (i != 0)
      ecriture.wait(this);
    -- nEcrivainsEnAttente;
    nLecteurs = -1;
  }
```

En conclusion, on constate que contrôler finement la synchronisation peut être complexe.

Exercice 38 Donner un exemple précis où *notifyAll* fait une différence avec *notify*.

Exercice 39 Montrer que le réveil des processus n'est pas forcément dans l'ordre premier en attente, premier réveillé.

Exercice 40 Donner un exemple où un processus peut attendre un temps infini avant d'entrer en section critique.

Exercice 41 Comment programmer un service d'attente où les processus sont réveillés dans l'ordre d'arrivée.

6.8 Implémentation de la synchronisation

Avec des fonctions synchronisées et des opérations atomiques comme *wait* ou *signal*, on peut aisément réaliser la synchronisation entre les processus. Il reste à comprendre comment implémenter ces deux fonctions et la prise d'un verrou. En général, cela dépend de l'architecture matérielle disponible. Très souvent, les ordinateurs disposent d'une instruction machine ininterruptible *Test and Set*. Cette opération *TAS(m)* teste atomiquement si la variable m vaut *false* et la positionne si $m = \mathit{false}$. Le résultat est vrai si $m = \mathit{true}$ avant l'exécution, faux sinon. C'est sur les ordinateurs avec plusieurs processeurs que cette opération prend tout son sens, où une mémoire spéciale de « verrous », accessibles par *TAS*, est souvent partagée entre tous les processeurs.

On programme une section critique ainsi, en posant $m = \mathit{false}$ comme valeur initiale de m :

```
while ( TAS(m) )
  ;
// section critique
m = false;
```

L'opération *TAS(m)* permet de tester et de modifier atomiquement la variable partagée m, et donc de garantir l'exclusion mutuelle pour l'exécution de la section critique. Cependant ce programme fait une attente active sur le passage de la valeur de m à *false*. En fait, comme nous le verrons plus loin avec les sémaphores, l'environnement d'exécution gère un ensemble de processus en attente sur le changement de valeur de certaines variables. Ces processus sont alors suspendus pendant cette attente.

Mais, sans l'instruction *TAS*, on peut tout de même y arriver, en n'utilisant que l'indivisibilité de la lecture ou de l'écriture en mémoire d'un scalaire de base. Ces algorithmes compliqués (Dekker, Peterson) sont des curiosités, car inutiles puisque le matériel procure toujours des verrous avec *TAS*. Cependant, ces algorithmes démontrent bien l'aspect complexe de la concurrence.

```
class Peterson extends Thread {
  static int tour = 0;
  static boolean[ ] actif = {false, false};
  int i, j;
  Peterson (int x) { i = x; j = 1 - x; }

  public void run() {
    while (true) {
      actif[i] = true;
      tour = j;
      while (actif[j] && tour == j)
        ;
      // section critique
      ...
      // fin de section critique
      actif[i] = false;
      Thread.yield();
  } }

  public static void main (String[ ] args) {
    Thread t0 = new Peterson(0), t1 = new Peterson(1);
    t0.start(); t1.start();
} }
```

La démonstration est loin d'être évidente. Montrons d'abord la sureté de ce mécanisme d'exclusion mutuelle. Si t_0 et t_1 entrent tous les deux dans leur section critique, alors $actif[0] = actif[1] = true$. C'est impossible car les deux tests auraient été franchis en même temps alors que *tour* favorise l'un d'entre eux. Donc un des deux est entré. Disons t_0. Cela veut dire que t_1 n'a pu avoir trouvé le *tour* à 1 et qu'il n'est pas entré en section critique.

Montrons à présent la vivacité, c'est-à-dire qu'un processus voulant entrer dans la section critique finit par y entrer. En effet, supposons t_0 bloqué dans le *while*. On a deux cas. Dans le premier cas, le processus t_1 n'est pas intéressé par rentrer dans la section critique. Alors $actif[1] = false$. Et donc t_0 ne peut être bloqué par le *while*. Dans le deuxième cas, le processus t_1 est aussi bloqué dans le *while*. C'est impossible car *tour* vaut 0 ou 1. Donc l'un de t_0 ou t_1 ne peut rester dans le *while*.

Cette démonstration est formalisée avec des assertions logiques faisant intervenir la valeur des variables et des compteurs ordinaux c_0 et c_1, c'est-à-dire des emplacements de l'exécution dans chacun des deux processus. Les valeurs de c_i sont les numéros affichés ci-dessous à gauche des lignes de la fonction *run*. Décorons le code de la fonction avec les assertions suivantes :

```
.  public void run() {
.    while (true) {
       {¬ actif[i] ∧ c_i ≠ 2}
1      actif[i] = true;
       {actif[i] ∧ c_i = 2}
2      tour = j;
       {actif[i] ∧ c_i ≠ 2}
3      while (actif[j] && tour == j)
.        ;
       {actif[i] ∧ c_i ≠ 2 ∧ (¬ actif[j] ∨ tour = i ∨ c_j = 2)}
.      // section critique
.      ...
.      // fin de section critique
5      actif[i] = false;
       {¬ actif[i] ∧ c_i ≠ 2}
6      Thread.yield();
.  } }
```

En outre, on adjoint à toutes les assertions la proposition suivante : $j = 1 - i \wedge (tour = 0 \vee tour = 1)$. On peut vérifier que chaque assertion est vérifiée quand chacun des deux processus s'exécute. On en déduit alors que les deux programmes ne peuvent atteindre la section critique simultanément, puisqu'alors on aurait simultanément les deux invariants de la ligne 5, c'est-à-dire :

$$\begin{array}{rl} & (tour = 0 \vee tour = 1) \\ \wedge & actif[0] \ \wedge \ c_0 \neq 2 \ \wedge \ (\neg\, actif[1] \vee tour = 0 \vee c_1 = 2) \\ \wedge & actif[1] \ \wedge \ c_1 \neq 2 \ \wedge \ (\neg\, actif[0] \vee tour = 1 \vee c_0 = 2) \end{array}$$

qui équivaut à

$$(tour = 0 \vee tour = 1) \wedge tour = 0 \wedge tour = 1$$

ce qui est impossible.

On peut aussi prouver la vivacité avec les assertions. Par exemple, il est impossible que t_0 soit en dehors de l'accès à la section critique pendant que t_1 reste bloqué dans le *while*, car alors :

$$\begin{array}{ll} \wedge & \neg\, actif[0] \wedge c_0 \neq 2 \wedge (tour = 0 \vee tour = 1) \\ \wedge & actif[1] \wedge c_1 \neq 2 \\ \wedge & actif[0] \wedge tour = 0 \end{array}$$

ce qui implique $\neg\, actif[0] \wedge actif[0]$. Impossible! De même, les deux processus t_0 et t_1 ne peuvent rester tous les deux bloqués dans l'instruction *while*, car alors on a :

$$actif[1] \wedge tour = 1 \wedge actif[0] \wedge tour = 0 \wedge (tour = 0 \vee tour = 1)$$

ce qui est aussi impossible.

Exercice 42 Généraliser l'algorithme de Peterson au cas de n processus et d'une section critique en exclusion mutuelle.

6.9 Sémaphores

Une autre forme de synchronisation de bas niveau est la notion de sémaphore due à Dijkstra. Un *sémaphore* est une variable s booléenne, sur laquelle on fait les deux opérations atomiques suivantes :

- prendre (*proberen* en hollandais) le sémaphore : $P(s)$ teste la valeur de s. Si s vaut *true*, on met s à *false*. Sinon l'instruction attend sur s.
- libérer (*verhogen* en hollandais) le sémaphore : $V(s)$ réveille un processus en attente sur s s'il existe un tel processus, et la valeur de s est inchangée. Sinon s est mis à *true*.

A la différence des conditions de Java, les sémaphores ne sont pas attachés à un verrou. Ce sont des variables qui ont un état. On peut les utiliser pour justement programmer un verrou comme dans l'exclusion mutuelle de sections critiques. Ainsi la construction de Java

```
synchronized (o) {
  // section critique
}
```

s'écrit en termes de sémaphores de la manière suivante :

```
P(s0);
try {
  // section critique
} finally V(s0);
```

en supposant que s_o est un sémaphore associé à l'objet o. On peut aussi définir une classe *Semaphore* avec deux méthodes P et V en fonction des primitives *wait* et *notify* de Java :

```
class Semaphore {
  boolean libre;
  Semaphore (boolean x) { libre = x; }

  static void P(Semaphore s) {
    synchronized (s) {
      while (!s.libre)
        try {
          s.wait();
        } catch (InterruptedException x) { }
```

```
      s.libre = false;
      }
    }

    static void V(Semaphore s) {
      synchronized (s) {
      s.libre = true;
      s.notify();
  } } }
```

D'autres définitions de sémaphores sont possibles. Par exemple, pour $V(s)$, on peut d'abord mettre le sémaphore à *true*, puis réveiller un des processus en attente sur s, s'il en existe un. Ainsi, le processus t effectuant $V(s)$ peut continuer à s'effectuer, même si d'autres processus t_1, t_2, $\ldots t_n$ sont en attente sur s (c'est aussi le cas avec la première définition), mais maintenant t peut reprendre le sémaphore avant que le processus t_i réveillé ne passe. On peut aussi définir des sémaphores *FIFO* tels que le premier processus en attente sur s soit le premier réveillé. En fait, ces différences n'interviennent pas dans les propriétés de sureté, mais n'affectent que la vivacité des algorithmes. Cela signifie que les exclusions mutuelles des sections critiques sont bien respectées, mais que la garantie de rentrer dans une section critique n'est pas toujours vraie, car cela peut dépendre de l'équité de la politique d'ordonnancement entre les divers processus.

Avec les sémaphores, on peut implémenter les conditions Posix avec les deux fonctions *wait* et *notify* de la manière suivante :

```
class ConditionPosix {
  Semaphore sem;
  Condition () { s = new Semaphore(false); }

  public void wait (Semaphore m) {
    Semaphore.V(m);
    Semaphore.P(sem);
    Semaphore.P(m);
  }

  public void notify () {
    Semaphore.V(sem);
  }
}
```

Toutefois, il est beaucoup plus difficile de définir *notifyAll* avec des sémaphores, puisqu'on doit alors faire intervenir l'ensemble des processus en attente sur le sémaphore.

On peut enfin généraliser légèrement la notion de sémaphore, en utilisant un compteur. Un *sémaphore généralisé* est une variable s entière positive, sur laquelle on fait deux opérations atomiques suivantes :

- $P(s)$ teste la valeur de s. Si $s > 0$, on décrémente s de 1. Sinon on attend sur s.
- $V(s)$ réveille un processus en attente sur s, s'il existe un tel processus. Sinon s est incrémenté de 1.

6.10 Producteur – Consommateur

Les sémaphores sont des primitives de bas niveau. Toutefois, elles peuvent servir à programmer quelques exemples non triviaux, même si la bonne manière consiste presque toujours à utiliser des sections critiques en exclusion mutuelle avec des conditions.

L'exemple type est celui du producteur – consommateur. Un processus produit de l'information, et un autre la consomme. Le premier processus produit son résultat dans un medium (par exemple une file d'attente) dans lequel l'autre processus retire l'information. Comme déjà vu pour la file d'attente concurrente, il faut gérer les cas où l'on veut consommer dans un medium vide, et où l'on veut produire dans un medium plein. Supposons le medium infini, seul le cas vide importe. Alors la solution standard s'écrit de la manière suivante :

```
class FIFO {
  static final int N = 100000;
  int         debut, fin;
  int[]       contenu;
  Semaphore   s, s_occupees;

  FIFO (int n) {
    debut = fin = 0; contenu = new int[N];
    s_occupees = new Semaphore(0);
    s = new Semaphore(1);
  }

  void ajouter (int x) {
    Semaphore.P(s);
    contenu[fin++] = x;
    Semaphore.V(s);
    Semaphore.V(s_occupees);
  }

  int supprimer () {
    int res;
    Semaphore.P(s_occupees);
    Semaphore.P(s);
    res = contenu[debut++];
    Semaphore.V(s);
    return res;
  }
}
```

On peut aussi traiter le cas où la file est de longueur bornée avec un autre sémaphore généralisé comptant le nombre de cases vides dans la file d'attente. Les deux fonctions de manipulation de la file concurrente deviennent alors :

```
  FIFO (int n) {
    debut = fin = 0; contenu = new int[N];
    s_occupees = new Semaphore(0); s_libres = new Semaphore(N);
    s = new Semaphore(1);
  }

  void ajouter (int x) {
    Semaphore.P(s_libres);
    Semaphore.P(s);
    contenu[fin] = x;
    fin = (fin + 1) % N;
    Semaphore.V(s);
    Semaphore.V(s_occupees);
  }

  int supprimer () {
    int res;
    Semaphore.P(s_occupees);
    Semaphore.P(s);
```

```
    res = contenu[debut];
    debut = (debut + 1 ) % N;
    Semaphore.V(s);
    Semaphore.V(s_libres);
    return res;
  }
```

Exercice 43 Implanter les sémaphores généralisés en Java avec avec les verrous et conditions.

Exercice 44 Dans le langage Ada, la communication se fait par rendez-vous. Deux processus P et Q font un rendez-vous s'ils s'attendent mutuellement chacun à un point de son programme. On peut en profiter pour passer une valeur. Comment organiser cela en Java ?

Exercice 45 Une barrière de synchronisation entre plusieurs processus est un endroit commun que tous les processus actifs doivent franchir simultanément. C'est donc une généralisation à n processus d'un rendez-vous. Comment la programmer en Java ?

Exercice 46 Toute variable peut être partagée entre plusieurs processus, quel est le gros danger de la communication par variables partagées ?

Exercice 47 Un ordonnanceur est juste s'il garantit à tout processus prêt de passer dans l'état exécution. Comment le construire ?

Chapitre 7

Machines finies et infinies

Le caractère fini d'un programme est une notion qui se retrouve tant dans les propriétés de terminaison, que dans celles de puissance d'expressivité. Un programme est la description finie d'un calcul. Cette simple remarque est à la base de la théorie de la calculabilité. Supposons qu'on puisse avoir droit à un nombre infini d'instructions. On n'aurait pas attendu des siècles pour connaitre la réponse à la conjecture de Fermat. Il aurait suffit d'exécuter en parallèle toutes les équations $x^n + y^n = z^n$ pour tous les quadruplets d'entiers positifs (x, y, z, n) $(n > 2)$ en écrivant une astérisque sur l'imprimante si l'équation est vérifiée. Au bout d'un temps constant, si on ne voit rien écrit sur l'imprimante, la conjecture de Fermat est vraie. Pourtant, cette machine infinie semble irréaliste. Non seulement la taille du programme doit être finie, mais chaque donnée manipulée semble aussi devoir être bornée. Dans la vérification du théorème de Fermat, les nombres deviennent arbitrairement grands, et il semble illogique de donner un coût constant pour calculer la somme, l'exponentielle ou l'égalité de deux nombres arbitrairement grands. Enfin, la taille de la mémoire utilisée pour faire un calcul importe aussi. Peut-elle être infinie ? Finie non-bornée ? Doit-elle être bornée ? Ce sont ces questions que nous abordons dans ce chapitre, en considérant deux modèles de machines : les automates finis et les machines de Turing. Il ne s'agira que d'une première approche vers la théorie de la calculabilité, inventée par Gödel, Church, Turing et Kleene. Cette théorie a de multiples rapports avec la théorie des langages formels associée à l'analyse syntaxique, ou avec la logique mathématique utilisée dans les propriétés de correction des programmes.

7.1 Exemples de machines finies

Nous commençons par passer en revue quelques exemples de machines à nombre d'états fini.

Exemple 7.1 *Distributeur de café.* La machine de la figure 7.1 comporte deux fentes pour les pièces de 10 et 20 centimes, et un bouton K pour obtenir un café. Au début, le bouton K est bloqué, les deux fentes sont ouvertes. Le café coute 30 centimes. Si on a mis suffisamment de pièces, les fentes se ferment, le bouton K est libéré, on ne peut plus mettre de pièces, et on peut enfoncer le bouton K. Alors un gobelet tombe dans l'espace vide au bas de la machine, et le café se verse dans le gobelet. L'automatisme correspondant comporte 4 états. Dans les états 0, 1, 2, les fentes sont ouvertes, le bouton K est bloqué. Dans l'état 4, les fentes sont fermées, le bouton K est débloqué. Les états 0, 1, 2, 3 enregistrent le crédit courant en multiples de 10 centimes. On passe d'un état à un autre soit en insérant une pièce dans une des deux fentes, soit en appuyant sur le bouton K. Au début, le distributeur est dans l'état 0. Ceci est résumé par le schéma à

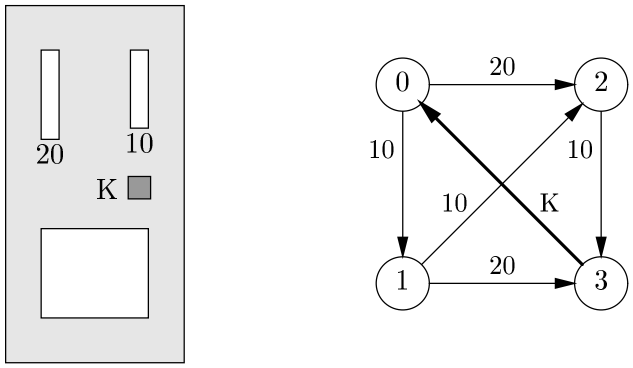


FIG. 7.1 – L'automate d'un distributeur de café

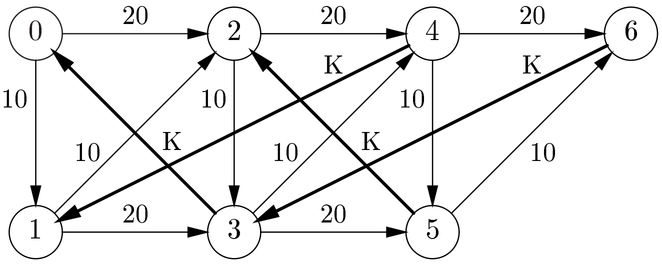


FIG. 7.2 – L'automate d'un distributeur de café avec mémoire

droite sur la figure 7.1.

Exemple 7.2 *Distributeur de café avec un tampon de longueur 2.* L'appareil précédent ne rend pas la monnaie. Il n'autorise pas de mettre plus de 30 centimes avant que le café ne soit versé. Nous l'améliorons légèrement. On peut maintenant obtenir jusqu'à deux cafés consécutivement à condition d'avoir mis 60 centimes. Pour aller plus vite, la machine libère le bouton K dès que suffisamment de pièces ont été introduites. Ainsi on peut mettre deux pièces de 20 centimes, appuyer sur K, puis remettre deux autres pièces de 20 centimes, et une de 10 centimes, puis appuyer deux fois de suite sur K, etc. Au total, notre automatisme a maintenant les 7 états décrits sur la figure 7.2.

Les mécanismes de nos deux machines à café sont simples, ils décrivent le fonctionnement d'un compteur sur un intervalle fini. L'intérêt de cette description est qu'on peut se contenter de l'automate pour chacune de ces machines. Ce n'est pas la peine de disposer d'un compteur. De simples dispositifs matériels peuvent réaliser notre automate. Il y a peu d'états, peu de transitions, le tout est bien sûr fini. On dit que l'automate décrivant notre machine est un automate fini. Beaucoup d'automatismes de la vie courante (feux rouges, barrières, etc) sont des automates finis.

Les circuits logiques sont un autre exemple de machines finies. Les circuits ont une partie combinatoire (enchaînements de portes pour calculer une fonction booléenne) et une partie de contrôle fini pour assurer la séquence entre les parties combinatoires. Par exemple, un processeur commence le traitement d'une instruction machine après avoir fini celui de celle qui la précède.

Exemple 7.3 *Additionneur binaire série.* A partir d'un additionneur sur 1-bit *ADD1*, on fabrique un additionneur série sur n bits. La porte *ADD1* a un fil d'entrée pour

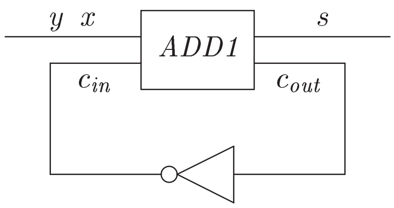


FIG. 7.3 – Additionneur série

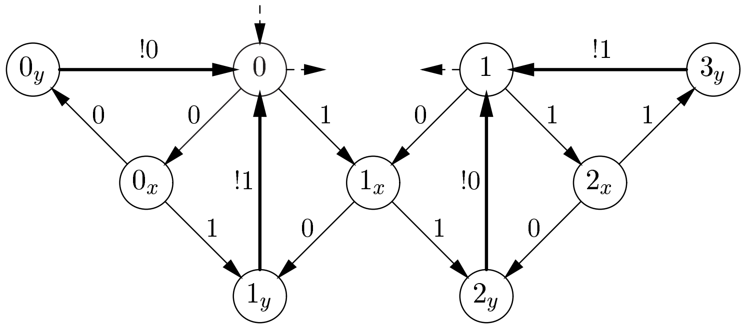


FIG. 7.4 – L'automate d'un additionneur série

les entreés x et y sérialisées, et un autre fil d'entrée pour la retenue entrante c_{in} et deux fils de sortie pour la somme résultat s et la retenue sortante c_{out}. Pour faire un additionneur sur n bits, la retenue de sortie reboucle sur l'entrée (après amplification), et on suppose le dispositif synchronisé sur une horloge pour faire d'abord entrer c_{in} et x, puis y, puis laisser sortir s et c_{out}, comme indiqué sur la figure 7.3.

Les données x et y arrivent entrelacées sur un seul fil d'entrée. L'additionneur réagit à des données de la forme $x_0y_0x_1y_1 \ldots x_ny_n$ où x_i et y_i sont les $(i+1)$-ème bit en partant de la droite dans la représentation binaire de x et de y. Le résultat produit est $s_0s_1 \ldots s_n$ représentant en binaire la somme s de x et de y, en sortant d'abord le bit de poids faible s_0, puis le suivant s_1, etc. Sur la figure 7.4, le fonctionnement de l'additionneur sur n bits est décrit par un petit automate. Au début, on est dans l'état 0. Si on a $x_0 = 0$, on évolue vers l'état 0_x, sinon on va en 1_x. Dans l'état 0_x, si on a $y_0 = 0$, on va en 0_y et alors on envoie 0 comme résultat pour s_0, et on revient dans l'état 0 pour lire x_1, puis y_1. On recommence le même raisonnement avec x_1 et y_1. Etc. Si dans l'état 0_x, on a $y_0 = 1$, on va en 1_y et on envoie 1 comme résultat pour s_0, et on revient dans l'état 0 pour aussi traiter x_1, et y_1. Etc.

Les états 0 et 1 sont des états où on lit un x_i avec la retenue courante valant respectivement 0 ou 1. Les états v_x et v_y signifient que la valeur v a été accumulée en additionnant la retenue courante à la valeur respectivement de x_i, ou de x_i et y_i. La transtion $!v$ signifie que v est la valeur de s_i.

La description du fonctionnement de cette machine est proche de celui fait pour la machine à café. Il consiste encore à faire un calcul sur un domaine à peu de valeurs. Il est aussi très simple car le séquencement de l'additionneur est trivial. Dans le cas de circuits plus réalistes (par exemple un processeur avec plusieurs niveaux de pipeline), le nombre d'états peut être beaucoup plus grand.

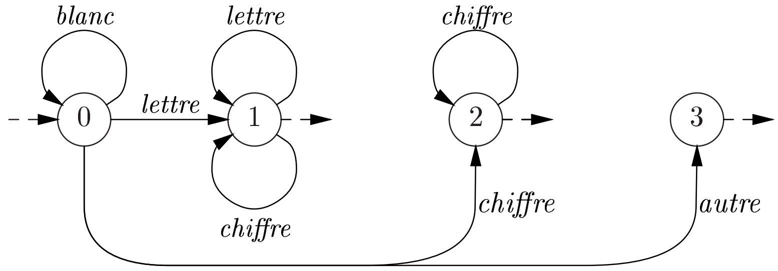


FIG. 7.5 – L'automate d'un analyseur lexical

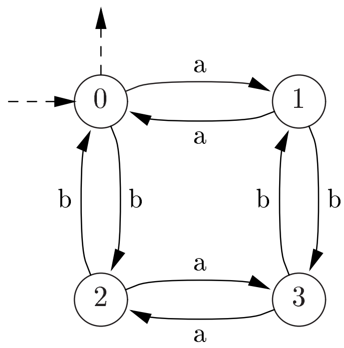


FIG. 7.6 – Mots contenant un nombre pair de a et de b

Exemple 7.4 *Analyseur lexical.* Dans la section 2.5, le code d'un analyseur lexical a été présenté. Il consistait à trouver dans un chaîne de caractères des lexèmes comme ceux correspondants aux identificateurs, aux nombres, ou opérateurs utilisés dans les expressions arithmétiques. Le code de notre analyseur correspond exactement à l'automate fini de la figure 7.5. Dans l'état initial, on saute tous les caractères blancs (espace, tabulation, retour ligne). Si on rencontre une lettre, c'est le début d'un identificateur, c'est-à-dire une suite de lettres et de chiffres. On génère le lexème dès que le caractère courant n'est plus ni une lettre, ni un chiffre. Dans l'état 2, on génère le lexème correspondant à un nombre. Dans l'état 3, il s'agit d'un opérateur.

Exemple 7.5 *Mots avec des nombres pairs de a et de b.* Dans cet exemple, plus académique, il s'agit de reconnaitre les mots sur l'alphabet $\{a, b\}$ contenant un nombre pair de a et de b. On considère l'automate de la figure 7.6. On démarre dans l'état 0 et on accepte un mot quand on revient à l'état 0 après avoir lu toutes ses lettres. Si on lit un a, on passe à l'état 1, si c'est un b on passe à l'état 2. L'état 1 signifie qu'on a lu un nombre impair de a et un nombre pair de b ; l'état 2 veut dire qu'on a lu un nombre pair de a et un nombre impair de b. Si dans l'état 1, on lit un a on revient à l'état 0 ; si dans l'état 1, on lit un b, on passe à un état 3 (nombre impair de a et nombre impair de b) ; etc. Remarquer que la chaîne vide est acceptée puisque contenant un nombre nul de a et de b.

7.2 Automate fini

Les exemples précédents sont des exemples d'automates finis. Cette structure a été définie dans les années 50 ; elle constitue le premier exemple de machine abstraite considérée dans ce chapitre. Un automate fini $\mathcal{A}$ est un quintuplet $(Q, \Sigma, \delta, q_0, F)$ où

1. Σ est l'alphabet d'entrée sur lequel sont construits les mots à reconnaitre,
2. Q est un ensemble *fini* d'états,
3. $q_0 \in Q$ est l'état initial,
4. $F \subset Q$ est l'ensemble des états de fin,
5. δ est une fonction $Q \times \Sigma \to Q$, dite fonction de transition.

L'automate est à tout moment dans un des états de Q ; au début il est dans l'état q_0. L'automate est une machine finie permettant de reconnaître des mots sur l'alphabet Σ. Pour un mot w de Σ^*, au début, l'automate lit la première lettre a_0 de w. Il passe donc dans l'état $q_1 = \delta(q_0, a_0)$. Puis il lit la deuxième lettre a_1 de w et il passe dans l'état $q_2 = \delta(q_1, a_1)$, etc., jusqu'à q_{n-1} après avoir lu les n lettres du mot w. Si $q_{n-1} \in F$, le mot w est accepté par l'automate. Sinon, le mot w est refusé.

Dans l'exemple des mots avec un nombre pair de a et de b. On a $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$, $F = \{0\}$, et δ donné par les formules suivantes :

$$\begin{array}{llll} \delta(0,a) = 1 & \delta(1,a) = 0 & \delta(2,a) = 3 & \delta(3,a) = 2 \\ \delta(0,b) = 2 & \delta(1,b) = 3 & \delta(2,b) = 0 & \delta(3,b) = 1 \end{array}$$

Formellement, la fonction de transition δ de type $Q \times \Sigma^* \to Q$ devient la fonction $\hat{\delta}$ de type $Q \times \Sigma^* \to Q$, définie récursivement sur w comme suit :

$$\hat{\delta}(q, \epsilon) = q \qquad \hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$$

Le langage $T(\mathcal{A})$ reconnu par l'automate $\mathcal{A}$ est défini par :

$$T(\mathcal{A}) = \{w \mid \hat{\delta}(q_0, w) \in F\}$$

Cette définition ne fait que formaliser la discussion précédente. A partir de q_0, on doit finir dans un état de fin dans F. Traditionnellement les automates sont représentés par un graphe dont les sommets sont les états, les arcs sont étiquetés par la lettre permettant la transition. Les états de fin sont marqués par des sommets dont partent des flèches tiretées vers l'extérieur. L'état initial est également marqué par une flèche tiretée incidente. Graphiquement, comme indiqué sur la figure 7.7, l'automate est une machine disposant d'une bande de lecture, d'une tête de lecture, d'un état courant q, et d'une fonction de transition δ. Au début la tête de lecture est sous le premier caractère de la bande à gauche. A chaque transition, en fonction de son état interne et du caractère sous la tête de lecture, il évolue vers un nouvel état en se déplacant d'un caractère vers la droite. L'automte se déplace vers la droite en lisant tous les caractères de la bande, et on regarde l'état final pour voir s'il est dans F. Remarquons que la bande de lecture est *non-modifiable*.

La représentation d'un automate fini peut se faire en dur dans un programme. Plusieurs manières existent : avoir une variable explicite pour l'état (comme dans la

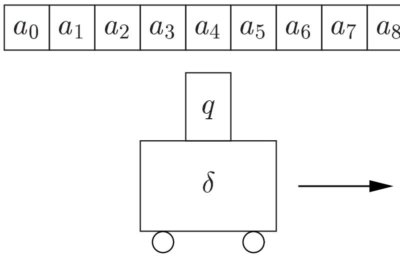


FIG. 7.7 – Un automate fini

fonction *accepter* suivante), faire coïncider le contrôle du programme avec l'état (certains points du programme correspondent aux valeurs de l'état, et on se branche à ces points du programme en fonction de l'état désiré), ou tabuler la fonction de transition comme dans la fonction *accepter1* suivante.

```
static boolean accepter (String w) {
  int n = w.length();
  int q = 0;
  for (int i = 0; i < n; ++i) {
    char c = w.charAt(i);
    switch (q) {
    case 0: if (c == 'a') q = 1;
       else if (c == 'b') q = 2;
       else return false;
       break;
    case 1: if (c == 'a') q = 0;
       else if (c == 'b') q = 3;
       else return false;
       break;
    case 2: if (c == 'a') q = 3;
       else if (c == 'b') q = 0;
       break;
    case 3: if (c == 'a') q = 2;
       else if (c == 'b') q = 1;
       else return false;
       break;
    }
  }
  return q == 0;
}

static boolean accepter1 (String w, int[ ][ ] delta) {
  int n = w.length();
  int q = 0;
  for (int i = 0; i < n; ++i) {
    char c = w.charAt(i);
    q = delta [q][c];
  }
  return q == 0;
}
```

Toutefois, la solution qui consiste à tabuler la fonction de transition pose souvent un problème de place mémoire. En Java, un caractère est représenté sur 16 bits, le tableau peut donc avoir $65536n$ éléments pour un automate de n états. Comme ce tableau est souvent creux, on a intérêt à le compresser. Dans notre exemple, c'est simple puisque seuls les caractères a et b importent, mais en général cela peut être plus ardu. Une

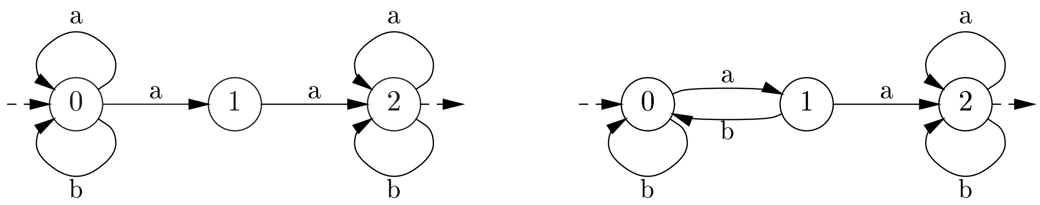


FIG. 7.8 – Mots contenant deux a consécutifs : (i) à gauche l'automate non-détermiste, (ii) à droite, l'automate déterministe.

solution potentielle consiste à faire des listes d'association pour chaque état, en adoptant une représentation proche de celle des graphes avec des tableaux de successeurs adoptée au chapitre 1.

7.3 Automates finis non-déterministes

La définition précédente des automates finis leur donne un comportement déterministe. A tout moment, une seule transition est possible en fonction de l'état q et de la lettre sous la tête de lecture a. Dans les automates non-déterministes, plusieurs transitions sont possibles.

Exemple 7.6 *Mots avec deux a consécutifs.* Il s'agit de reconnaitre les mots sur l'alphabet $\{a, b\}$ contenant un facteur aa. On les reconnait par l'automate de la figure 7.8(i). Dans l'état initial 0, si le mot w à reconnaitre commence par b, on reste dans l'état 0 ; si la première lettre de w est un a, on a le choix d'aller en 0 ou en 1. Dans l'état 1, on passe à l'état 2 avec un a ; avec un b, l'automate se bloque, aucune transition n'est possible. Etc. L'état 2 est le seul état de fin. Les mots acceptés sont tous ceux pour lesquels il existe une série de choix pouvant mener à l'état 2. Ce sont bien tous les mots contenant un facteur aa.

L'intérêt de la présentation non-déterministe des automates finis est que sa description est souvent plus succincte et naturelle qu'une version déterministe équivalente. Par exemple, les mots reconnus par l'automate non-déterministe de la figure 7.6(i) sont aussi reconnus par l'automate déterministe de la figure 7.6(ii). La présentation déterministe est légèrement plus compliquée, mais cela peut être plus compliquée encore comme dans l'exemple suivant.

Exemple 7.7 *Mots avec a en avant-dernière lettre.* Il s'agit de reconnaitre les mots sur l'alphabet $\{a, b\}$ décrits par l'expression régulière $(a+b)^*a(a+b)$. On les reconnait par les automates de la figure 7.9. On remarque que l'automate non-déterministe correspond exactement à l'expression régulière, alors que l'automate déterministe implémente un système de registre à retard pour accepter le mot dans les états 2 ou 3.

En fait, le nombre d'états devient encore plus grand si on cherche à reconnaitre les mots dont la n-ième lettre à partir de la fin est a.

Exercice 48 Trouver le nombre d'états pour l'automate déterministe reconnaissant les mots dont la 3-ième lettre à partir de la fin est a.

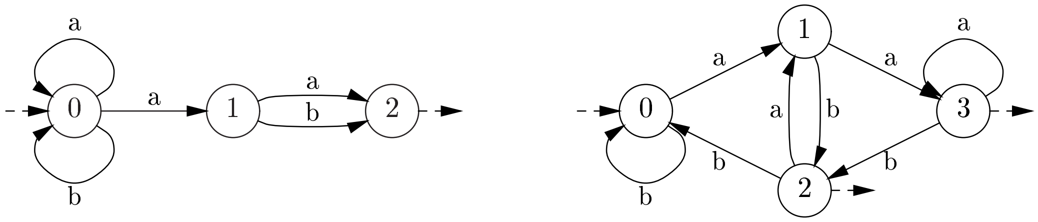


FIG. 7.9 – Mots contenant a en avant-dernière position : (i) à gauche l'automate non-détermiste, (ii) à droite, l'automate déterministe.

La définition formelle d'un automate fini non-déterministe $\mathcal{A}$ est pratiquement identique à celle des automates déterministes ; seule la fonction de transition diffère. Donc un automate fini non-déterministe est un quintuplet $(Q, \Sigma, \delta, q_0, F)$ où

1. Σ est l'alphabet d'entrée sur lequel sont construits les mots à reconnaitre,
2. Q est un ensemble *fini* d'états,
3. $q_0 \in Q$ est l'état initial,
4. $F \subset Q$ est l'ensemble des états de fin,
5. δ est une fonction $Q \times \Sigma \to 2^Q$ est la fonction de transition.

Dans l'automate de l'exemple de la figure 7.8, on a $\Sigma = \{a, b\}$, $Q = \{0, 1, 2\}$, $q_0 = 0$, $F = \{2\}$, et δ donné par les équations suivantes :

$$\begin{array}{lll} \delta(0, a) = \{0, 1\} & \delta(1, a) = \{2\} & \delta(2, a) = \{2\} \\ \delta(0, b) = \{0\} & \delta(1, b) = \emptyset & \delta(2, b) = \{2\} \end{array}$$

Comme dans le cas déterministe, la fonction de transition δ peut être étendue en la fonction $\hat{\delta}$ de type $Q \times \Sigma^* \to 2^Q$, définie récursivement sur w comme suit :

$$\hat{\delta}(q, \epsilon) = \{q\} \qquad \hat{\delta}(q, wa) = \{q'' \mid q' \in \hat{\delta}(q, w),\ q'' \in \delta(q', a)\}$$

Le langage $T(\mathcal{A})$ reconnu par l'automate $\mathcal{A}$ est défini par :

$$T(\mathcal{A}) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Pour écrire une fonction *ndAccepter* implémentant la fonction d'acceptation d'un automate fini non-déterministe, on peut écrire le programme non-déterministe suivant :

```
class NDAutomate {
  int q0;
  Liste[ ][ ] delta;
  Liste fin;
  NDAutomate (int q, Liste f, Liste[ ][ ]d) { q0 = q; delta = d; fin = f; }

  static boolean ndAccepter (String w, NDAutomate a) {
    int n = w.length();
    int q = a.q0;
    for (int i = 0; i < n; ++i) {
      int c = w.charAt(i);
      Liste e = a.delta [q][c];
      if (e == null) return false;
      int k = Math.max(1,
```

```
                 (int)(Math.ceil(Math.random() * Liste.longueur(e))));
      q = Liste.kieme(e, k);
    }
    return Liste.estDans(a.fin, q);
  }
}
```

qui utilise une fonction de tirage au sort pour choisir parmi les états possibles à chaque transition. On a également besoin des fonctions suivantes sur les ensembles d'états, réalisés ici par des listes :

```
class Liste {
  int val;
  Liste suivant;
  Liste (int v, Liste x) { val = v; suivant = x; }

  static int longueur (Liste x) {
    if (x == null) return 0; else return 1+longueur(x.suivant);
  }

  static int kieme (Liste x, int k) {
    if (k == 1) return x.val; else return kieme(x.suivant, k-1);
  }

  static boolean estDans (Liste x, int v) {
    if (x == null) return false;
    else return x.val == v || estDans(x.suivant, v);
  }
}
```

Si le résultat de *ndAccepter* est vrai, le mot donné en argument est bien accepté par l'automate. En revanche, si le résultat est faux, il n'est pas sûr qu'une exécution suivante ne puisse retourner vrai avec la même chaîne d'entrée. Mais, on peut aussi écrire la fonction suivante qui énumère tous les choix possibles et qui par énumération exhaustive donne une réponse exacte sur l'acceptation d'une chaîne de caractères par un automate fini non-déterministe.

```
static boolean ndAccepter (String w, Automate a) {
   return ndAccepterE (w, 0, a, a.q0);
}

static boolean ndAccepterE (String w, int i, Automate a, int q) {
  int n = w.length();
  if (i == n)
    return Liste.estDans(a.fin, q);
  else {
    boolean res = false;
    int c = w.charAt(i);
    for (Liste e = a.delta [q][c]; e != null; e = e.suivant) {
      int q1 = e.val;
      res = res || ndAccepterE (w, i+1, a, q1);
    }
    return res;
  }
}
```

Conformément aux méthodes exposées dans le chapitre 4, les appels récursifs de la fonction *ndAccepterE* consistent à faire des retours en arrière (*backtracking*) sur la chaîne d'entrée et revenir sur un choix différent.

Il existe bien d'autres formulations pour les automates finis en dehors des deux principales que nous venons de voir. En voici quelques unes qui s'obtiennent par variations sur la définition de la fonction δ de transition :

1. $\delta : Q \times \Sigma \cup \{\epsilon\} \to 2^Q$
   L'automate non-déterministe avec transitions vides peut à tout moment, dans l'état q avec la lettre a sous la tête de lecture, choisir entre faire une des transitions données par $\delta(q,a)$, ou d'abord faire une transition vide qui change spontanément l'état en un état de $\delta(q,\epsilon)$. Formellement

$$\begin{array}{l}\hat{\hat{\delta}}(q,\epsilon) = \{q\} \ \cup \ \delta(q,\epsilon) \\ \hat{\delta}(q,uav) = \{q_3 \mid q_1 \in \hat{\delta}(q,u),\ q_2 \in \delta(q',a),\ q_3 \in \hat{\delta}(q_2,v)\}\end{array}$$

2. $\delta : Q \times \Sigma^* \to 2^Q$ défini sur un sous-ensemble *fini* de Σ^*
   L'automate fini non-déterministe avec transitions composées peut à tout moment choisir une des transitions données par $\delta(q,u)$ si la lettre sous la tête de lecture est la première du mot u dans le mot à analyser. Bien remarquer qu'on exige que le nombre de transitions décrites par δ est fini. Formellement

$$\begin{array}{l}\hat{\hat{\delta}}(q,\epsilon) = \{q\} \ \cup \ \delta(q,\epsilon) \quad \text{si } \delta(q,\epsilon) \text{ est défini} \\ \hat{\delta}(q,\epsilon) = \{q\} \quad \text{sinon} \\ \hat{\delta}(q,vuw) = \{q_3 \mid q_1 \in \hat{\delta}(q,v),\ q_2 \in \delta(q_1,u),\ q_3 \in \hat{\delta}(q_2,w)\}\end{array}$$

3. $\delta : Q \times \Sigma \to 2^{Q \times \{gauche, droite\}}$
   L'automate fini non déterministe avec déplacement dans les 2 sens fonctionne comme un automate fini non-déterministe, sauf qu'après avoir lu une lettre, il peut se déplacer d'une case vers la droite, ou d'une case vers la gauche pour relire un caractère qu'il a déja lu. Si le déplacement vers la gauche n'est pas possible, l'automate se bloque. La condition d'arrêt est toujours que l'état obtenu soit dans F, mais il se peut qu'on n'ait pas lu tout le mot à reconnaitre. Formellement, nous considérons les configurations (u,q,v) où u est le mot (strictement) à gauche de la tête de lecture, v le mot à droite, et q l'état courant, ainsi que la relation binaire associée $\longrightarrow$ sur les configurations définie par

$$\begin{array}{rcl}(q', droite) \in \delta(q,a) & \Rightarrow & (u,q,av) \longrightarrow (ua,q',v) \\ (q', gauche) \in \delta(q,a) & \Rightarrow & (ub,q,av) \longrightarrow (u,q',bav)\end{array}$$

   Si $\xrightarrow{*}$ est la fermeture réflexive et transitive de $\longrightarrow$, alors les mots acceptés par un tel automate sont les mots de

$$T(\mathcal{A}) = \{w \mid (\epsilon,q_0,w) \xrightarrow{*} (u,q,v),\ q \in F\}$$

On peut donc se poser le problème de l'expressivité de ces diverses définitions. En fait, on peut montrer qu'elles sont toutes équivalentes. Un ensemble de mots reconnu par l'une est aussi reconnaissable par l'autre. Nous laissons ces démonstrations en exercice. On pourra donc choisir entre ces définitions d'automates finis non-déterministes en fonction de la commodité d'utilisation.

7.4 Les trois théorèmes fondamentaux

La théorie des automates finis est riche. Nous ne ferons que l'aborder à travers les trois théorèmes suivants.

Théorème 7.1 (Rabin, Scott) *Les langages reconnus par les automates finis déterministes sont exactement ceux reconnus par les automates finis non-déterministes.*

Démonstration Trivialement les mots reconnus par un automate déterministe le sont aussi par un automate non-déterministe, puique les automates finis déterministes forment un cas particulier des automates non-déterministes. Réciproquement, si $\mathcal{A}$ est un automate fini non-déterministe $(Q, \Sigma, \delta, q_0, F)$, considérons l'automate

$$\mathcal{A}' = (2^Q, \Sigma, \delta', q_0, F')$$

où 2^Q est l'ensemble des parties de Q, où l'ensemble de fin F' vérifie

$$F' = \{Q \mid Q \cap F \neq \emptyset)$$

et la fonction de transition δ' est définie, pour tout $a \in \Sigma$ et tout $E \subset Q$, par

$$\delta'(E, a) = \bigcup_{q \in E} \delta(q, a)$$

Alors on montre facilement que les mots acceptés par $\mathcal{A}$ le sont aussi par l'automate déterministe $\mathcal{A}'$. $\square$
Comme vu dans les exemples précédents, le nombre d'états peut être plus grand dans la version déterministe. En fait, on peut montrer qu'il peut devenir exponentiel. Le théorème précédent indique simplement que le non-déterminisme n'introduit pas un nouvel ensemble de langages reconnus. Toutes les définitions précédentes d'automates finis sont donc équivalentes.

Nous énonçons deux autres théorèmes sans démonstration.

Théorème 7.2 (Myhill, Nerode) *Tout langage reconnu par un automate fini est reconnu par un automate déterministe minimal unique à un isomorphisme près sur le nom des états.*

Ce théorème dit que, pour tout langage reconnu par un automate fini, il existe un automate déterministe minimal canonique le reconnaissant. On peut donc déterminiser un automate non-déterministe et calculer ensuite l'automate minimal correspondant. Remarquons que le théorème n'est pas vrai pour les automates non-déterministes.

Théorème 7.3 (Kleene) *Les langages reconnus par un automate fini sont exactement ceux décrits par les expressions régulières.*

Ce théorème, attribué un peu rapidement au promoteur des expressions régulières, Kleene, connecte la notion d'expression régulière et la notion d'automates finis. Ainsi les langages reconnus par des automates finis sont aussi appelés *langages réguliers*. On comprend donc mieux les analyseurs lexicaux du chapitre 2. Les programmes calculant les lexèmes décrits à partir d'expressions régulières étaient en fait des implémentations d'un automate fini (déterministe) les calculant. Le contrôle de ces programmes représentaient les états de cet automate. La mémoire utilisée pour calculer les lexèmes ne dépassait pas celle nécessaire pour représenter les états de ces automates ; elle était donc constante, ou encore en $O(1)$.

$\underline{a}aabbb$	$X\underline{a}aYbb$	$XX\underline{a}YYb$	$XXX\underline{Y}YY$
$X\underline{a}abbb$	$XX\underline{a}Ybb$	$XXX\underline{Y}Yb$	$XXXY\underline{Y}Y$
$Xa\underline{a}bbb$	$XXa\underline{Y}bb$	$XXXY\underline{Y}b$	$XXXYY\underline{Y}$
$Xaa\underline{b}bb$	$XXaY\underline{b}b$	$XXXYY\underline{b}$	$XXXYYY\underline{B}$
$Xa\underline{a}Ybb$	$XXa\underline{Y}Yb$	$XXXY\underline{Y}Y$	$XXXYYYB\underline{B}$
$X\underline{a}aYbb$	$XX\underline{a}YYb$	$XXX\underline{Y}YY$	
$\underline{X}aaYbb$	$X\underline{X}aYYb$	$XX\underline{X}YYY$	

FIG. 7.10 – Etats successifs de la bande d'une machine de Turing lors de la reconnaissance du mot a^3b^3.

Exercice 49 Trouver un automate fini non-déterministe de n états dont la version déterministe a 2^n états.

Exercice 50 Montrer que, si on change la définition d'automate fini (déterministe ou non-déterministe) pour autoriser plusieurs états initiaux, on reconnait encore le même ensemble de mots.

7.5 Machines de Turing

En 1936, Turing a défini une puissante notion de machine étonnament simple. Il s'agit de pratiquement la même définition que celle des automates finis (fonctionnant dans les deux sens), avec une différence importante : on peut écrire sur la bande de lecture.

Exemple 7.8 *Mots a^nb^n.* Le langage à reconnaitre est l'ensemble des mots de la forme a^nb^n ($n \geq 0$). Avec un automate fini, on peut montrer que c'est impossible. Avec une machine de Turing, on écrit sur la bande pour compter les nombres relatifs de a et de b. On suppose également une infinité de caractères blancs B derrière le mot à reconnaître. La machine de Turing a le fonctionnement suivant : on remplace le premier a par un X, puis on cherche le premier b à droite, on le remplace par un Y. On repart à gauche jusqu'à un X. Si juste à sa droite, on trouve un a, on le remplace à nouveau par un X et on repart à droite à la recherche d'un b, qu'on remplace par un Y, puis on repart à gauche. Etc. Si juste à droite de X, on avait trouvé un Y, on doit s'assurer qu'il n'y a plus que des Y sur la droite jusqu'au premier caractère blanc. Si c'est le cas, on accepte le mot ; sinon on le refuse. Graphiquement, la figure 7.10 résume les états successifs que prend la bande. La tête de lecture est sous la lettre soulignée.

Plus précisément, une machine de Turing est un 7-uplet $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ où :

1. Σ est un alphabet d'entrée,
2. Γ est un alphabet des symboles de la bande ($\Sigma \subset \Gamma$),
3. $B \in \Gamma - \Sigma$ est le symbole blanc,
4. Q est un ensemble *fini* d'états,
5. $q_0 \in Q$ est l'état initial,
6. $F \subset Q$ est l'ensemble des états de fin,

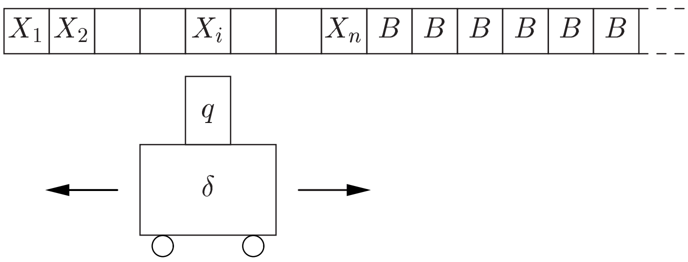


FIG. 7.11 – Configuration d'une machine de Turing.

7. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{gauche, droite\}$ est la fonction de transition.

Le fonctionnement peut se deviner à partir de celui des automates finis déterministes. Le deuxième résultat renvoyé par δ est le symbole qu'on écrit sous la tête de lecture, avant de faire le mouvement indiqué par le troisième résultat. La bande de lecture (et d'écriture) est supposée infinie vers la droite. Elle a une portion finie à gauche non blanche. A partir d'une certaine case, la bande est constamment blanche vers la droite. Au début, la machine est dans l'état q_0, la tête de lecture est sur la première lettre d'un mot w placé sur la partie gauche de la bande. La machine accepte tout mot dès qu'un état de fin est atteint. (Tous les caractères de w ne sont pas forcément lus.) Formellement, un configuration est un triplet (u, q, v) où

- $q \in Q$ est l'état courant,
- $u = X_1 X_2 \dots X_{i-1} \in \Gamma^*$ est le mot strictement à gauche de la tête de lecture,
- $v = X_i X_{i+1} \dots X_n \in \Gamma^*$ est le mot à droite de la tête de lecture jusqu'au caractère non blanc le plus à droite.

Graphiquement, une configuration est illustrée sur la figure 7.11. Les transitions sont définies par la relation binaire suivante sur les configurations :

$$\begin{array}{rcl}
\delta(q, X) = (q', Y, droite) & \Rightarrow & (u, q, Xv) \longrightarrow (uY, q', v) \\
\delta(q, B) = (q', Y, droite) & \Rightarrow & (u, q, \epsilon) \longrightarrow (uY, q', \epsilon) \\
\delta(q, X) = (q', Y, gauche) & \Rightarrow & (uZ, q, Xv) \longrightarrow (u, q', ZYv) \\
\delta(q, B) = (q', Y, gauche) & \Rightarrow & (uZ, q, \epsilon) \longrightarrow (u, q', ZY)
\end{array}$$

Le langage reconnu par la machine $\mathcal{M}$ est le sous-ensemble de mots

$$T(\mathcal{M}) = \{w \mid w \in \Sigma^*,\ (\epsilon, q_0, w) \stackrel{*}{\longrightarrow} (u, q, v),\ q \in F\}$$

Un langage reconnu par une machine de Turing est dit *récursivement énumérable* pour des raisons que nous ne pouvons expliquer ici.

Pour l'exemple 7.8, on considère la machine de Turing $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ où $\Sigma = \{a, b\}$, $\Gamma = \{a, b, X, Y, B\}$, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, q_0 est l'état initial, $F = \{q_4\}$, et la fonction de transition δ est décrite par le tableau à double entrée suivant :

	a	b	X	Y	B
q_0	$(q_1, X, droite)$			$(q_3, Y, droite)$	
q_1	$(q_1, a, droite)$	$(q_2, Y, gauche)$		$(q_1, Y, droite)$	
q_2	$(q_2, a, gauche)$		$(q_0, X, droite)$	$(q_2, Y, gauche)$	
q_3				$(q_3, Y, droite)$	$(q_4, B, droite)$
q_4					

Le principe de fonctionnement d'une machine de Turing est donc très simple : le nombre d'états est fini, on a une fonction de transition entre ces états avec la bande servant de mémoire annexe. Cela fait une grande différence par rapport aux automates finis, puisque la taille mémoire utilisée est a priori non bornée. Dans le cas de $a^n b^n$, on se sert d'une mémoire en $O(n)$. Tout le coté fini de la description d'un calcul est résumé dans les machines de Turing : états, alphabet d'entrée et de symbole de bande, partie non blanche de la bande. Pourtant, la programmation avec des machines de Turing est fastidieuse. Une des raisons provient de l'accès séquentiel aux données rangées en mémoire (c'est-à-dire sur la bande). On est donc en droit de se demander si en rajoutant des instructions, on va pouvoir calculer plus de fonctions et avec plus de confort.

Comme pour les automates finis, on peut écrire la fonction *accepter* d'acceptation d'un mot w par une machine $\mathcal{M}$. On remarque qu'à la différence des automates finis, la bande est maintenant décrite par une chaîne de caractères *modifiable* de la classe *StringBuffer*. On note aussi que sa taille est augmentée dynamiquement grâce à la méthode *append* de la classe *StringBuffer*. On suppose également que cette opération est toujours possible, et qu'on dispose donc d'une mémoire de taille non bornée.

```
class Turing {
  final static int GAUCHE = 0, DROITE = 1;
  final static char BLANC = 'B';
  int q0;
  Liste fin;
  Action[ ][ ] delta;

  static boolean accepter (String w, Turing m) {
    StringBuffer bande = new StringBuffer (w);
    int tete = 0;
    int q = m.q0;
    while ( !Liste.estDans(m.fin, q) ) {
      char c = tete < bande.length() ? bande.charAt(tete) : BLANC;
      Action a = m.delta [q][c];
      if (a == null) return false;
      q = a.q;
      if (a.dept == GAUCHE && tete > 0)
        bande.setCharAt(tete--, a.c);
      else if (a.dept == DROITE && tete < bande.length())
        bande.setCharAt(tete++, a.c);
      else if (a.dept == DROITE && tete == bande.length()) {
        bande.append(a.c);
        ++tete;
      } else return false;
    }
    return true;
  }

class Action { int q; char c; int dept; }
```

Exercice 51 Donner une machine de Turing qui reconnaisse les palindromes w formés de a et de b tels que $w = w^R$ où w^R est l'image miroir de w.

Exercice 52 Trouver une machine de Turing qui reconnaisse les mots bien parenthésés formés de a et de b.

Exercice 53 Donner une machine de Turing qui calcule le successeur $x + 1$ de tout nombre binaire x.

Exercice 54 Donner une machine de Turing qui calcule la somme $x+y$ de tous nombres binaires x et y.

Exercice 55 Soit $\Sigma(n)$ le nombre maximal d'étapes pris par une machine de Turing (déterministe) à n états avant de s'arréter en partant d'une bande complètement blanche. Calculer $\Sigma(n)$ pour $n \leq 4$ (le castor affairé de Rado).

7.6 Autres définitions de machines de Turing

Plusieurs définitions alternatives existent pour les machines de Turing. Nous en considérons quelques unes :

1. $\delta : Q \times \Gamma \to 2^{Q\times\Gamma\times\{gauche,droite\}}$
   Les machines de Turing non-déterministes peuvent évoluer, dans tout état, au choix avec plusieurs transitions possibles. Formellement,

$$\begin{array}{rcl}
(q', Y, droite) \in \delta(q, X) & \Rightarrow & (u, q, Xv) \longrightarrow (uY, q', v) \\
(q', Y, droite) \in \delta(q, B) & \Rightarrow & (u, q, \epsilon) \longrightarrow (uY, q', \epsilon) \\
(q', Y, gauche) \in \delta(q, X) & \Rightarrow & (uZ, q, Xv) \longrightarrow (u, q', ZYv) \\
(q', Y, gauche) \in \delta(q, B) & \Rightarrow & (uZ, q, \epsilon) \longrightarrow (u, q', ZY)
\end{array}$$

2. Machines de Turing avec la bande infinie à gauche et à droite. Les transitions sur les configurations sont alors définies par :

$$\begin{array}{rcl}
\delta(q, X) = (q', Y, droite) & \Rightarrow & (u, q, Xv) \longrightarrow (uY, q', v) \\
\delta(q, B) = (q', Y, droite) & \Rightarrow & (u, q, \epsilon) \longrightarrow (uY, q', \epsilon) \\
\delta(q, X) = (q', Y, gauche) & \Rightarrow & (uZ, q, Xv) \longrightarrow (u, q', ZYv) \\
\delta(q, B) = (q', Y, gauche) & \Rightarrow & (uZ, q, \epsilon) \longrightarrow (u, q', ZY) \\
\delta(q, X) = (q', Y, gauche) & \Rightarrow & (\epsilon, q, Xv) \longrightarrow (\epsilon, q', BYv) \\
\delta(q, B) = (q', Y, gauche) & \Rightarrow & (\epsilon, q, \epsilon) \longrightarrow (\epsilon, q', BY)
\end{array}$$

3. Machines de Turing avec transitions immobiles.
   $\delta : Q \times \Gamma \to Q \times \Gamma \times \{gauche, immobile, droite\}$ Les transitions sont définies par la relation binaire suivante sur les configurations :

$$\begin{array}{rcl}
\delta(q, X) = (q', Y, droite) & \Rightarrow & (u, q, Xv) \longrightarrow (uY, q', v) \\
\delta(q, B) = (q', Y, droite) & \Rightarrow & (u, q, \epsilon) \longrightarrow (uY, q', \epsilon) \\
\delta(q, X) = (q', Y, gauche) & \Rightarrow & (uZ, q, Xv) \longrightarrow (u, q', ZYv) \\
\delta(q, B) = (q', Y, gauche) & \Rightarrow & (uZ, q, \epsilon) \longrightarrow (u, q', ZY) \\
\delta(q, X) = (q', Y, immobile) & \Rightarrow & (u, q, Xv) \longrightarrow (u, q', Yv) \\
\delta(q, B) = (q', Y, immobile) & \Rightarrow & (u, q, \epsilon) \longrightarrow (u, q', Y)
\end{array}$$

4. Machines avec n bandes. $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q^0, B, F)$ est construit à partir de n machines $\mathcal{M}_k = (Q_k, \Sigma, \Gamma, \delta_k, q_k^0, B, F_k)$ à une seule bande en considérant :
   (a) $Q = Q_1 \times Q_2 \times \ldots Q_n$,
   (b) $\delta(q_1, q_2, \ldots q_n) = (\delta_1(q_1), \delta_2(q_2), \ldots \delta_n(q_n))$,
   (c) $q^0 = q_1^0 \times q_2^0 \times \ldots q_n^0$,
   (d) $F = F_1 \times F_2 \times \ldots F_n$,

Au début, le mot w à droite de la première tête de lecture, et les autres bandes sont blanches. Une n-configuration est un n-uplet de configurations $(c_1, c_2, \dots c_n)$ où c_k est une configuration de $\mathcal{M}_k$ pour $1 \leq k \leq n$. La relation de transition devient

$$(c_1, c_2, \dots c_n) \longrightarrow (c'_1, c'_2, \dots c'_n)$$

où $c_k \longrightarrow_k c'_k$ et $\longrightarrow_k$ est la relation de transition sur les configurations de $\mathcal{M}_k$.

5. Machines de Turing avec transitions composées.
   $\delta : Q \times \Gamma^* \to 2^{Q \times \Gamma^* \times \{gauche, droite\}}$ défini sur un sous-ensemble *fini* de Γ^*
   C'est une machine non déterministe dont les transitions sont exactement les mêmes que dans le cas non-déterministe en remplaçant partout X par un mot non vide $x \in \Gamma^+$ et Y par un mot $y \in \Gamma$.

On peut montrer que toutes ces définitions sont équivalentes. Donc si un mot w est accepté par l'une de ces machines, il l'est aussi par une machine d'une autre sorte. Ces équivalences sont anecdotiques ; leurs démonstrations sont intéressantes pour apprendre la technique de manipulation des machines de Turing. Philosophiquement, il apparait que le modèle de machine de Turing résiste bien aux extensions.

Exercice 56 Ecrire la fonction *accepter* d'acceptation d'un mot par une machine de Turing non-déterministe (selon la première définition de machine non-déterministe).

Exercice 57 Montrer que les 5 définitions de machines de Turing considérées reconnaissent les mêmes langages.

7.7 Thèse de Church

Dans les années 1920-1930, à la suite des résultats de Gödel, plusieurs modèles de la calculabilité ont été introduits : le λ-calcul de Church, les fonctions récursives de Kleene, les machines de Turing, les systèmes de réécritures de Post. Ces modèles, quoique très différents dans leur présentation, ont tous été montrés équivalents. Ce qu'on calculait dans le calcul de Church pouvait être aussi calculé avec les machines de Turing, ou s'exprimait aussi avec une fonction récursive à la Kleene, et réciproquement. Church a donc émis le postulat suivant :

Thèse 7.1 (Church) *Tous les modèles de la calculabilité sont équivalents.*

Cette hypothèse, parfois aussi attribuée à Turing (élève de Church), stipule qu'il y a un modèle unique de la calculabilité. Ce que l'on peut calculer avec une machine de Turing peut aussi être calculé en Java sur un PC ; tout ce qui peut être calculé sur un PC peut être calculé sur un Mac ; tout ce qu'on peut calculer en Java peut être aussi calculé en Ocaml, et réciproquement. Quelque soit le langage de programmation (Java, ML, Ocaml, Ada), on calcule les mêmes fonctions. Quelquesoit la machine support (PC, Mac, Alpha, HP), on calcule les mêmes fonctions. Le modèle unique de la calculabilité repose sur des hypothèses fortes de représentation finie du calcul, et sur rien de plus.

Pourtant, ce qui est calculé avec un automate fini est strictement moindre que ce qu'on peut faire avec une machine de Turing. Nous avons cité l'exemple du langage $\{a^n b^n \mid n \geq 0\}$ reconnaissable par une machine de Turing, mais pas par un automate fini. Bien d'autres exemples existent. La raison profonde provient de l'espace mémoire borné seulement disponible avec un automate fini. En effet, à partir d'un certain n, il sera impossible de se souvenir du nombre de a rencontrés dans le mot $a^n b^n$, et donc impossible de compter le nombre identique de b. Une machine de Turing dispose d'un espace mémoire non-borné grâce à sa bande (réinscriptible). C'est cela qui donne plus de puissance à une machine de Turing.

Traditionnellement, on utilise la thèse de Church en informatique sans montrer dans les détails qu'un langage de programmation ou un modèle de machine peut être codé en une machine de Turing. Si le raisonnement est parfaitement exact au niveau d'un langage de programmation puisqu'on peut toujours le compiler vers du code exécutable par une machine de Turing, on peut plus douter de l'argument pour comparer un ordinateur moderne à une machine de Turing. Un ordinateur dispose-t-il d'une mémoire non bornée ? Les réponses peuvent diverger : les pragmatiques diront que la mémoire centrale est bornée, les idéalistes diront qu'on peut toujours rajouter de la mémoire externe (disques, bandes), autant que nécessaire, les ultra-sceptiques diront que le nombre d'atomes de l'univers est borné et qu'on ne peut donc disposer d'une mémoire arbitrairement grande. En règle générale, s'il est vrai qu'on peut comparer un ordinateur à un automate fini, le nombre d'états sera très grand et une bonne approximation consiste à prendre le modèle d'une machine de Turing.

Exercice 58 Montrer qu'une machine de Turing qui n'écrit que sur une partie bornée de la bande est équivalente à un automate fini.

Si nous avons vu qu'un modèle très simple suffisait pour faire tous les calculs imaginables, il n'est absolument pas garanti que l'écriture de programmes soit commode dans le modèle simplifié. Au contraire, on a besoin de langages de programmation de haut-niveau pour avoir une écriture plus concise. Par ailleurs, on peut se demander si on sait calculer toute fonction des mathématiques, ou s'il existe des fonctions non-calculables.

Depuis Gödel, nous savons qu'il existe des objets non-calculables. Un argument savant consiste à dire qu'il existe un nombre dénombrable ($\aleph_0$) de machines de Turing, alors qu'il y a un bien plus grand nombre ($\aleph_1$) de langages sur l'alphabet Σ. Un tout simple argument par diagonalisation donne ausi la solution. Comme les programmes ont une description finie, on peut les énumérer, c'est-à-dire trouver une fonction ϕ de $\mathbf{N}$ dans les machines de Turing tel que $\phi(i) = \mathcal{M}_i$ est la i-ème machine de Turing. De même, on peut énumérer tous les mots sur un alphabet Σ avec une fonction $\psi(i) = w_i$ est le i-ème mot. On peut donc parler de la i-ème machine et du i-ème mot. Considérons l'ensemble

$$L = \{\, w_i \ \mid \ w_i \notin T(\mathcal{M}_i)\,\}$$

Alors L ne peut être reconnu par une machine de Turing. Sinon, il existerait une machine $\mathcal{M}_k$ telle que $L = T(\mathcal{M}_k)$, et on aurait $w_k \in T(\mathcal{M}_k)$ si et seulement si $w_k \notin T(\mathcal{M}_k)$. Contradiction.

Théorème 7.4 (Gödel) *Il existe des langages non reconnaissables par une machine de Turing.*

Supposons que nous disposions d'une représentation $\underline{\mathtt{n}}$ pour tout entier naturel n. Par exemple, $\underline{\mathtt{n}}$ sera la chaîne de caractères en représentation décimale donnée par l'expression *Integer.toString(n)* de Java. On peut alors parler de fonctions calculables sur $\mathbf{N}$ par machines de Turing. En effet, nous considérerons les fonctions f_i définies par

$$f_i(n) = m \qquad ssi \qquad (\epsilon, q_0, \underline{\mathtt{n}}) \xrightarrow{*} (\underline{\mathtt{m}}, q, v) \quad (q \in F)$$

pour la i-ème machine $\mathcal{M}_i = (Q, \Sigma, q_0, F, \delta)$. Au début, on met $\underline{\mathtt{n}}$ sur la bande, et, à la fin (si le calcul termine),on ne considère le résultat comme un entier que si une représentation d'entier $\underline{\mathtt{m}}$ figure à gauche de la tête de lecture. Si ce n'est pas le cas, la fonction n'est pas définie pour n. La fonction f_i n'est donc pas forcément définie pour tout n, soit parce que la machine de Turing correspondante ne termine pas, soit parce qu'elle retourne un résultat non entier sur sa bande. Au total, la fonction f_i est donc une fonction calculable de type $\mathbf{N} \to \mathbf{N}$. Par la thèse de Church, ce sont aussi toutes les fonctions calculées en Java par une fonction de signature :

```
static int f (int n) { ... }
```

(en se restreignant aux arguments et résultats positifs). A nouveau, il existe plein de fonctions de $\mathbf{N}$ dans $\mathbf{N}$ non calculables, puisque le nombre de programmes écrits en Java est dénombrable. Plus généralement, on peut aussi définir des fonctions calculables sur d'autres domaines que $\mathbf{N}$, mais la théorie demande plus d'habileté.

Exercice 59 Montrer rigoureusement qu'il existe des fonctions sur $\mathbf{N}$ non calculables.

7.8 Indécidabilité de l'arrêt

Parmi les fonctions non calculables, la fonction décidant de l'arrêt est célèbre. Il s'agit de tester si une machine $\mathcal{M}$ s'arrête avec un mot w donné sur la bande dans l'état initial. Pour les fonctions calculables, cela revient à trouver si $f(n)$ est défini pour un n donné. Turing a montré que c'était impossible. Faire la démonstration avec les machines de Turing est un peu délicate, car faisant appel à des notions avancées de la théorie de la calculabilité. Nous en considérons une version simplifiée sur des programmes Java.

Ecrivons $e\downarrow$ et $e'\uparrow$ pour signifier que l'évaluation de l'expression e termine et que celle de e' ne termine pas. Supposons qu'il existe une fonction *termine* qui prend un objet o en argument et teste si sa méthode f (sans arguments) termine. On a donc

$$Turing.termine(o) = \begin{cases} true & \text{si} \quad o.f(\,)\downarrow \\ false & \text{si} \quad o.f(\,)\uparrow \end{cases}$$

Dans le programme suivant, où le code de *termine* n'est pas précisé, l'impression finale donne donc *true* pour o_1 et *false* pour o_2, puisque $o_1.f()\downarrow$ et $o_2.f()\uparrow$.

```
class Turing {

  static boolean termine (Object o) {
    // La valeur retournée vaut true si o.f()
    // termine. Sinon la valeur retournée vaut false.
    // (le code n'est pas public car breveté par le vendeur)
} }

class Facile {
  void f () { }
}

class Boucle {
  void f () {
    for (int i = 1; i > 0; ++i)
      ;
} }
```

```
class Absurde {
  void f () {
    while (Turing.termine(this))
      ;
} }

class Test {
  public static void main (String[ ] args) {
    Facile o1 = new Facile();
    System.out.println (Turing.termine(o1));
    Boucle o2 = new Boucle();
    System.out.println (Turing.termine(o2));
    Absurde o3 = new Absurde();
    System.out.println (Turing.termine(o3));
} }
```

Pour o_3 le problème est plus complexe. En effet

$$Turing.termine(o_3) = true \quad \text{ssi} \quad o_3.f() \downarrow \quad \text{ssi} \quad Turing.termine(o_3) = false$$

Contradiction ! La fonction *termine* n'existe pas. Il peut sembler que la difficulté provient de l'impossibilité de regarder avec attention le code de la méthode f, puisqu'on ne peut que l'appliquer dans nos programmes. En fait, même en disposant du code de f, on ne peut toujours pas décider de la terminaison. L'argument peut être conduit rigoureusement sur les machines de Turing. Nous ne faisons que suggérer l'énoncé de l'indécidabilité de l'arrêt.

Théorème 7.5 *Il n'existe pas de fonction calculable h telle que pour tout i et n, on a $h(i,n) = 1$ si $f_i(n)$ est défini, et $h(i,n) = 0$ si $f_i(n)$ n'est pas défini.*

La théorie de la calculabilité est amusante. Elle apporte principalement des résultats d'impossibilité. Il y a pourtant quelques résultats positifs. En voici un donné en exercice.

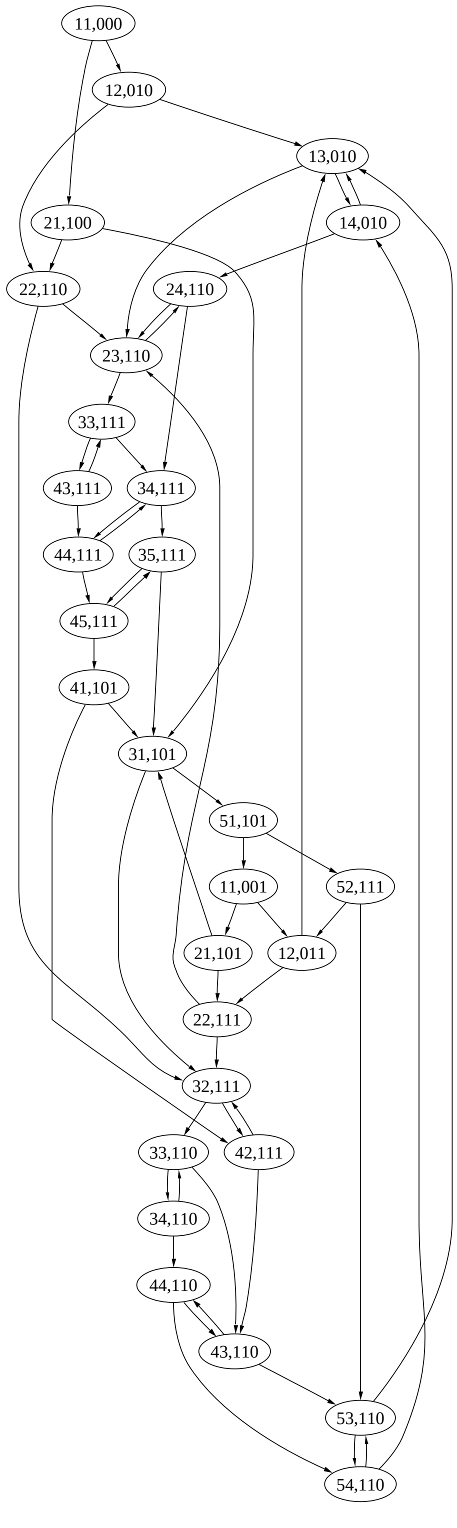
Exercice 60 Trouver un programme auto-reproductible en Java, c'est-à-dire un programme dont le résultat (par impression sur la sortie) est son propre code.

7.9 Applications des automates finis à la concurrence

Parmi les applications des automates finis, un certain nombre d'entre elles consistent à faire des vérifications, principalement de programmes concurrents, par méthode exhaustive (encore dénommé *model checking* en anglais). Si ces méthodes sont parfois peu esthétiques, elles sont souvent beaucoup plus efficaces que d'autres méthodes de correction de programmes, utilisant la logique ou les méthodes formelles. Toutefois, elles se prêtent moins à la paramétrisation (et donc à la modularité) des preuves de correction.

Exemple 7.9 *Vérification de l'algorithme de Peterson.* Souvent la correction de programmes concurrents se ramène à l'étude d'un nombre fini de cas, que l'on peut caractériser par un automate fini. C'est le cas de l'algorithme de Peterson considéré en 6.8 où l'exclusion des sections critique est caractérisée par l'inaccessibilité de certains états, comme expliqué sur la figure 7.12.

Nous considérons à présent deux protocoles d'échanges de messages sur les réseaux. Ces exemple sont un peu plus longs à expliquer. En fait il ne sont pas si éloignés de



Dans l'algorithme de Peterson, une configuration est représentée par le quintuplet $(p, p', \mathit{actif}[0], \mathit{actif}[1], \mathit{tour})$ où p est le compteur ordinal dans le premier programme, p' est celui du deuxième programme, et $\mathit{actif}[0]$, $\mathit{actif}[1]$, tour sont les valeurs des variables de synchronisation. Il y a 5 points possibles dans le code de chacun des processus : pour t_0, alors $p = 1$ correspond au début du code, $p = 2$ est le point du programme avant de mettre $\mathit{actif}[0]$ à vrai (1 ici dans notre codage), $p = 3$ est le point du programme avant le test de $\mathit{actif}[1]$, $p = 4$ est le point avant le test du tour, $p = 5$ est la section critique avant de remettre $\mathit{actif}[0]$ à faux et de retourner en $p = 1$; pour t_1, alors p' prend les valeurs symétriques. La fonction de transition de l'automate est décrite ci-dessous. A chaque état, deux transitions sont possibles selon que le processus t_0 ou t_1 s'exécute. On vérifie sur l'automate de gauche qu'on n'atteint jamais un état de la forme ((5 5 , - - -)).

L'automate complet a $25 \times 8 = 200$ états. Nous n'avons dessiné que les états accessibles depuis ((1 1 , 0 0 1)).

(1 - , - - -)	→	(2 - , 1 - -)
(2 - , - - -)	→	(3 - , - - 1)
(3 - , - 0 -)	→	(5 - , - 0 -)
(3 - , - 1 -)	→	(4 - , - 1 -)
(4 - , - - 0)	→	(5 - , - - 0)
(4 - , - - 1)	→	(3 - , - - 1)
(5 - , - - -)	→	(1 - , 0 - -)
(- 1 , - - -)	→	(- 2 , - 1 -)
(- 2 , - - -)	→	(- 3 , - - 0)
(- 3 , 0 - -)	→	(- 5 , 0 - -)
(- 3 , 1 - -)	→	(- 4 , 1 - -)
(- 4 , - - 1)	→	(- 5 , - - 1)
(- 4 , - - 0)	→	(- 3 , - - 0)
(- 5 , - - -)	→	(- 1 , - - -)

FIG. 7.12 – Vérification par méthode exhaustive de l'algorithme de Peterson.

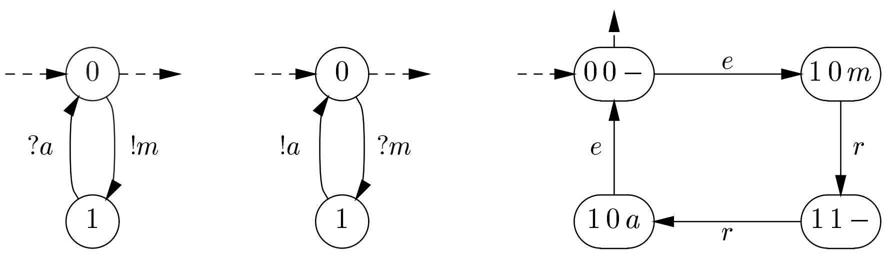


FIG. 7.13 – L'automate d'un simple contrôle de flux sur un réseau. (i) à gauche : l'émetteur ; (ii) au centre : le récepteur ; (iii) à droite : le système en entier

l'exemple précédent. Il s'agit d'échanger de manière sure des messages à travers un réseau. Nous considérons deux protocoles : un simple contrôle de flux et le protocole du bit alterné, dans le cas d'une liaison uni-directionnelle.

Exemple 7.10 *Protocole réseau : contrôle de flux.* Deux processus évoluent en parallèle reliés par un canal de communication. Le premier, l'émetteur, envoie sans cesse des messages m_0, m_1, m_2, ... sur le canal à destination du deuxième processus, le récepteur, qui lit continument ces messages. Les deux processus doivent suivre un protocole de contrôle de flux pour éviter que le récepteur ne soit submergé une multitude de messages. L'émetteur émet un message (action $!m$), puis attend un accusé de réception envoyé par le lecteur (action $?a$). Il peut alors recommencer à émettre un nouveau message à destination du lecteur. Le fonctionnement à deux états de l'émetteur est représenté à gauche sur la figure 7.13. Le processus de lecture lui attend un message émis par l'émetteur. Il le lit (action $?m$), puis émet un accusé de réception vers l'émetteur (action $!a$). Il revient à l'attente de lecture du prochain message. Son fonctionnement est au centre sur la figure 7.13. Le système en entier, constitué de l'émetteur et du récepteur et du contenu du canal de communication, est résumé à droite sur la figure 7.13. Il a 4 états XYZ où X est l'état de l'émetteur, Y celui du récepteur, Z est le contenu du canal de communication, e signifie action de l'émetteur, r signifie action du récepteur.

Le contrôle de flux est proche du contrôle considéré dans les files concurrentes vu en 6.5 ou encore dans le problème du producteur-consommateur de 6.10. Toutefois, il n'y a pas de mémoire commune pour assurer la communication ; il faut envoyer des messages de l'émetteur au récepteur et réciproquement pour assurer la synchronisation.

Exemple 7.11 *Protocle réseau : protocole du bit alterné.* Le protocole précédent suppose qu'aucun message ne puisse se perdre. Or, les pertes de messages sont fréquentes sur un réseau (à un certain niveau de protocole). Il faut donc considérer le cas où le message m_i émis se perd, alors le récepteur ne reçoit rien et ne peut envoyer d'accusé de réception. L'émetteur doit ré-émettre le message m_i au bout d'un certain temps (*time-out*), jusqu'au moment où l'accusé de réception lui parvienne. Mais, il se peut que l'accusé de réception se perde aussi, et alors le récepteur va lire deux fois le même message m_i. Pour éviter la duplication de la lecture d'un même message, le protocle du bit alterné consiste à utiliser un bit pour caractériser le message émis, et mettre le récepteur dans un mode où il attend un certain type de message pour lecture effective. Sur la figure 7.14, on trouve en haut à gauche l'automate décrivant le protocole de

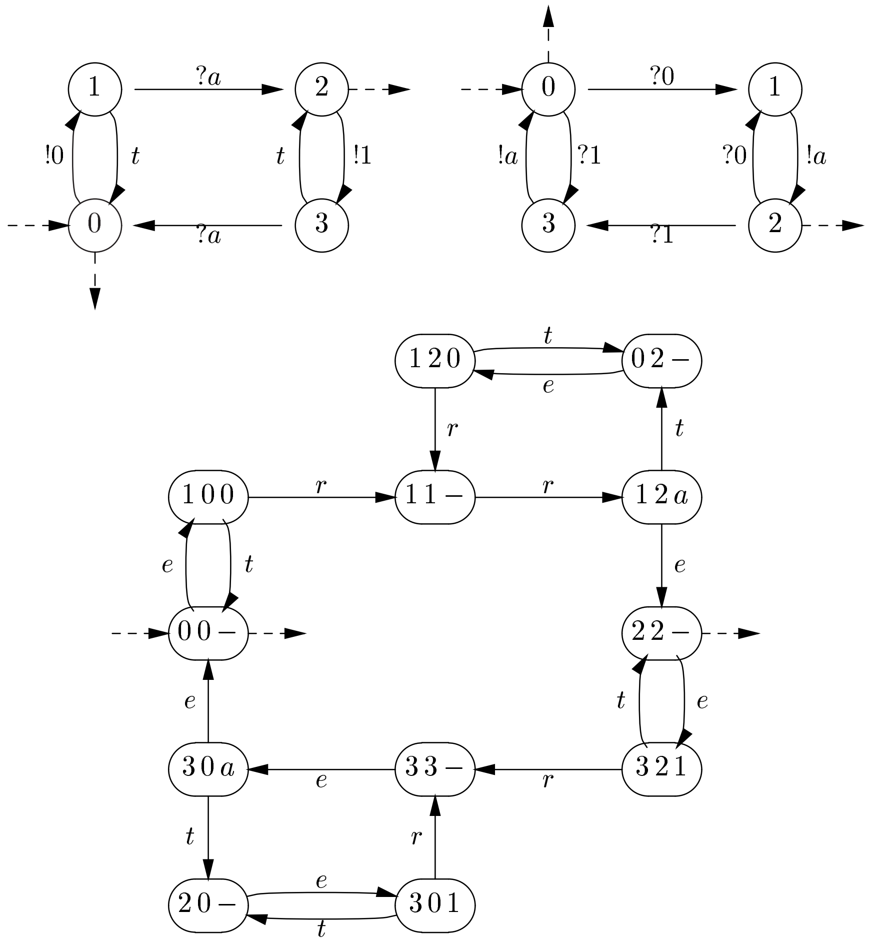


FIG. 7.14 – L'automate du protocole du bit alterné sur une liaison uni-directionnelle. (*i*) en haut à gauche : l'émetteur ; (*ii*) en haut à droite : le récepteur ; (*iii*) en bas : le système en entier.

l'émetteur, en haut à droite celui du récepteur, en bas l'automate de tout le système (émetteur, récepteur, état du canal de communication). Chaque état XYZ de ce dernier automate est donné par l'état X de l'émetteur, l'état Y du récepteur, et le contenu Z du canal ; l'action e veut dire action de l'émetteur, r veut dire action du récepteur, t veut dire dépassement de temps.

Dans l'état 0, l'émetteur envoie un message de type 0 et passe dans l'état 1. Dans l'état 1, l'émetteur attend un accusé de réception. S'il arrive, on le lit et l'émetteur passe à l'état 2. Si un dépassement de temps se produit, l'émetteur revient à l'état 0. Dans l'état 2, l'émetteur envoie un message de type 1 et passe dans l'état 3. Là, il attend un accusé de réception. S'il arrive, il le lit et passe dans l'état 0. Si un dépassement de temps se produit, l'émetteur revient à l'état 2.

Coté récepteur, au début, on est dans l'état 0. Le récepteur attend alors un message de type 0. S'il arrive, on le lit et on va à l'état 1. Si c'est un message de type 1 qui arrive, c'est un message dû à une ré-émission causée par un dépassement de temps, on le saute et on passe à l'état 3. Dans l'état 3, le récepteur émet un accusé de réception et passe à l'état 0. Dans l'état 1, après la lecture d'un message effectif de type 0, on émet un accusé de réception et on passe à l'état 2. Dans l'état 2, on attend un message de type 1 et on va à l'état 3. Si c'est un message de type 0 qui arrive, cela veut dire que c'est un message ré-émis dû à un dépassement de temps, on saute le message, et on revient à l'état 1 pour re-émettre un accusé de réception.

Sur l'automate du système tout entier, on voit comment le système évolue dans le temps en faisant progresser concurremment les deux automates de l'émetteur et du récepteur. Dans l'état XYZ, l'émetteur émet un message de type $\lfloor X/2 \rfloor$, le récepteur est en attente d'un message de type $\lfloor Y/2 \rfloor$. On vérifie donc qu'on ne lit jamais deux fois le même message m_i, puisqu'il est lu la première fois, et sauté s'il est réémis. En fait le lecteur lit m_i sur la transition de 100 à $11-$ et m_{i+1} sur la transition de 321 à $33-$. L'émetteur lit l'accusé de réception sur les transitions de $12a$ à $22-$ et de $30a$ à $00-$. Toutes les autres lectures de messages ou d'accusés de réception sont dues à des réémissions.

Le protocole du bit alterné a en fait une belle structure symétrique qui permet de le généraliser à des canaux de communication bi-directionnels. En outre, au lieu d'assurer le passage d'un seul message, on peut avoir un protocole sur une fenêtre de plusieurs messages, ce qui permet d'accélérer les vitesses de transmissions. Les automates de ces protocoles à fenêtres ont un bien plus grand nombre d'états.

Une toute autre classe d'exemples d'automates finis décrivant des fonctionnements concurrents est celle des automates cellulaires. Les automates cellulaires sont des automates finis dont on suppose les fonctionnements synchronisés par une même horloge. Le prototype des automates cellulaires est celui du jeu de la vie de Conway. Un exemple plus militaire est celui du peloton d'exécution, laissé en exercice (difficile).

Exercice 61 *Le peloton d'exécution.* Il y a une forte brûme et un vent violent ce matin sur le plateau. On ne voit rien, on n'entend rien. Le général convoque $n-1$ soldats, les aligne sur son coté, et veut les faire tirer tous en même temps. Le général et les soldats sont des machines synchrones. Ils réagissent tous en phase en ne tenant compte que des états de leurs deux voisins de gauche ou de droite. Il y a 3 types de machines : le général qui n'a qu'un seul voisin (le premier soldat), les soldats, le soldat en bout de chaîne qui n'a aussi qu'un seul voisin. Montrer qu'il existe une solution, avec un nombre fini d'états, indépendant de n, pour chaque machine. (Indication : supposer $n = 2^k$ et trouver le soldat du milieu)

Chapitre 8
Graphique

Les interfaces graphiques entre les utilisateurs et les ordinateurs sont très importantes. Depuis la première station de travail, l'Alto de Xerox en 1974, ces interfaces n'ont cessé de s'améliorer. En fait, une bonne partie de la programmation consiste à réaliser une interface agréable avec l'utilisateur. Dans ce chapitre, nous allons considérer les fonctions graphiques de base pour réaliser les tracés de vecteurs, de cercles, de coniques, ou de cubiques. Nous regarderons également les interactions possibles que l'utilisateur peut avoir pour entrer des données avec le clavier ou la souris. Depuis l'Alto, l'écran d'un ordinateur est assimilé à un bureau virtuel, sur lequel on peut empiler des feuilles de papier, encore appelées les fenêtres. On écrit en sortie dans une fenêtre ; on envoie des événements en entrée à une fenêtre. L'interaction du système de fenêtre avec les fonctions graphiques est important. Malheureusement le système de fenêtres dépend souvent du langage de programmation utilisé ; peu de standards existent. Ici, nous prendrons l'exemple de la bibliothèque AWT (*Abstract Window Toolkit*) de Sun Microsystems, et en soulignerons son style de programmation par objets. Enfin, nous survolerons quelques problèmes géométriques classiques.

8.1 Graphique « bitmap »

Depuis les années 1970-80, l'écran d'un ordinateur est assimilé à une mémoire (vidéo) directement accessible par les processeurs de calcul ; c'est une matrice de pixels (*picture elements*). Sa dimension vaut typiquement 1152×768. En noir et blanc, chaque pixel est représenté par un bit (0 pour blanc, 1 pour noir, ou le contraire) ; en couleur chaque pixel est représenté par un entier, qui tient par exemple sur 32 bits si on dispose de 2^{32} couleurs. On dit aussi que 32 est la profondeur. Au total, l'écran est représenté par un tableau à trois dimensions, par exemple $1152 \times 768 \times 32$. On dit aussi que la résolution de l'écran est 1152×768.

Chaque pixel peut être assimilé à un disque de diamètre unité, repéré par les coordonnées entières de son centre (x, y). Pour dessiner une courbe sur l'écran, il faut positionner les pixels les plus proches de la courbe souhaitée, et donc discrétiser son tracé. Sur la figure 8.1, on voit les tracés de deux vecteurs de coordonnées $(11, 8)$ et $(11, 4)$. A basse résolution, ils ne ressemblent pas trop à des vecteurs, mais ils semblent parfaits aux résolutions habituelles des écrans.

Supposons les segments de droite donnés par les coordonnées de leurs extrémités. Soit donc un segment de droite P_0P_1 entre les deux points P_0 et P_1 de coordonnées (x_0, y_0) et (x_1, y_1). Posons $dx = x_1 - x_0$ et $dy = y_1 - y_0$. Soit m la pente du segment de droite, $m = dy/dx$. Pour simplifier, nous supposerons que l'on a $0 < m \leq 1$, $dx > 0$ et $dy > 0$. Comme la pente est plus petite que 1, on itère sur la coordonnée x à partir

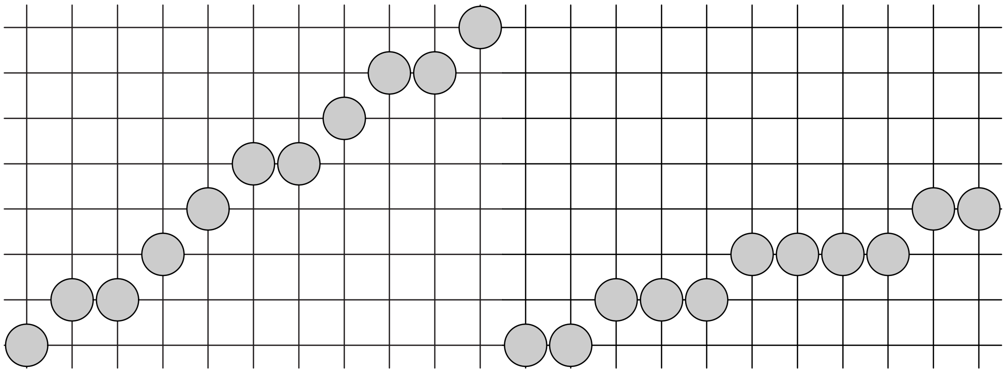


FIG. 8.1 – Tracés de vecteur

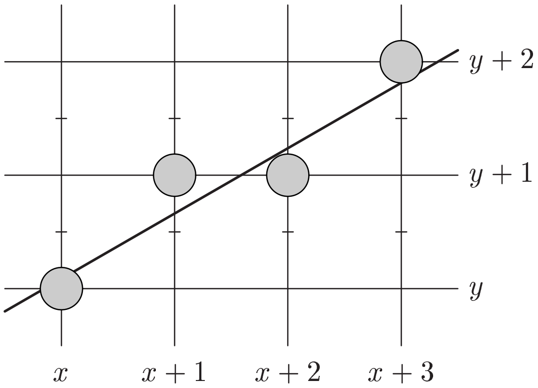


FIG. 8.2 – Tracé de vecteur

de x_0 jusqu'à x_1 en calculant l'ordonnée y donnée par l'équation $y = y_0 + m(x - x_0)$ de la droite support. A chaque itération, on positionne le point $(x, Math.round(y))$ où la fonction *Math.round* calcule l'entier le plus proche de son argument flottant. Ce qui donne la fonction suivante, qui utilise une fonction *tracerPixel* qui est supposée positionner sur l'écran le pixel dont les coordonnées sont passées en arguments.

```
static void tracerVecteur (int x0, int y0, int x1, int y1) {
  int dx = x1 - x0, dy = y1 - y0;
  float m = ((float)dy) / dx;
  for (int x = x0, float y = y0; x <= x1; ++x) {
    tracerPixel(x, Math.round(y));
    y = y + m;
  }
}
```

Cette fonction utilise des opérations flottantes, elle fait un calcul d'arrondi pour chaque pixel, elle est un peu lente. En fait, on peut la transformer en une fonction ne manipulant que des entiers. En effet, l'équation de la droite passant par (x_0, y_0) et (x_1, y_1) est

$$xdy - ydx - x_0dy + y_0dx = 0$$

où $dy = y_1 - y_0$ et $dx = x_1 - x_0$. Si on maintient une erreur e entre le point (x, y) et la droite donnée par la formule suivante

$$e = xdy - ydx - x_0dy + y_0dx$$

(en supposant toujours la pente $0 \leq dy/dx \leq 1$). Pour savoir si le pixel suivant sur la droite $y = x+1$ est à l'est ou au nord-est du point (x, y), on regarde le signe de l'erreur pour le point $(x+1, y+0.5)$

$$e_m = (x+1)dy - (y+0.5)dx - x_0dy + y_0dx$$

Soit

$$e_m = e + dy - dx/2$$

En multipliant par 2, on obtient

$$2e_m = 2e + 2dy - dx$$

Si $2e_m \geq 0$, on positionne le pixel au nord-est et l'erreur devient $2e' = 2e_m - dx$. Sinon on positionne le pixel à l'est et l'erreur devient $2e'' = 2e_m + dx$. On en déduit la fonction suivante pour tracer un vecteur de pente positive inférieure à 1.

```
static void tracerVecteur (int x0, int y0, int x1, int y1) {
  int dx = x1 - x0, dy = y1 - y0;
  int e = 0;
  for (int x = x0, y = y0; x <= x1; ++x) {
    tracerPixel(x,y);
    int em = e + 2*dy - dx;
    if (em >= 0) {
      ++y;
      e = em - dx;
    } else
      e = em + dx;
  }
}
```

Comme x_0, y_0, x_1, y_1 sont entiers, toutes les opérations sont des additions entières de constantes. Cette fonction est souvent attribuée à Bresenham. Ce nom est passé dans le langage courant, et, de manière rapide, on dit que le tracé de vecteur est obtenu par un Bresenham.

Exercice 62 Compléter par symétrie le programme pour qu'il trace un vecteur de pente arbitraire.

Exercice 63 Ecrire une fonction à la Bresenham pour les tracés de cercles et d'ellipses.

Souvent on doit tracer un segment de droite à l'intérieur d'un rectangle donné. On peut remplacer l'appel à *tracerPixel* dans la fonction précédente par un appel à une fonction *tracerPixelDansRectangle* qui prendrait en troisième argument le rectangle en question. Mais une telle solution est lente, et ferait perdre beaucoup de temps si la majorité du segment est à l'extérieur du rectangle. Une meilleure solution consiste donc à calculer l'intersection du segment de droite avec le rectangle, et calculer le point le plus proche en conservant l'erreur introduite par le choix de ce point. Par exemple supposons que le bord gauche est de coordonnée x_{min} d'un rectangle. On calcule

$$y = y_0 + \lfloor (x - x_0)\frac{dy}{dx} + 0.5 \rfloor$$

et on démarre le tracé avec une erreur non nulle

$$2e = 2xdy - 2ydx - 2x_0dy + 2y_0dx$$

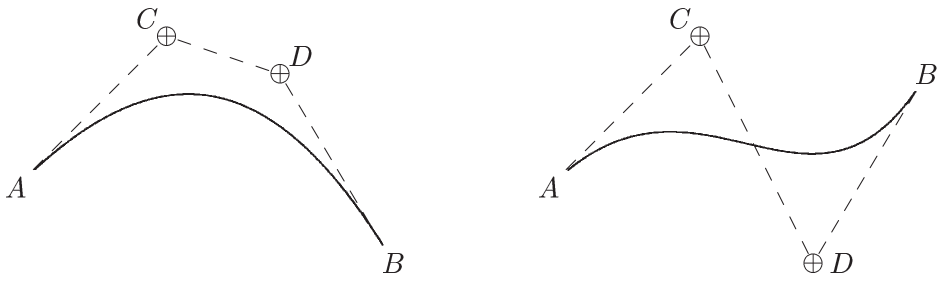


FIG. 8.3 – Cubiques de Bézier

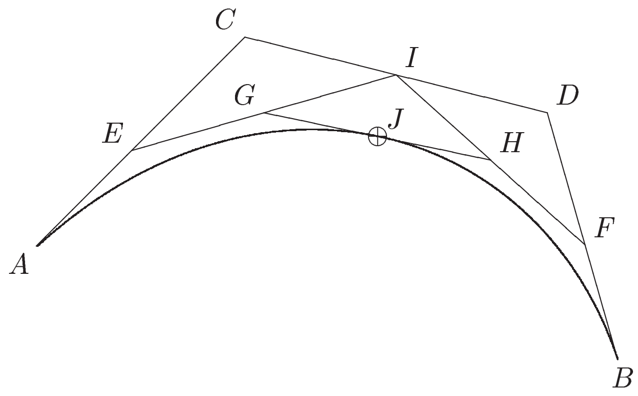


FIG. 8.4 – Décomposition d'une cubique de Bézier

Ainsi si on trace un même segment à l'intérieur de deux rectangles contigus, le segment aura toujours l'air d'un segment de droite, plutôt qu'une ligne brisée si on avait oublié l'erreur pour lancer la fonction de tracé à la Bresenham. Ce point importe quand les rectangles correspondent à des parties de fenêtres cachées.

Un autre exemple de courbe classique est celui des cubiques de Bézier (*Bézier spline*). Ces courbes interviennent dans les dessins de carrosseries (Pierre Bézier était ingénieur chez Renault) ou dans les tracés de lettres. Une cubique de Bézier est donné par quatre points A, B, C, D, elle passe par les deux points A et B, et la valeur de ses tangentes en A sont les vecteurs $\overrightarrow{AC}$ en A et $\overrightarrow{DB}$ en B comme indiqué sur la figure 8.3. On peut montrer que la le segment de cubique entre A et B est toujours inscrit dans le quadrilatère $ACDB$. Ces cubiques ont la belle loi de décomposition suivante. Soient E, F, I, G, H, J, les milieux respectifs des segments AC, BD, CD, EI, FI, GH. Alors, la cubique de Bézier pour les points A, B, C, D est la courbe obtenue pour les points A, J, E, G suivie de celle pour les points J, B, H, F comme indiqué sur la figure 8.4. On peut donc tracer cette courbe récursivement en recalculant les points de contrôle de la cubique à chaque appel récursif. Quand le quadrilatère est très plat, on peut approximer le tracé par le segment de droite AB. D'où la fonction suivante :

```
static Point milieu (Point a, Point b) {
  return new Point ((a.x + b.x)/2, (a.y + b.y)/2);
}

static void tracerBezier (Point a, Point b, Point c, Point d) {
  if (estPlat (a, c, d, b) )
```

```
    tracerVecteur (a.x, a.y, b.x, b.y);
  else {
    Point e = milieu (a, c);
    Point f = milieu (b, d);
    Point i = milieu (c, d);
    Point g = milieu (e, i);
    Point h = milieu (f, i);
    Point j = milieu (g, h);
    tracerBezier (a, j, e, g);
    tracerBezier (j, b, h, f);
  }
}
```

où *estPlat* est une fonction testant si les hauteurs CH et DH' du quadrilatère $ACDB$ sont petites. En utilisant le produit vectoriel de $\overrightarrow{AC}$ et de $\overrightarrow{CD}$, on a $CH < \epsilon$ si et seulement si

$$|\overrightarrow{AC} \wedge \overrightarrow{AB}|^2 < 4\epsilon^2 \times AB^2$$

En prenant $\epsilon = 1$, puisqu'il est inutile de descendre en dessous du pixel comme précision, on écrit la fonction *estPlat*.

```
static int produitVect (Point a, Point b, Point c, Point d) {
  return (b.x - a.x) * (d.y - c.y) - (b.y - a.y) * (d.x - c.x);
}

static int norme2 (Point a, Point b) {
  return (b.x - a.x) * (b.x - a.x) + (b.y - a.y) * (b.y - a.y);
}

static boolean estPlat (Point a, Point c, Point d, Point b) {
  int v = produitVect(a, c, a, b);
  int w = produitVect(b, d, b, a);
  int l2 = 4 * norme2(a, b);
  return v*v < l2 && w*w < l2;
}
```

Remarquons qu'on peut éviter d'allouer de la mémoire pour les nouveaux points dans la fonction *tracerBezier* en calculant directement les coordonnées des points J, E, F, G et H.

Les tracés de cubiques interviennent dans les dessins des lettres en imprimerie. Par exemple, le tracé du caractère (a) en police de caractères romane est grossi sur la figure 8.5. Les contours arrondis de la lettre sont des cubiques. Les coordonnées de leurs points de contrôle sont données sur la partie droite de la figure. Le programme *Metafont* qui permet de définir les polices de caractères minimisee le nombre de points contrôle à donner. Ce programme, dû à Knuth, est très astucieux. Il génère des caractères compatibles avec TeX et Latex. Les cubiques sont aussi à la base du fonctionnement de PostScript, le langage de programmation de la majorité des imprimantes, et de sa version simplifiée PDF.

Sur une imprimante, il y a typiquement 600 à 1200 points par pouce. Sur un écran d'ordinateur, la résolution est plus faible puisqu'il y a moins de 100 points par pouce. Du coup, les cubiques sont moins utilisées dans les tracés sur un écran. Elles apparaissent dans des systèmes comme MacOS X (avec *Display PDF*), ou autrefois NeXT-Step et NeWS (avec *Display PostScript*). Souvent, on se contente de tracés de vecteurs et de rectangles. En effet, les lettres digitalisées sont stockées comme des tableaux rectangulaires de pixels monocolores. Par exemple, la lettre (a) devient le rectangle de dimension 6×6 pixels représenté sur la figure 8.6, ou encore le tableau de 6 entiers

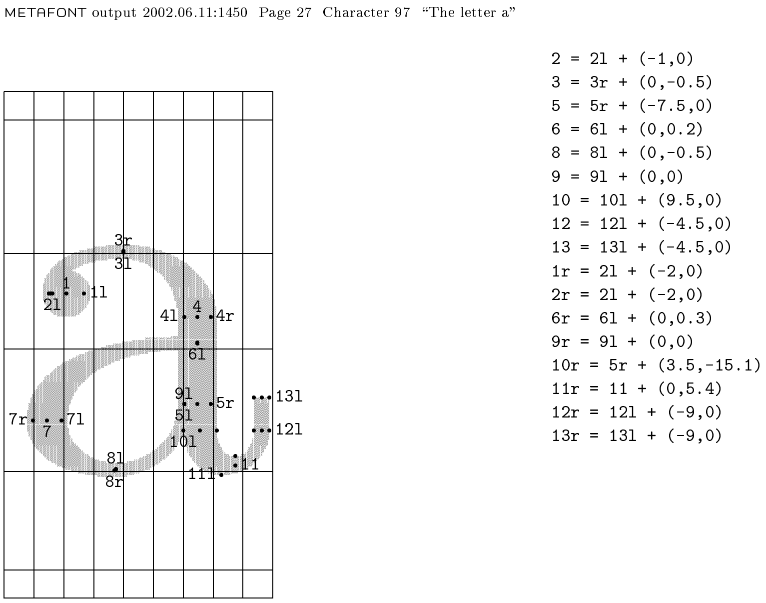


FIG. 8.5 – La définition de la lettre (a) en Metafont dans la police *cmr8*

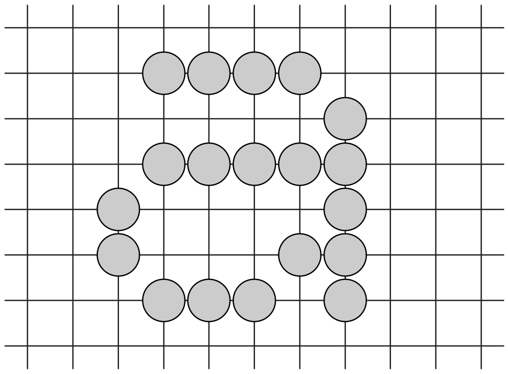


FIG. 8.6 – Le « bitmap » de la lettre (a) dans une police fixe 8×13

$\{30, 1, 31, 33, 35, 29\}$ en représentation binaire. Pour afficher la lettre (a) sur l'écran, il faut copier rapidement ce tableau rectangulaire de bits dans la mémoire vidéo de l'écran à l'endroit désiré pour l'afficher.

Une autre opération fréquente sur les écrans est le défilement de texte dans une fenêtre. Il s'agit encore de copier la partie rectangulaire R de la fenêtre (la fenêtre sans la première ligne de texte) vers une autre partie rectangulaire R' (le haut de la fenêtre) pour libérer l'espace rectangulaire S (la dernière ligne de texte) qu'il faut mettre à blanc avant d'y afficher des lettres, comme indiqué sur la figure 8.7. Pour faire défiler le texte dans la fenêtre, on doit donc faire $R' \leftarrow R$ et $S \leftarrow 0$. La fenêtre peut avoir une dimension proche de l'écran. Alors les rectangles R et R' peuvent contenir 1000×800 pixels, soit 800000 pixels à recopier rapidement. Remarquons aussi que R et R' se recouvrent et que la recopie doit se faire de haut vers le bas pour ne pas écraser les pixels à recopier. Le rectangle S est en général plus petit, il contient de l'ordre de 20000 pixels. Une

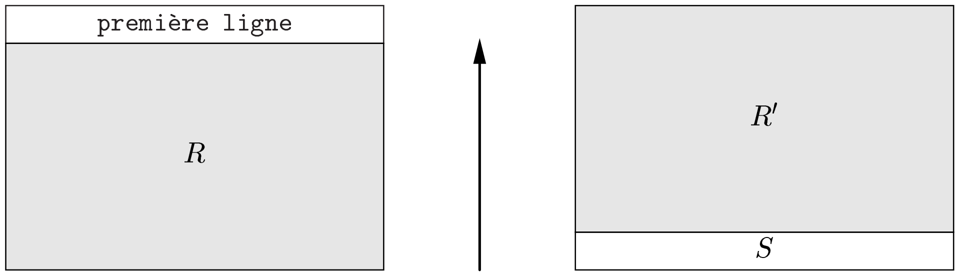


FIG. 8.7 – Défilement de texte dans une fenêtre

opération de base du graphique « bitmap »est donc la copie d'une zone rectangulaire dans une autre (*bit block transfert* ou encore *bitblt*). La forme générale de cette opération est $R \leftarrow R \otimes S$, où R, S sont deux rectangles de même taille et $\otimes$ est une opération booléenne sur les pixels. Pendant longtemps, l'optimisation de ces instructions a été un signe de qualité pour les stations de travail, avec des techniques sophistiquées comme le dépliage des boucles, ou la compilation au vol de code. Actuellement, ces opérations sont faites par le matériel, puisque les circuits graphiques ne sont plus compliqués à concevoir avec le développement foudroyant du matériel.

Considérons le dépliage de boucle, comme exemple d'optimisation. Supposons que x_1, y_1, x_2, y_2 sont les champs désignant les abcisses et ordonnées des extrémités sud-ouest et nord-est d'un rectangle. Alors la fonction suivante recopie les pixels d'un rectangle source s dans un rectangle destination d.

```
static void bitblt (Rect d, Rect s) {
  for (int j = 0; j < d.y2 - d.y1; ++j)
    for (int i = 0; i < d.x2 - d.x1; ++i)
      copierPixel(d.x1+i, d.y1+j, s.x1+i, s.x2+j);
}
```

où *copierPixel(dx,dy,sx,sy)* copie le pixel de coordonnées (sx, sy) dans le pixel (dx, dy). Souvent cette fonction peut être réalisée par une seule instruction machine. Comme on l'itère plusieurs centaines de fois, on exécute autant de fois le test de fin de boucle sur i. On obtient un facteur 2 en vitesse si on déplie la boucle un certain nombre de fois. Ici pour simplifier, nous ne la déplions que 4 fois ; ce qui donne la fonction suivante :

```
static void bitblt (Rect d, Rect s) {
  for (int j = 0; j < d.y2 - d.y1; ++i) {
    int i = 0;
    switch ((d.y2-d.y1) % 4) {
      case 3: copierPixel(d.x1+i, d.y1+j, s.x1+i, s.x2+j); ++i;
      case 2: copierPixel(d.x1+i, d.y1+j, s.x1+i, s.x2+j); ++i;
      case 1: copierPixel(d.x1+i, d.y1+j, s.x1+i, s.x2+j); ++i;
      case 0:
    }
    while (i < d.x2 - d.x1) {
      copierPixel(d.x1+i, d.y1+j, s.x1+i, s.x2+j); ++i;
      copierPixel(d.x1+i, d.y1+j, s.x1+i, s.x2+j); ++i;
      copierPixel(d.x1+i, d.y1+j, s.x1+i, s.x2+j); ++i;
      copierPixel(d.x1+i, d.y1+j, s.x1+i, s.x2+j); ++i;
    }
  }
}
```

8.2 Entrées graphiques

Les dispositifs de pointage (souris, trackpad, tablettes) permettent de désigner des parties de l'écran. Les entrées graphiques se font en combinant la position du curseur et l'utilisation des boutons de la souris ou des touches du clavier. Supposons qu'il existe deux fonctions : *button* qui retourne un booléen qui ne vaut vrai que si un des boutons de la souris est enfoncé, et *getMouse* qui retourne un point donnant les valeurs des coordonnnées courantes du curseur.

Une entrée graphique peut se faire sur un front descendant d'un bouton de la souris. Cela correspond à la fonction suivante :

```
static Point lirePoint () {
  while (!button())
    ;
  Point p = getMouse ();
  while (button())
    ;
  return p;
}
```

On attend que le bouton soit enfoncé, on note alors les coordonnées du curseur, et on attend que le bouton soit relâché. Si on ne fait pas cette dernière attente, on peut relire deux fois de suite le même point si on itère sur la fonction *lirePoint*. Une technique plus sûre consiste à lire les coordonnées du curseur sur un front montant. On attend qu'un bouton de la souris soit relâché avant de faire la lecture des coordonnées du curseur.

```
static Point lirePoint () {
  while (!button())
    ;
  while (button())
    ;
  return getMouse ();
}
```

Ces deux dernières fonctions font des attentes actives. Par ailleurs, elles posent un problème quand on veut à la fois faire des entrées à la souris et au clavier. Il faut alors attendre un événement sur la souris ou un événement sur le clavier. La fonction *lire-Point* est blocante ; sur le clavier, la lecture de caractères se fait aussi par une fonction blocante *read()*. On doit donc soit créer deux processus pour faire concurremment la lecture à la souris et au clavier, soit multiplexer les deux événements possibles à la souris et au clavier. La première solution est lourde. En général, on adopte la deuxième solution. Les bibliothèques de fonctions graphiques fournissent une structure d'événements multiplexés entre le clavier et la souris, générés dans une file d'attente. On peut lire ces événements dans cette file d'attente comme on lit les caractères sur un terminal. Un événement est émis quand on appuie sur un bouton de la souris, quand on le relâche, quand la souris se déplace, quand une touche du clavier est enfoncé, ou relâché. Pour chacun d'entre eux, la position courante du curseur est fournie, ainsi que le temps. Alors, la programmation avec les événements suit le schéma suivant :

```
for (;;) {
  Event e = nextEvent();
  switch (e.type) {
  case keyPressed: ... break;
  case keyReleased: ... break;
  case mousePressed: ... break;
  case mouseReleased: ... break;
```

```
      ...
    }
  }
```

On peut remarquer qu'on ne fait plus d'attente active, car tout le problème est reporté sur la fonction de bibliothèque *nextEvent*.

La programmation avec les événements n'est pas très modulaire, puisqu'elle consiste à faire une grande boucle réagissant à tous les événements possibles. Si on veut détecter un double-clic à la souris, par rapport à la suite de deux événements souris, puis clavier, ou simplement deux clics simples consécutifs, il est plus facile de représenter la gestion des événements clavier-souris par un petit automate fini déterministe. En fait, la logique de ces interactions peuvent être incroyablement compliquée, et plusieurs systèmes ont été proposés pour en simplifier la programmation (Squeak, Esterel, Statecharts). En fait, ces systèmes permettent de traiter plus généralement de la *programmation réactive* dont la gestion des événements clavier-souris est un cas particulier. Souvent, cela revient à traiter les entrées graphiques par des automates indépendants, et la déterminisation de l'automate non-déterministe correspondant à leur réunion est assuré par le système proposé.

Une autre manière de gérer l'interaction avec le clavier et la souris se fait souvent par des fonctions de rappel (*callbacks*) dans les systèmes orientés vers la programmation par objets. Dans chaque classe graphique, on définit la fonction de rappel, c'est-à-dire la fonction à appeler lorsqu'un événement dû à une entrée graphique se produit. C'est la hiérarchie des classes qui permet d'appeler la bonne fonction. Nous allons voir cela dans le cas particulier de la bibliothèque *AWT*.

8.3 La bibliothèque AWT

Un système de fenêtres sur l'écran d'une station de travail est censé représenter les différentes feuilles de papier que l'on peut avoir sur un bureau. Certaines sont cachées par d'autres ; certaines sont côte à côte ; on écrit ou on dessine dans la majorité des fenêtres. Cette analogie a résisté à trente années d'évolution de l'informatique. Le modèle utilisé dans la bibliothèque AWT de Java provient de Smalltalk, un des premiers langages de programmation par objets inventé à Xerox. Il tend à séparer les parties modèle, présentation, et contrôle de chaque élément graphique. En effet, une interface graphique permet de représenter graphiquement un certains nombre de données du modèle sous-jacent. Il y a souvent plusieurs présentations possibles pour les mêmes données : par exemple des pourcentages peuvent être représentés par des barres colorées, ou par diagrammes en forme de camembert ; un booléen peut être représenté par l'affichage textuelle de sa valeur, ou par l'état d'un bouton (enfoncé ou pas), ou par la couleur d'une fenêtre ; etc. Pour le corps du programme, l'important est la valeur des données du modèle, et non leur représentation graphique. Pour l'interface graphique, au contraire, la présentation constituera l'essentiel de la programmation. La partie contrôle d'une interface consiste à définir l'interaction avec l'utilisateur, et la répercussion sur le modèle. Il repose aussi fortement sur la représentation hiérarchique des classes. Il est impossible d'exposer ici de manière didactique toute la bibliothèque AWT. Nous ne ferons ici qu'en présenter quelques exemples idiomatiques.

La classe *Component* est la classe abstraite de base du paquetage AWT. Un objet de cette classe, un composant, peut être spécialisé en un bouton, une étiquette, un canevas,

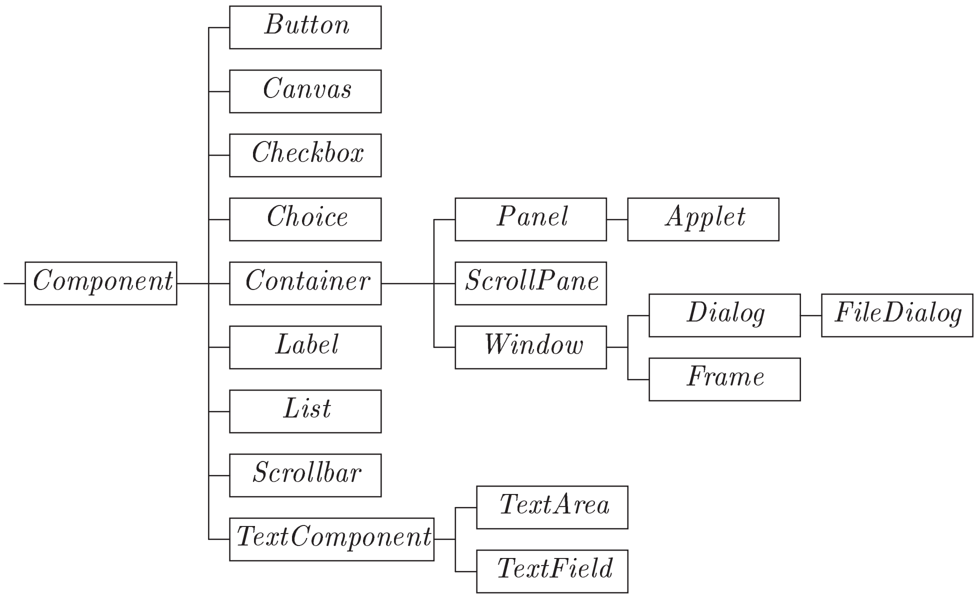


Fig. 8.8 – La hiérarchie des composants graphiques en AWT

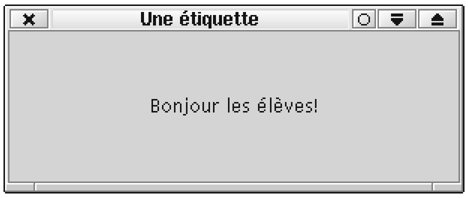


Fig. 8.9 – Un cadre en AWT

un *checkbox*, un choix, un conteneur, une liste de textes, une barre de défilement, un texte, comme indiqué sur la figure 8.8. Le composant le plus simple est une étiquette (classe *Label*) qui permet de placer un texte dans un conteneur. Un conteneur (classe *Container*) est une liste de composants, affichés de devant vers l'arrière. Une fenêtre (classe *Window*) est un conteneur correspondant à une fenêtre dans le système de fenêtres du système sous-jacent. Un cadre (classe *Frame*) est une fenêtre avec un titre et un bord. Un exemple simple d'interface graphique est le suivant :

```
import java.awt.*;
class Frame1 {
  public static void main (String[ ] args) {
    Frame f = new Frame ("Une étiquette");
    f.add ("Center", new Label ("Bonjour les élèves!", Label.CENTER));
    f.setSize (300, 100);
    f.show();
} }
```

Une cadre de taille 300×100 pixels est allouée avec le titre « Une étiquette ». Ce cadre peut contenir plusieurs composants comme tout conteneur ; ici il ne contient qu'une étiquette avec le texte "Bonjour les élèves !" centré. La méthode *add* permet de rajouter des composants dans le cadre, initialement vide. Son premier argument est un attribut de placement de son second argument dans le cadre. Ici, l'étiquette est centrée. Le deuxième argument du constructeur *Label* indique la justification du texte dans l'étiquette. Enfin, la méthode *show* affiche ce cadre en le plaçant devant les autres fenêtres.

Le programme précédent peut aussi s'écrire dans un style plus orienté objet :

```
class Frame2 extends Frame {
  Frame2 (String s) {
    super(s);
    add ("Center", new Label ("Bonjour les élèves!", Label.CENTER));
    setSize (300, 400);
    show();
  }

  public static void main (String[ ] args) {
    Frame2 f = new Frame2 ("Une étiquette");
} }
```

La méthode *paint* est la méthode appelée dès que le système veut afficher le cadre. Par défaut, elle a une action standard dans tous les composants. On peut toutefois la redéfinir. Ici au bas du cadre, on dessine deux rectangles adjacents. L'argument de *paint* est un contexte graphique, qui donne l'état courant de dessin pour l'objet à dessiner. C'est le système qui appelle *paint* avec le bon état. Celui-ci contient des informations sur l'origine du rectangle contenant le cadre, sur sa taille, sur la couleur courante de dessin, sur la police de caractères courante, etc. La fonction *getSize* donne la taille du cadre. L'appel à *super.paint* passe l'ordre d'affichage à la méthode *paint* du conteneur qui la redispatche à tous les éléments du cadre.

```
class Frame3 extends Frame {
  Frame3 (String s) {
    super(s);
    add ("North", new Label ("Deux couleurs", Label.CENTER));
    setSize (300, 400);
    show();
  }

  public void paint (Graphics g) {
    super.paint(g);
    int w = getSize().width;
    int h = getSize().height;
    int x = w/2; int y = h/2;
    g.setColor(Color.red);
    g.fillRect(x-w/4, y-w/4, w/4, w/2);
    g.setColor(Color.yellow);
    g.fillRect(x, y-w/4, w/4, w/2);
  }

  public static void main (String[ ] args) {
    Frame3 f = new Frame3 ("Un contexte graphique");
  }
}
```

Des détecteurs d'événements peuvent être attachés à certains composants par la méthode *addActionListener*. Le paramètre est un objet implémentant une *ActionListener*, c'est-à-dire ayant un champ *actionPerformed*. Ce sera la fonction déclenchée quand l'action se produira. Dans l'exemple de base suivant, un bouton est créé dans un cadre, avec la fonction *System.exit* attachée. Le code de sortie est 0, indiquant une sortie sans erreur. Le bouton porte l'inscription « Quitter » :

Fig. 8.10 – Un bouton en AWT

```
import java.awt.*;
import java.awt.event.*;

public class Frame4 extends Frame {

  public Frame4 (String title) {
    super (title);
    setLayout (new FlowLayout());
    Button q = new Button ("Quitter");
    add(q);
    setSize (300, 100);
    q.addActionListener (new Quit());
    show();
  }

  public static void main (String[ ] args) {
    Frame4 f = new Frame4 ("Un bouton");
} }

class Quit implements ActionListener {
  public void actionPerformed (ActionEvent e) {
    System.exit(0);
} }
```

On peut compliquer l'exemple en mettant cote à cote deux boutons « Quitter » et « A » et une zone de texte modifiable. Au premier bouton, on attache le même détecteur qu'auparavant. Le deuxième est associé à une fonction de la classe *ActionA* qui remplit la zone de texte avec la chaîne « Bouton a ! ». Remarquons que la politique de disposition des composants dans le cadre est le mode *FlowLayout*, indiquant que les boutons doivent être mis cote à cote.

```
public class Frame5 extends Frame {

  public Frame5 (String title) {
    super (title);
    setSize (300, 100);
    setLayout (new FlowLayout());
    Button q = new Button ("Quitter");
    Button a = new Button ("A");
    TextField t = new TextField (20);
    add(q); add(a); add(t);
    q.addActionListener (new Quit());
    a.addActionListener (new ActionA(t, "Bouton a!"));
    show();
  }

  public static void main (String[ ] args) {
    Frame5 f = new Frame5 ("Deux boutons et un texte");
} }
```

FIG. 8.11 – Deux boutons et un texte en AWT

```
class ActionA implements ActionListener {
  TextField t; String s;
  ActionA (TextField t0, String s0) { t = t0; s = s0; }

  public void actionPerformed (ActionEvent e) {
    t.setText(s);
} }
```

Voici un autre exemple d'interface où on a deux détecteurs d'actions déclenchées par des boutons, et une autre plus basique déclenchée par la souris. Cette dernière est mise en place par la fonction *addMouseListener*. L'action correspondante écrit sur le terminal les coordonnées courantes du curseur au moment où l'événement e de la souris s'est produit. Ces coordonnées sont accessibles par les méthodes *getX* et *getY*. Dans cet exemple, on utilise les fonctions *validate* et *pack* pour placer les boutons et la zone de texte.

```
public class Frame6 extends Frame {

  public Frame6 (String title) {
    super (title);
    setLayout (new FlowLayout());
    Button q = new Button ("Quit"); add(q);
    Button a = new Button ("A"); add(a);
    TextField t = new TextField (20); add(t);
    validate(); pack();
    q.addActionListener (new Quit());
    a.addActionListener (new ActionA(t, "Bouton a!"));
    addMouseListener (new ActionB());
    show();
  }

  public static void main (String[ ] args) {
    Frame6 f = new Frame6 ("Une interaction souris");
  }
}

class ActionA implements ActionListener {
  TextField t; String s;
  ActionA (TextField t0, String s0) { t = t0; s = s0; }

  public void actionPerformed (ActionEvent e) {
    t.setText(s);
  }
}

class ActionB extends MouseAdapter {
  public void mouseReleased (MouseEvent e) {
    System.out.println ("Bouton B: " + e.getX() + "," + e.getY());
```

```
      System.exit(1);
    }
  }
```

Les événements souris sont : bouton enfoncé, bouton relâché, souris déplacée, souris déplacée bouton enfoncé (*drag*). Un détecteur d'événements-souris *MouseListener* est une interface contenant les quatre méthodes *mouseClicked*, *mouseEntered*, *mouseExited*, *mouseReleased*. Pour simplifier la programmation, on peut utiliser un *MouseAdapter* qui est une implémentation de l'interface précédente avec les quatre méthodes prédéfinies comme étant vides. Ainsi on peut considérer des sous-classes de *MouseAdapter* en ne redéfinissant que les méthodes à spécifier.

Exercice 64 Programmer une interface utilisateur où on imprime les coordonnées du point courant sur un clic souris.

Exercice 65 Programmer une interface utilisateur qui dessine un vecteur entre deux points rentrés à la souris.

8.4 Un exemple d'appliquette

Une appliquette (*applet*) est une sous-classe des paneaux, et donc aussi une sous-classe des conteneurs de AWT, comme indiqué par la hiérarchie des classes de la figure 8.8. Une appliquette permet à un programme Java de s'exécuter dans un paneau à l'intérieur d'une page en format Html affichée par un navigateur (Tous les navigateurs modernes contiennent un interpréteur Java). Voici un exemple de telle page avec un paneau calculé par une appliquette dont le code est le code compilé d'une classe *Mondrian* et dont la taille est un rectangle 48×32 sur l'écran.

```
<html>
<head>
<title>Mondrian</title>
</head>
<body text="#00004c" bgcolor="#ffffff" link="#00004c">
<br><br><br>
<applet
    code=Mondrian.class
    width=48
    height=32> Mondrian
</applet> Une belle appliquette!
</body>
</html>
```

Le code (trop long) de l'appliquette n'a pas un grand intérêt, mais est assez représentatif. Il se trouve à de multiples endroits sur le Web : il s'agit de faire des tableaux imaginaires de Piet Mondrian, artiste abstrait du début du 20ème siècle. Quand l'appliquette est chargée, le contrôle est donné à sa méthode *init* ; c'est là qu'on crée le processus exécutant l'appliquette et que l'on note les tailles du rectangle englobant passées en paramètres dans la page Web. L'appliquette contient aussi deux méthodes : *start* pour redémarrer l'appliquette chaque fois qu'elle a été stoppée, *stop* pour arréter l'exécution de l'appliquette. Typiquement le navigateur relance l'appliquette chaque fois que la page Web est affichée et l'arrête lorsque la page Web disparait. La méthode *destroy* sert à récupérer les ressources allouées à l'appliquette, quand on est sûr qu'elle ne réapparaitra pas. Enfin, la méthode *run* comme dans tout processus d'une classe *Runnable* est le point d'entrée du code de ce processus, appelé par *t.start()*. Cette fonction repeint le paneau de l'appliquette toutes les 4s ou quand le bouton de la souris est pressé.

Le code suivant tire au sort un certain nombre de lignes horizontales et verticales. Puis il tire un certain nombre de points en traçant les lignes horizontales ou verticales les plus longues ne coupant pas les lignes déjà tracées. Enfin, il tire au sort un autre ensemble de points pour colorier dans une couleur aléatoire les rectangles les contenant.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Mondrian extends Applet implements Runnable {
  Thread t;
  final static int nMaxL = 4, nMaxR = 7, epaisseur = 1, pas = 2;
  static int nLV, nLH, nRects;
  static int xmax, ymax;
  Ligne[ ] lV = new Ligne[nMaxL*2], lH = new Ligne[nMaxL*2];
  Rect[ ] rect = new Rect[nMaxR];
  Random rand = new Random();

  public Mondrian() { super(); }

  public void init()
  {
    xmax = getSize().width; ymax = getSize().height;
    repaint();
    addMouseListener (new Action(this));
    t = new Thread (this);
    t.start();
  }

  public void start() {
    if (t.isAlive()) t.resume();
    else t.start();
    repaint();
  }

  public void stop() { t.suspend(); }
  public void destroy() { t.stop(); }
  public void run() {
    while (true) {
      repaint();
      try { Thread.sleep(4000); }
      catch (InterruptedException e) { stop(); }
    }
  }

  public void paint (Graphics g) {
   nLV = 0; nLH = 0; nRects = 0;
   g.setColor (Color.white);
   g.fillRect (0, 0, xmax, ymax);
   g.setColor (Color.black);
   choisirLV (1+randInt(nMaxL-1));
   choisirLH (1+randInt(nMaxL-1));
   choisirLV2 (1+randInt(nMaxL-1));
   choisirLH2 (1+randInt(nMaxL-1));
   choisirRects(3+randInt(nMaxR-3));
   dessinerRects (g);
   dessinerLV (g);
```

```
 dessinerLH (g);
}

int randInt (int m) {
  return Math.round(rand.nextFloat() * m) ;
}

void choisirLV (int n) {
  for (int i = 0; i < n; ++i) {
    int x = (randInt(xmax) / pas) * pas;
    lV[nLV++] = new Ligne(x, 0, x, ymax);
  }
}

void choisirLV2 (int n) {
  for (int i = 0; i < n; ++i) {
    int x = (randInt(xmax) / pas) * pas;
    int y = (randInt(ymax) / pas) * pas;
    lV[nLV++] = new Ligne(x, enDessousDe(x,y), x, auDessusDe(x,y));
   }
}

void choisirLH (int n) {
  for (int i = 0; i < n; ++i) {
    int y = (randInt(ymax) / pas) * pas;
    lH[nLH++] = new Ligne(0, y, xmax, y);
  }
}

void choisirLH2 (int n) {
  for (int i = 0; i < n; ++i) {
    int x = (randInt(xmax) / pas) * pas;
    int y = (randInt(ymax) / pas) * pas;
    lH[nLH++] = new Ligne(aGaucheDe(x,y), y, aDroiteDe(x,y), y);
  }
}

void dessinerLV (Graphics g) {
  for (int i = 0; i < nLV; ++i) {
    int dy = lV[i].y2 - lV[i].y1;
    g.fillRect(lV[i].x1, lV[i].y1, epaisseur, dy);
  }
}

void dessinerLH (Graphics g) {
  for (int i = 0; i < nLH; ++i) {
    int dx = lH[i].x2 - lH[i].x1;
    g.fillRect(lH[i].x1, lH[i].y1, dx, epaisseur);
  }
}

void choisirRects (int n) {
  for (int i = 0; i < n; ++i) {
    int x = randInt (xmax);
    int y = randInt (ymax);
    int x0 = aGaucheDe (x,y);
    int y0 = enDessousDe (x,y);
    int w = aDroiteDe (x,y) - x0;
    int h = auDessusDe (x,y) - y0;
    rect[nRects++] = new Rect (x0, y0, w, h);
  }
```

```
  }

  int aGaucheDe (int x, int y) {
    int res = 0;
    for (int i = 0; i < nLV; ++i) {
      int x2 = lV[i].x1;
      if (lV[i].y1 <= y && y <= lV[i].y2 && res < x2 && x2 <= x)
        res = x2;
    }
    return res;
  }

  int aDroiteDe (int x, int y) {
    int res = xmax;
    for (int i = 0; i < nLV; ++i) {
      int x2 = lV[i].x1;
      if (lV[i].y1 <= y && y <= lV[i].y2 && x <= x2 && x2 < res)
        res = x2;
     }
     return res;
  }

  int enDessousDe(int x, int y) {
    int res = 0;
    for (int i = 0; i < nLH; ++i) {
      int y2 = lH[i].y1;
      if (lH[i].x1 <= x && x <= lH[i].x2 && res < y2 && y2 <= y)
        res = y2;
    }
    return res;
  }

  int auDessusDe(int x, int y) {
    int res = ymax;
    for (int i = 0; i < nLH; ++i) {
      int y2 = lH[i].y1;
      if (lH[i].x1 <= x && x <= lH[i].x2 && y <= y2 && y2 < res)
        res = y2;
    }
    return res;
  }

  void dessinerRects (Graphics g) {
    for (int i = 0; i < nRects; ++i) {
      g.setColor(choisirCouleur());
      g.fillRect(rect[i].x, rect[i].y, rect[i].w, rect[i].h);
    }
    g.setColor(Color.black);
  }

  Color choisirCouleur() {
    int couleur = randInt(4);
    if (couleur == 0) return Color.yellow;
    if (couleur == 1) return Color.yellow;
    if (couleur == 2) return Color.blue;
    if (couleur == 3) return Color.red;
    return Color.red;
  }
}

class Ligne {
```

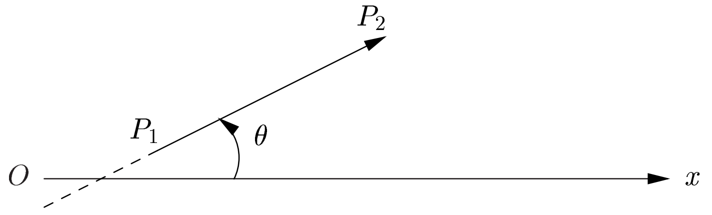


FIG. 8.12 – Angle formé par un vecteur et l'axe Ox

```
  int  x1,y1, x2,y2;
  Ligne(int a1, int a2, int a3, int a4) { x1 = a1; y1 = a2; x2 = a3; y2 = a4; }
}

class Rect {
  int  x,y,w,h;
  Rect(int a1, int a2, int a3, int a4) { x = a1; y = a2; w = a3; h = a4; }
}

class Action extends MouseAdapter {
  Mondrian t;
  Action (Mondrian t0) {t = t0; }
  public void mousePressed (MouseEvent e) {
    t.repaint();
  }
}
```

8.5 Enveloppe convexe

Le calcul de l'enveloppe convexe de n points est un des tous premiers algorithmes de l'algorithmique géométrique. Les deux algorithmes que nous considérerons, les marches de Jarvis et de Graham, s'appuient sur un calcul d'angles formés par les vecteurs avec l'axe horizontal. L'angle θ formé par le vecteur $\overrightarrow{P_1P_2}$ de coordonnées (dx, dy) avec l'axe Ox vérifie $\theta = atan(dy/dx)$, comme indiqué sur la figure 8.12. Mais le calcul de la fonction $Math.atan$ peut être long. Une astuce, due à Sedgewick, consiste à approximer cette valeur par une fonction $theta(P_1, P_2)$ plus rapide à calculer et telle que $\theta < \theta'$ si et seulement si $theta(P_1, P_2) < theta(P'_1, P'_2)$:

```
static float theta (Point p1, Point p2) {
  float t; int dx = p2.x - p1.x; int dy = p2.y - p1.y;
  if (dx == 0 && dy == 0) t = 0;
  else t = (float) dy / (Math.abs(dx) + Math.abs(dy));
  if (dx < 0) t = 2 - t;
  else if (dy < 0) t = 4 + t;
  return t;
}
```

Pour le calcul de l'enveloppe convexe de n points P_0, P_1, $\ldots P_{n-1}$, le premier algorithme cherche d'abord un point d'ordonnée minimale, puis il consiste à trouver les points successifs de l'enveloppe dans l'ordre trigonométrique. Si $P_0, P_1, \ldots P_m$ sont déjà sur l'enveloppe convexe, le point P_{m+1} sur l'enveloppe est le P_i tel que $\overrightarrow{P_mP_i}$ fait un angle θ minimal et positif avec l'axe Ox pour les P_i pas encore sur l'enveloppe, c'est-à-dire pour i plus grand que m. D'où la fonction de calcul d'enveloppe convexe (marche de Jarvis) retournant le nombre de points sur l'enveloppe :

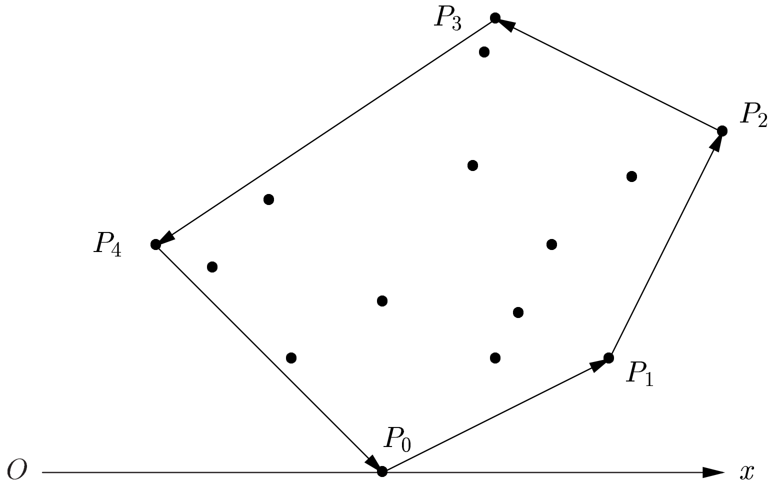


FIG. 8.13 – Enveloppe convexe

```
static int enveloppe (Point[ ] p) {
  int m = 0, n = p.length;
  if (n > 0) {
    int min = 0;
    for (int i = 1; i < n; ++i) if (p[i].y < p[min].y) min = i;
    float angle1 = 0, angle2 = 400; do {
      Point t = p[m]; p[m] = p[min]; p[min] = t;
      ++m; min = 0;
      for (int i = m; i < n; ++i) {
        float alpha = theta(p[m-1], p[i]);
        if (angle1 < alpha && alpha < angle2) {
            min = i; angle2 = alpha;
        }
      }
      angle1 = angle2; angle2 = theta(p[min], p[0]);
     } while (min != 0);
  }
  return m;
}
```

Cette fonction a une complexité quadratique (en $O(n^2)$).

Une autre technique permet d'aller plus vite. Soit P_0 un point d'ordonnée minimale. Si on trie les points selon l'angle θ_i formé par $\overrightarrow{P_0P_i}$ avec l'axe Ox, en ordre croissant, on trouve rapidement les points de l'enveloppe. En effet, on doit toujours tourner à gauche pour parcourir l'enveloppe dans l'ordre trigonométrique à partir de P_0. Ainsi si P_0, P_1, $\ldots P_m$ sont déjà sur l'enveloppe, le point P_i est le point suivant sur l'enveloppe si i est minimum tel que $i > m$ avec un angle positif formé par les vecteurs $\overrightarrow{P_{m-1}P_m}$ et $\overrightarrow{P_mP_i}$. Si l'angle est négatif, on tourne à droite en passant de P_{m-1} à P_m puis à P_i. Il faut donc retirer le point P_m de l'enveloppe trouvée jusque là. Sur l'exemple de la figure 8.14. Au début P_0, P_1, P_2 sont sur l'enveloppe convexe. Le virage P_1, P_2, P_3 est à gauche, on rajoute donc P_3 sur l'enveloppe. Mais le virage suivant P_2, P_3, P_4 se fait à droite, on doit donc retirer P_3. On considère maintenant le virage correspondant à P_1, P_2, P_4. Il tourne à gauche, on garde donc P_4. L'enveloppe est alors formée par P_0, P_1, P_2, P_4. On considère P_5 et le virage P_2, P_4, P_5, etc. Cels donnne la fonction suivante, dite marche de Graham :

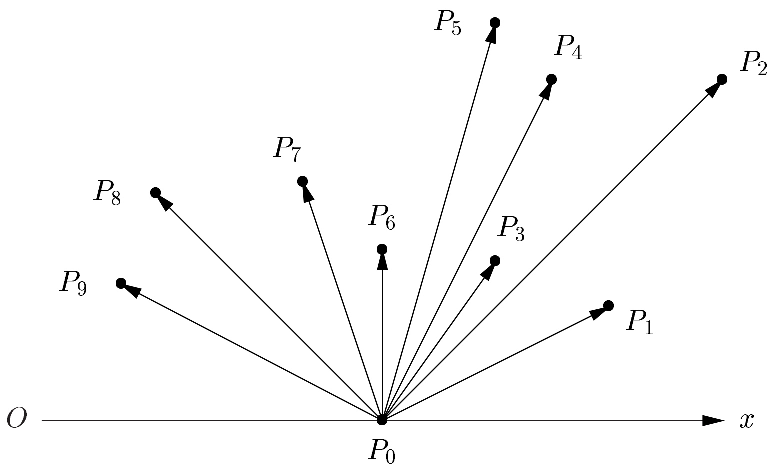


FIG. 8.14 – Enveloppe convexe

```
static int enveloppe (Point[ ] p) {
  int m = 0; int n = p.length;
  if (n <= 2) return n;
  else {
    int min = 0;
    for (int i = 1; i < n; ++i) if (p[i].y < p[min].y) min = i;
    for (int i = 0; i < n; ++i)
      if (p[i].y == p[min].y && p[i].x > p[min].x) min = i;
    Point t = p[0]; p[0] = p[min]; p[min] = t;
    tri(p); m = 2;
    for (int i = 3; i < n; ++i) {
      while (trigonometrique(p[m], p[m-1], p[i]) >= 0)
        --m;
      ++m;
      t = p[m]; p[m] = p[i]; p[i] = t;
    }
    return m+1;
  }
}
```

Cette fonction a une complexité quadratique (en $O(n \log n)$).

Pour tester si l'angle $\widehat{P_1 P_0 P_2}$ est positif, on peut calculer le produit vectoriel $\overrightarrow{P_0 P_1} \wedge \overrightarrow{P_0 P_2}$, comparer les normes, s'il l'angle est nul.

```
static int trigonometrique (Point p0, Point p1, Point p2) {
  int dx1 = p1.x - p0.x; int dy1 = p1.y - p0.y;
  int dx2 = p2.x - p0.x; int dy2 = p2.y - p0.y;
  if (dx1 * dy2 > dy1 * dx2) return 1;
  else if (dx1 * dy2 < dy1 * dx2) return -1;
  else {
    if (dx1 * dx2 < 0 || dy1 * dy2 < 0) return -1;
    else if (dx1*dx1 + dy1*dy1 > dx2*dx2 + dy2*dy2) return 1;
    else if (dx1*dx1 + dy1*dy1 == dx2*dx2 + dy2*dy2) return 0;
    else return -1;
  }
}
```

Exercice 66 Construire un interface graphique pour calculer une enveloppe convexe de n points, entrés à l'aide d'une souris.

L'algorithmique géométrique a bien d'autres développements. C'est une des parties les plus difficiles de l'algorithmique, entrainant la révision de beaucoup de notions usuelles en mathématiques. Un des problèmes courants est la présence de points singuliers ou exceptionnels. Nos programmes de calcul d'enveloppe convexe marchent-ils encore lorsque tous les points sont alignés ? C'est ce genre de problèmes qu'il faut traiter avec soin dans ce genre d'algorithmes.

Annexe A
Java

Le langage Java a été conçu par Gosling et ses collègues à Sun microsystems comme une simplification du C++ de Stroustrup [50]. Le langage, prévu initialement pour programmer de petits automatismes, s'est vite retrouvé comme le langage de programmation du World Wide Web, car son interpréteur a été intégré dans pratiquement tous les navigateurs existants. Il se distingue de C++ par son typage fort (il n'y pas par exemple de possibilité de changer le type d'une donnée sans vérifications) et par son système de récupération automatique de la mémoire (glaneur de cellules ou GC ou *garbage collector* en anglais). Très rapidement, le langage est devenu très populaire, quoiqu'ayant besoin d'une technologie de compilation plus avancée que C++. De multiples livres le décrivent, et il est un peu vain de vouloir les résumer ici. Parmi ces livres, les plus intéressants nous semblent dans l'ordre [j1,j2,j3,j4].

Comme C++, Java fait partie des langages orientés-objets, dont les ancêtres sont le Simula-67 de Dahl et Nygaard et Smalltalk de Deutsch, Kay, Goldberg et Ingalls [26], auquel il faut adjoindre les nombreuses extensions objets de langages préexistants comme Mesa, Cedar, Modula-3, Common Lisp, Caml. Cette technique de programmation peu utilisée dans notre cours (mais enseignée en cours de majeure) a de nombreux fans, et est même devenu un argument commercial pour la diffusion d'un langage de programmation.

A.1 Un exemple simple

Considérons l'exemple des carrés magiques. Un carré magique est une matrice a carrée de dimension $n \times n$ telle que la somme des lignes, des colonnes, et des deux diagonales soient les mêmes. Si n est impair, on met 1 au milieu de la dernière ligne en $a_{n,\lfloor n/2 \rfloor+1}$. On suit la première diagonale (modulo n) en mettant 2, 3, ... Dès qu'on rencontre un élément déjà vu, on monte d'une ligne dans la matrice, et on recommence. Ainsi voici des carrés magiques d'ordre 3, 5, 7

$$\begin{pmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{pmatrix} \qquad \begin{pmatrix} 11 & 18 & 25 & 2 & 9 \\ 10 & 12 & 19 & 21 & 3 \\ 4 & 6 & 13 & 20 & 22 \\ 23 & 5 & 7 & 14 & 16 \\ 17 & 24 & 1 & 8 & 15 \end{pmatrix} \qquad \begin{pmatrix} 22 & 31 & 40 & 49 & 2 & 11 & 20 \\ 21 & 23 & 32 & 41 & 43 & 3 & 12 \\ 13 & 15 & 24 & 33 & 42 & 44 & 4 \\ 5 & 14 & 16 & 25 & 34 & 36 & 45 \\ 46 & 6 & 8 & 17 & 26 & 35 & 37 \\ 38 & 47 & 7 & 9 & 18 & 27 & 29 \\ 30 & 39 & 48 & 1 & 10 & 19 & 28 \end{pmatrix}$$

Exercices 1- Montrer que les sommes sont bien les mêmes, 2- Peut-on en construire d'ordre pair ?

```
import java.io.*;
import java.lang.*;
import java.util.*;

class CarreMagique {

  final static int N = 100;
  static int[ ][ ] a = new int[N][N];

  static void init (int n){
    for (int i = 0 ; i < n; ++i)
        for (int j = 0; j < n; ++j)
            a[i][j] = 0;
  }

  static void magique (int n) {
    int i = n - 1; int j = n / 2;
    for (int k = 1; k <= n * n; ++k) {
        a[i][j] = k;
        if ((k % n) == 0)
            i = i - 1;
        else {
            i = (i + 1) % n;
            j = (j + 1) % n;
        }
    }
  }

  static void erreur (String s) {
    System.err.println ("Erreur fatale: " + s);
    System.exit (1);
  }

  static int lire () {
    BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    System.out.print ("Taille du carré magique, svp?:: ");
    int n;
    try {
      n = Integer.parseInt (in.readLine());
    }
    catch (IOException e) {
      n = 0;
    }
    catch (ParseException e) {
      n = 0;
    }
    if ((n <= 0) || (n > N) || (n % 2 == 0))
        erreur ("Taille impossible.");
    return n;
  }

  static void imprimer (int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            System.out.print (leftAligned(5, a[i][j] + " "));
        System.out.println ();
    }
```

```
  }

  static String leftAligned (int size, String s) {
    StringBuffer t = new StringBuffer (s);
    for (int i = s.length(); i < size; ++i)
        t = t.append(" ");
    return new String (t);
  }

  public static void main (String[ ] args) {
    int n = lire();
    init(n);       // inutile , mais pédagogique!
    magique(n);
    imprimer(n);
  }
}
```

D'abord, on remarque qu'un programme est une suite de directives et de déclarations de classes. Ici, nous n'avons qu'un seule classe `CarreMagique`. Auparavant nous avons quelques directives sur un nombre de classes standard dont nous allons nous servir sans utiliser la notation longue (cf plus loin). Chaque classe contient un certain nombre de déclarations de variables et de fonctions ou procédures. (En programmation objet, on parle de *méthodes* au lieu de fonctions ou de procédures. Ici nous utiliserons les deux terminologies). Une des fonctions, conventionnellement de nom `main`, est le point de départ du programme. Ignorons ses arguments pour le moment.

Dans notre exemple, la fonction `main` lit l'entier n à la console, initialise la matrice a avec des zéros, puis calcule un carré magique d'ordre n et l'imprime. Nous regardons maintenant les autres fonctions et procédures. Remarquons que les commentaires sont compris entre les séparateurs /∗ et `*/`, ou après // comme en C++.

La classe `CarreMagique` commence par la déclaration de trois variables. La première déclaration définit une constante `N` entière (*integer* en anglais, `int` en abrégé) qui représente la taille maximale du carré autorisé. Il est fréquent d'écrire les constantes avec des majuscules uniquement. Nous adoptons la convention suivante sur les noms : les noms de constantes ou de classes commencent par des majuscules, les noms de variables ou de fonctions commencent par une minuscule. On peut ne pas respecter cette convention, mais cela rendra les programmes moins lisibles. (Pour le moment, nous n'essayons pas de comprendre les mots clés `final` ou `static` — ce deuxième mot-clé reviendra très souvent dans nos programmes).

Après `N`, on déclare un tableau `a` d'entiers à deux dimensions (la partie écrite avant le symbole `=`) et on alloue un tableau $N \times N$ d'entiers qui sera sa valeur. Cette déclaration peut sembler très compliquée, mais les tableaux adoptent la syntaxe des objets (que nous verrons plus tard) et cela permettra d'initialiser les tableaux par d'autres expressions. Remarquons que les bornes du tableau ne font pas partie de la déclaration. Enfin, une troisième déclaration dit qu'on se servira d'une variable entière `n`, qui représentera l'ordre du carré magique à calculer.

Ensuite, dans la classe `CarreMagique`, nous n'avons plus que des définitions de fonctions. Considérons la première `init`, pas vraiment utile, mais intéressante puisque très simple. Le type `void` de son résultat est vide, il est indiqué avant la déclaration de son nom, comme pour les variables ou les tableaux. (En Pascal, on parlerait de procédure). Elle a un seul paramètre entier `n` (donc différent de la variable globale définie auparavant). Cette procédure remet à zéro toute la matrice a. Remarquons qu'on écrit `a[i][j]` pour son élément $a_{i,j}$ $(0 \leq i, j < n)$, et que le symbole d'affectation est `=` comme en C ou en Fortran (l'opérateur = pour le test d'égalité s'écrit `==`). Les tableaux commencent toujours à l'indice 0. Deux boucles imbriquées permettent de parcourir la matrice. L'instruction `for` a trois clauses : l'initialisation, le test pour continuer à itérer et l'incrémentation à chaque itération. On initialise la variable fraîche i à 0, on continue tant que $i < n$, et on incrémente i de 1 à chaque itération (`++` et `--` sont les symboles d'incrémentation et de décrémentation).

Considérons la fonction `imprimer`. Elle a la même structure, sauf que nous imprimons chaque élément sur 5 caractères cadrés à gauche. Les deux fonctions de librairie `System.out.print` et `System.out.println` permettent d'écrire leur paramètre (avec un retour à la ligne dans le cas de la deuxième). Le paramètre est quelconque et peut même ne pas exister, c'est le cas ici pour `println`. Il est trop tôt pour expliquer le détail de `leftAligned`, introduit ici pour rendre l'impression plus jolie, et supposons que l'impression est simplement déclenchée par

```
System.out.print (a[i][j] + " ");
```

Alors, que veut dire `a[i][j] + " "` ? A gauche de l'opérateur `+`, nous avons l'entier $a_{i,j}$ et à droite une chaîne de caractères ! Il ne s'agit bien sûr pas de l'addition sur les entiers, mais de la concaténation des chaînes de caractères. Donc `+` transforme son argument de gauche dans la chaîne de caractères qui le représente et ajoute au bout la chaîne `" "` contenant un espace blanc. La procédure devient plus claire. On imprime tous les éléments $a_{i,j}$ séparés par un espace blanc avec un retour à la ligne en fin de ligne du tableau. La fonction compliquée `leftAligned` ne fait que justifier sur 5 caractères cette impression (il n'existe pas d'impression formattée en Java). En conclusion, on constate que l'opérateur `+` est *surchargé*, car il a deux sens en Java : l'addition arithmétique, mais aussi la concaténation des chaînes de caractères dès qu'un de ses arguments est une chaîne de caractères. C'est le seul opérateur surchargé (contrairement à C++ qui autorise tous ses opérateurs à être surchargés).

La procédure `erreur` prend comme argument la chaîne de caractères `s` et l'imprime précédée d'un message insistant sur le coté fatal de l'erreur. Ici encore, on voit l'utilisation de `+` pour la concaténation de deux chaînes de caractères. Puis, la procédure fait un appel à la fonction système `exit` qui arrête l'exécution du programme avec un code d'erreur (0 voulant dire arrêt normal, tout autre valeur un arrêt anormal). (Plus tard, nous verrons qu'il y a bien d'autres manières de générer un message d'erreur, avec les exceptions ou les erreurs pré-définies).

La fonction `lire` qui n'a pas d'arguments retourne un entier lu à la console. La fonction commence par la déclaration d'une variable entière `n` qui contiendra le résultat retourné. Puis, une ligne cryptique (à apprendre par cœur) permet de dire que l'on va faire une lecture (avec tampon) à la console. On imprime un message (sans retour à la ligne) demandant de rentrer la taille voulue pour le carré magique, et on lit par `readLine` une chaîne de caractère entrée à la console. Cette chaîne est convertie en entier par la fonction `parseInt` et le tout est rangé dans la variable `n`. Si une erreur se produit au cours de la lecture, on récupère cette erreur et on positionne `n` à zéro. L'instruction `try` permet de délimiter un ensemble d'instruction où on pourra récupérer une exception par un `catch` qui spécifie les exceptions attendues et l'action à tenir en conséquence. Tous les langages modernes ont un système d'exceptions, qui permet de séparer le traitement des erreurs du traitement normal d'un groupe d'instructions. Notre fonction `lire` finit par tester si $0 \leq n < N$ et si n est impair (c'est à dire $n \bmod 2 \neq 0$). Enfin, la fonction renvoie son résultat.

Il ne reste plus qu'à considérer le coeur de notre problème, la construction d'un carré magique d'ordre n impair. On remarquera que la procédure est bien courte, et que l'essentiel de notre programme traite des entrées-sorties. C'est un phénomène général en informatique : l'algorithme est à un endroit très localisé, mais très critique, d'un ensemble bien plus vaste. On a vu que pour construire le carré, on démarre sur l'élément $a_{n,\lfloor n/2 \rfloor+1}$. On y met la valeur 1. On suit une parallèle à la première diagonale (modulo n), en déposant 2, 3, ..., n. Quand l'élément suivant de la matrice est non vide, on revient en arrière et on recommence sur la ligne précédente, jusqu'à remplir tout le tableau. Comme on sait exactement quand on rencontre un élément non vide, c'est à dire quand la valeur rangée dans la matrice est un multiple de n, la procédure devient remarquablement simple. (Remarque syntaxique : le point-virgule avant le `else` ferait hurler tout compilateur Pascal. En Java comme en C, le point-virgule fait partie de l'instruction. Simplement toute expression suivie de point-virgule devient une instruction. Pour composer plusieurs instructions en séquence, on les concatène entre des accolades comme dans la deuxième alternative du `if` ou dans l'instruction `for`).

Remarquons que le programme serait plus simple si au lieu de lire n sur la console, on le prenait en arguments de `main`. En effet, l'argument de `main` est un tableau de chaînes de caractères, qui sont les différents arguments pour lancer le programme Java sur la ligne de commande. Alors on supprimerait `lire` et `main` deviendrait :

```
public static void main (String[ ] args) {

  if (args.length < 1)
      erreur ("Il faut au moins un argument.");
  n = Integer.parseInt(args[0]);
  init(n);       // inutile , mais pédagogique!
  magique(n);
  imprimer(n);
}
```

A.2 Quelques éléments de Java

A.2.1 Symboles, séparateurs, identificateurs

Les identificateurs sont des séquences de lettres et de chiffres commençant par une lettre. Les identificateurs sont séparés par des espaces, des caractères de tabulation, des retours à la ligne ou par des caractères spéciaux comme `+`, `-`, `*`. Certains identificateurs ne peuvent être utilisés pour des noms de variables ou procédures, et sont réservés pour des *mots clés* de la syntaxe, comme `class`, `int`, `char`, `for`, `while`, ...

A.2.2 Types primitifs

Les entiers ont le type `byte`, `short`, `int` ou `long`, selon qu'on représente ces entiers signés sur 8, 16, 32 ou 64 bits. On utilise principalement `int`, car toutes les machines ont des processeurs 32 bits, et bientôt 64 bits. Attention : il y a 2 conventions bien spécifiques sur les nombres entiers : les nombres commençant par `0x` sont des nombres hexadécimaux. Ainsi `0xff` vaut 255. De même, sur une machine 32 bits, `0xffffffff` vaut -1. Les constantes entières longues sont de la forme `1L`, `-2L`. Les plus petites et plus grandes valeurs des entiers sont `Integer.MIN_VALUE` $= -2^{31}$, `Integer.MAX_VALUE` $= 2^{31} - 1$; les plus petites et plus grandes valeurs des entiers longs sont `Long.MIN_VALUE` $= -2^{63}$, `Long.MAX_VALUE` $= 2^{63} - 1$; les plus petites et plus grandes valeurs des entiers sur un octet sont `Byte.MIN_VALUE` $= -128$, `Byte.MAX_VALUE` $= 127$; etc.

Les réels ont le type `float` ou `double`. Ce sont des nombres flottants en simple ou double précision. Les constantes sont en notation décimale `3.1416` ou en notation avec exposant `31.416e-1` et respectent la norme IEEE 754. Par défaut les constantes sont prises en double précision, `3.1416f` est un flottant en simple précision. Les valeurs minimales et maximales sont `Float.MIN_VALUE` et `Float.MAX_VALUE`. Il existe aussi les constantes de la norme IEEE, `Float.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY`, `Float.NaN`, etc.

Les booléens ont le type `boolean`. Les constantes booléennes sont `true` et `false`.

Les caractères sont de type `char`. Les constantes sont écrites entre apostrophes, comme `'A'`, `'B'`, `'a'`, `'b'`, `'0'`, `'1'`, `' '`. Le caractère apostrophe se note `'\''`, et plus généralement il y a des conventions pour des caractères fréquents, `'\n'` pour *newline*, `'\r'` pour retour-charriot, `'\t'` pour tabulation, `'\\'` pour `\`. Attention : les caractères sont codés sur 16 bits, avec le standard international Unicode. On peut aussi écrire un caractère par son code `'\u0'` pour le caractère nul (code 0).

Les constantes chaînes de caractères sont écrites entre guillemets, comme dans `"Pierre et Paul"`. On peut mettre des caractères spéciaux à l'intérieur, par exemple `"Pierre\net\nPaul\n"` qui s'imprimera sur trois lignes. Pour mettre un guillemet dans une chaîne, on écrit `\"`. Si les constantes de type chaînes de caractères ont une syntaxe spéciale, la classe `String` des châines de caractères n'est pas un type primitif.

En Java, il n'y a pas de type énuméré. On utilisera des constantes normales pour représenter de tels types. Par exemple :

```
final static int BLEU = 0, BLANC = 1, ROUGE = 2;
int c = BLANC;
```

A.2.3 Expressions

Expressions élémentaires

Les expressions arithmétiques s'écrivent comme en mathématiques. Les opérateurs arithmétiques sont `+`, `-`, `*`, `/`, et `%` pour modulo. Les opérateurs logiques sont `>`, `>=`, `<`, `<=`, `==` et `!=` pour faire des comparaisons (le dernier signifiant $\neq$). Plus intéressant, les opérateurs `&&` et `||` permettent d'évaluer de la gauche vers la droite un certain nombre de conditions (en fait toutes les expressions s'évaluent de la gauche vers la droite à la différence de C ou de Caml dont l'ordre peut être laissé à la disposition de l'optimiseur). Une expression logique (encore appelée booléenne) peut valoir `true` ou `false`. La négation est représentée par l'opérateur `!`. Ainsi

```
(i < N) && (a[i] != '\n') && !exception
```

donnera la valeur `true` si `i` $<$ `N` et si `a[i]` $\neq$ *newline* et si `exception` $=$ `false`. Son résultat sera `false` dès que `i` $\geq$ `N` sans tester les autres prédicats de cette conjonction, ou alors si `i` $<$ `N` et `a[i]` $=$ newline, ...

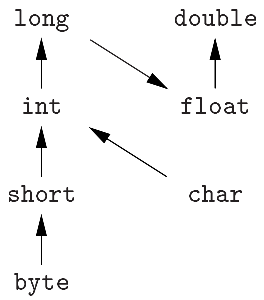


FIG. A.1 – Conversions implicites

Conversions

Il est important de bien comprendre les règles de conversions implicites dans l'évaluation des expressions. Par exemple, si `f` est réel, et si `i` est entier, l'expression `f + i` est un réel qui s'obtient par la conversion implicite de `i` vers un `float`. Certaines conversions sont interdites, comme par exemple indicer un tableau par un nombre réel. En général, on essaie de faire la plus petite conversion permettant de faire l'opération (cf. figure A.1). Ainsi un caractère n'est qu'un petit entier. Ceci permet de faire facilement certaines fonctions comme la fonction qui convertit une chaîne de caractères ASCII en un entier (`atoi` est un raccourci pour *Ascii To Integer*)

```
static int atoi (String s)
{
     int n = 0;
     for (int i = 0; i < s.length(); ++i)
         n = 10 * n + (s.charAt(i) - ’0’);
     return n;
}
```

On peut donc remarquer que `s.charAt(i) - ’0’` permet de calculer l'entier qui représente la différence entre le code de `s.charAt(i)` et celui de `’0’`. N'oublions pas que cette fonction est plus simplement calculée par `Integer.parseInt(s)`.

Les conversions implicites suivent la figure A.1. Pour toute opération, on convertit toujours au le plus petit commun majorant des types des opérandes. Des conversions explicites sont aussi possibles, et recommandées dans le doute. On peut les faire par l'opération de coercion (*cast*) suivante

(*type-name*) `expression`

L'expression est alors convertie dans le type indiqué entre parenthèses devant l'expression. L'opérateur `=` d'affectation est un opérateur comme les autres dans les expressions. Il subit donc les mêmes lois de conversion. Toutefois, il se distingue des autres opérations par le type du résultat. Pour un opérateur ordinaire, le type du résultat est le type commun obtenu par conversion des deux opérandes. Pour une affectation, le type du résultat est le type de l'expression à gauche de l'affectation. Il faut donc faire une conversion explicite sur l'expression de droite pour que le résultat soit cohérent avec le type de l'expression de gauche. Enfin, dans les appels de fonctions, il y a aussi une opération similaire à une affectation pour passer les arguments, et donc des conversions des arguments sont possibles.

Affectation

Quelques opérateurs sont moins classiques : l'affectation, les opérations d'incrémentation et les opérations sur les *bits*. L'affectation est un opérateur qui rend comme valeur la valeur affectée à la partie gauche. On peut donc écrire simplement

```
x = y = z = 1;
```

pour

```
x = 1; y = 1; z = 1;
```

Une expression qui contient une affectation modifie donc la valeur d'une variable pendant son évaluation. On dit alors que cette expression a un *effet de bord*. Les effets de bord sont à manipuler avec précautions, car leur effet peut dépendre d'un ordre d'évaluation très complexe. Il est par exemple peu recommandé de mettre plus qu'un effet de bord dans une expression.

Expressions d'incrémentation

D'autres opérations dans les expressions peuvent changer la valeur des variables. Les opérations de pré-incrémentation, de post-incrémentation, de pré-décrémentation, de post-décrémentation permettent de donner la valeur d'une variable en l'incrémentant ou la décrémentant avant ou après de lui ajouter ou retrancher 1. Supposons que `n` vaille 5, alors le programme suivant

```
x = ++n;
y = n++;
z = --n;
t = n--;
```

fait passer `n` à 6, met 6 dans `x`, met 6 dans `y`, fait passer `n` à 7, puis retranche 1 à `n` pour lui donner la valeur 6 à nouveau, met cette valeur 6 dans `z` et dans `t`, et fait passer `n` à 5. Plus simplement, on peut écrire simplement

```
++i;
j++;
```

pour

```
i = i + 1;
j = j + 1;
```

De manière identique, on pourra écrire

```
if (c != ' ')
    c = s.charAt.(i++);
```

pour

```
if (c != ' ') {
    c = s.charAt.(i);
    ++i;
}
```

En règle générale, il ne faut pas abuser des opérations d'incrémentation. Si c'est une commodité d'écriture comme dans les deux cas précédents, il n'y a pas de problème. Si l'expression devient incompréhensible et peut avoir plusieurs résultats possibles selon un ordre d'évaluation dépendant de l'implémentation, alors il ne faut pas utiliser ces opérations et on doit casser l'expression en plusieurs morceaux pour séparer la partie effet de bord.

On ne doit pas faire trop d'effets de bord dans une même expression

Expressions sur les bits

Les opérations sur les *bits* peuvent se révéler très utiles. On peut faire `&` (*et* logique), `|` (*ou* logique), `^` (*ou* exclusif), `<<` (décalage vers la gauche), `>>` (décalage vers la droite), `~` (complément à un). Ainsi

```
x = x & 0xff;
y = y | 0x40;
```

mettent dans `x` les 8 derniers bits de `x` et positionne le 6$^{\text{ème}}$ bit à partir de la droite dans `y`. Il faut bien distinguer les opérations logiques `&&` et `||` à résultat booléens 0 ou 1 des opérations `&` et `|` sur les *bits* qui donnent toute valeur entière. Par exemple, si `x` vaut 1 et `y` vaut 2, `x & y` vaut 0 et `x && y` vaut 1.

Les opérations `<<` et `>>` décalent leur opérande de gauche de la valeur indiquée par l'opérande de droite. Ainsi `3 << 2` vaut 12, et `7 >> 2` vaut 1. Les décalages à gauche introduisent toujours des zéros sur les bits de droite. Pour les bits de gauche dans le cas des décalages à droite, c'est dépendant de la machine ; mais si l'expression décalée est `unsigned`, ce sont toujours des zéros.

Le complément à un est très utile dans les expressions sur les *bits*. Il permet d'écrire des expressions indépendantes de la machine. Par exemple

```
x = x & ~0x7f;
```

remet à zéro les 7 bits de gauche de `x`, indépendamment du nombre de bits pour représenter un entier. Une notation, supposant des entiers sur 32 bits et donc dépendante de la machine, serait

```
x = x & 0xffff8000;
```

Autres expressions d'affectation

A titre anecdotique, les opérateurs d'affectation peuvent être plus complexes que la simple affectation et permettent des abréviations parfois utiles. Ainsi, si $\oplus$ est un des opérateurs `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, ou `|`,

$e_1 \oplus\!= e_2$

est un raccourci pour

$e_1 = e_1 \oplus e_2$

Expressions conditionnelles

Parfois, on peut trouver un peu long d'écrire

```
if (a > b)
    z = a;
else
    z = b;
```

L'expression conditionnelle

e_1 `?` e_2 `:` e_3

évalue e_1 d'abord. Si vrai, le résultat est e_2, sinon e_3. Donc le maximum de `a` et `b` peut s'écrire

```
z = (a > b) ? a : b;
```

Les expressions conditionnelles sont des expressions comme les autres et vérifient les lois de conversion. Ainsi si e_2 est flottant et e_3 est entier, le résultat sera toujours flottant.

Précédence et ordre d'évaluation

Certains opérateurs ont des précédences évidentes, et limitent l'utilisation des parenthèses dans les expressions. D'autres sont moins clairs. Voici la table donnant les précédences dans l'ordre décroissant et le parenthésage en cas d'égalité

<table>
<tr><th>Opérateurs</th><th>Associativité</th></tr>
<tr><td><code>() [ ] -> .</code></td><td>gauche à droite</td></tr>
<tr><td><code>! ~ ++ -- + - = * &</code> (type) <code>sizeof</code></td><td>droite à gauche</td></tr>
<tr><td><code>* / %</code></td><td>gauche à droite</td></tr>
<tr><td><code>+ -</code></td><td>gauche à droite</td></tr>
<tr><td><code><< >></code></td><td>gauche à droite</td></tr>
<tr><td><code>< <= > >=</code></td><td>gauche à droite</td></tr>
<tr><td><code>== !=</code></td><td>gauche à droite</td></tr>
<tr><td><code>&</code></td><td>gauche à droite</td></tr>
<tr><td><code>^</code></td><td>gauche à droite</td></tr>
<tr><td><code>|</code></td><td>gauche à droite</td></tr>
<tr><td><code>&&</code></td><td>gauche à droite</td></tr>
<tr><td><code>||</code></td><td>gauche à droite</td></tr>
<tr><td><code>?:</code></td><td>droite à gauche</td></tr>
<tr><td><code>= += -= /= %= &= ^= |= <<= >>=</code></td><td>droite à gauche</td></tr>
<tr><td><code>,</code></td><td>gauche à droite</td></tr>
</table>

En règle générale, il est conseillé de mettre des parenthèses si les précédences ne sont pas claires. Par exemple

```
if ((x & MASK) == 0) ...
```

A.2.4 Instructions

Toute expression suivie d'un point-virgule devient une instruction. Ainsi

```
x = 3;
++i;
System.out.print(...);
```

sont des instructions (une expression d'affectation, d'incrémentation, un appel de fonction suivi de point-virgule). Donc point-virgule fait partie de l'instruction, et n'est pas un séparateur comme en Pascal. De même, les accolades `{ }` permettent de regrouper des instructions en séquence. Ce qui permet de mettre plusieurs instructions dans les alternatives d'un `if` par exemple.

Les instructions de contrôle sont à peu près les mêmes qu'en Pascal. D'abord les instructions conditionnelles `if` sont de la forme

if (E)
    S_1

ou

if (E)
    S_1
else
    S_2

Remarquons bien qu'une instruction peut être une expression suivie d'un point-virgule (contrairement à Pascal). Donc l'instruction suivante est complètement licite

```
if (x < 10)
    c = '0' + x;
```

```
else
    c = 'a' + x - 10;
```

Il y a la même convention qu'en Pascal pour les `if` emboîtés. Le `else` se rapportant toujours au `if` le plus proche. Une série de `if` peut être remplacée par une instruction de sélection par cas, c'est l'instruction `switch`. Elle a la syntaxe suivante

```
switch (E) {
    case c1: instructions
    case c2: instructions
    ...
    case cn: instructions
    default: instructions
}
```

Cette instruction a une idiosyncrasie bien particulière. Pour sortir de l'instruction, il faut exécuter une instruction `break`. Sinon, le reste de l'instruction est fait en séquence. Cela permet de regrouper plusieurs alternatives, mais peut être particulièrement dangereux. Par exemple, le programme suivant

```
switch (c) {
    case '\t':
    case ' ':
        ++ nEspaces;
        break;
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        ++ nChiffres;
        break;
    default:
        ++ nAutres;
        break;
}
```

permet de factoriser le traitement de quelques cas. On verra que l'instruction `break` permet aussi de sortir des boucles. Il faudra donc bien faire attention à ne pas oublier le `break` à la fin de chaque cas, et à ce que `break` ne soit pas intercepté par une autre instruction.

Les itérations sont réalisées par les instructions `for`, `while`, et `do...while`. L'instruction `while` permet d'itérer tant qu'une expression booléenne est vraie, on itère l'instruction S tant que la condition E est vraie par

```
while (E)
     S
```

et on fait de même en effectuant au moins une fois la boucle par

```
do
     S
while (E);
```

L'instruction d'itération la plus puissante est l'instruction `for`. Sa syntaxe est

```
for (E1; E2; E3)
    S
```

qui est équivalente à

```
E1;
while (E2) {
     S;
     E3;
}
```

Elle est donc beaucoup plus complexe qu'en Pascal ou Caml et peut donc ne pas terminer, puisque les expressions E_2 et E_3 sont quelconques. On peut toujours écrire des itérations simples :

```
for (i = 0; i < 100; ++i)
     a[i] = 0;
```

mais l'itération suivante pour une recherche avec hachage est plus complexe

```
for (int i = h(x); i != -1; i = col[i])
    if (x.equals(nom[i]))
       return tel[i];
```

Nous avons vu que l'instruction `break` permet de sortir d'une instruction `switch`, mais aussi de toute instruction d'itération. De même, l'instruction `continue` permet de passer brusquement à l'itération suivante. Ainsi

```
for (i = 0; i < n; ++i) {
    if (a[i] < 0)
       continue;
    ...
}
```

C'est bien commode quand le cas $a_i \geq 0$ est très long. Les `break` et `continue` peuvent préciser l'étiquette de l'itération qu'elles référencent. Ainsi, dans l'exemple suivant, on déclare une étiquette devant l'instruction `while`, et on sort des deux itérations comme suit :

```
boucle:
  while (E) {
    for (i = 0; i < n; ++i) {
      if (a[i] < 0)
          break boucle;
      ...
    }
  }
```

Finalement, il n'y a pas d'instruction `goto`. Il y a toutefois des exceptions que nous verrons plus tard.

A.2.5 Procédures et fonctions

La syntaxe des fonctions et procédures a déjà été vue dans l'exemple du carré magique. Chaque classe contient une suite linéaire de fonctions ou procédures, non emboîtées. Par convention, le début de l'exécution est donné à la procédure publique `main` qui prend un tableau de chaînes de caractères comme argument. Pour déclarer une fonction, on déclare d'abord sa signature, c'est à dire le type de son résultat et des arguments, puis son corps, c'est à dire la suite d'instructions qui la réalisent entre accolades. Ainsi dans

```
static int suivant (int x) {
  if (x % 2 == 1)
      return 3 * x + 1;
  else
      return x / 2;
}
```

le résultat est de type entier (à cause du mot-clé `int` avant `suivant`) et l'unique argument x est aussi entier. L'instruction

```
return e;
```

sort de la fonction en donnant le résultat e, et permet de rendre un résultat à la fonction. Dans les deux fonctions qui suivent, le résultat est vide, donc de type `void` et l'argument est entier.

```
static void test (int x) {
  while (x != 1)
      x = suivant (x);
}

static void testConjecture (int n) {
  for (int x=1; x <= n; ++x) {
      test (x);
      System.out.println (x);
  }
}
```

On calcule donc les itérations de la fonction qui renvoie $3x+1$ si x est impair, et $\lfloor x/2 \rfloor$ sinon. (En fait, on ne sait pas démontrer qu'on finit toujours avec 1 pour tout entier x de départ. Par exemple, à partir de 7, on obtient 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1).

Il peut y avoir des variables locales dans une procédure, plus exactement dans toute instruction composée entourée par des accolades. Les variables globales sont déclarées au même niveau que les procédures ou fonctions. Les variables locales peuvent être initialisées. Cela revient à faire la déclaration et l'affectation par la valeur initiale, qui peut être une expression complexe et qui est évaluée à chaque entrée dans la fonction. Les variables locales disparaissent donc quand on quitte la fonction.

Dans une fonction, on peut accéder aux variables globales et éventuellement les modifier, quoiqu'il est recommandé de ne pas faire trop d'effets de bord, mais on ne pourra passer une variable en argument pour modifier sa valeur. Prenons l'exemple de la fonction `suivant` précédente, on a changé la valeur du paramètre x dans le corps de la fonction. Mais ceci n'est vrai qu'à l'intérieur du corps de la fonction. En Java, seule la valeur du paramètre compte, on ne modifiera donc pas ainsi une variable extérieure à la fonction, passée en paramètre.

$\boxed{\text{Les paramètres des fonctions sont passés par valeur}}$

A.2.6 Classes

Une classe est à la fois la déclaration d'un type non primitif et d'une série de fonctions ou procédures associés. L'idée est de découper en petits modules l'espace des données et des fonctions. Les éléments de l'ensemble représenté par une classe sont les objets. Dans une classe, il y a une partie champs de données, et une autre qui est un ensemble de fonctions ou procédures. Dans la programmation orientée-objet, on aime parler d'objets comme des intances d'une classe, et de méthodes pour les fonctions et procédures associées ("méthodes" car ce sont souvent des méthodes d'accès aux objets de la classe que réalisent ces fonctions). Bien sûr, il peut y avoir des classes sans données ou sans méthodes. Prenons le cas d'une structure d'enregistrement classique en Pascal ou en C. On peut écrire :

```
class Date {
  int        j;   /* Jour */
  int        m;   /* Mois */
  int        a;   /* Année */
};
```

et par la suite on peut déclarer des variables, comme on le faisait avec les variables de type primitif (entier, réel, caractère ou booléen). Ainsi on note deux dates importantes

```
static final int Jan=1, Fev=2, Mar=3, Avr=4, Mai=5, Juin=6,
        Juil=7, Aou=8, Sep=9, Oct=10, Nov=11, Dec=12;

Date bastille = new Date(), berlin = new Date();

bastille.j = 14; bastille.m = Juil; bastille.a = 1789;
berlin.j = 10; berlin.m = Nov; berlin.a = 1989;
```

Nous venons de définir deux objets `bastille` et `berlin`, et pour accéder à leurs champs on utilise la notation bien connue suffixe avec un point. Le champ jour de la date de la prise de la Bastille s'obtient donc par `bastille.j`. Rien de neuf, c'est la notation utilisée en Pascal, en C, ou en Caml. Pour créer un objet, on utilise le mot-clé `new` suivi du nom de la classe et de parenthèses. (Pour les experts, les objets sont représentés par un pointeur et leur contenu se trouve dans le tas). Un objet non initialisé vaut `null`.

Nos dates sont un peu lourdes à manipuler. Très souvent, on veut une méthode paramétrée pour construire un objet d'une classe. C'est tellement fréquent qu'il y a une syntaxe particulière pour le faire. Ainsi si on écrit

```
class Date {
  int        j;   /* Jour */
  int        m;   /* Mois */
  int        a;   /* Année */

  Date (int jour, int mois, int annee) {
     this.j = jour;
     this.m = mois;
     this.a = annee;
};
```

on pourra créer les dates simplement avec

```
  static Date berlin = new Date(10, Nov, 1989),
              bastille = new Date(14, Juil, 1789);
```

Un constructeur est donc une méthode (non statique) sans nom. On indique le type de son résutat (ie le nom de la classe où il se trouve) et ses paramètres. Le corps de la fonction est quelconque, mais on ne retourne pas explicitement de valeur, puisque son résultat est toujours l'objet en cours de création. Le constructeur est utilisé derrière un mot-clé `new`. Dans le cas où il n'y a pas de constructeur explicite, le constructeur par défaut (sans arguments) réserve juste l'espace mémoire nécessaire pour l'objet construit. Remarquons que dans le constructeur, on a utilisé le mot-clé `this` qui désigne l'objet en cours de création pour bien comprendre que `j`, `m` et `a` sont des champs de l'objet construit. Le constructeur se finit donc implicitement par `return this`. Mais, ce mot-clé n'était pas vraiment utile. On aurait pu simplement écrire

```
class Date {
  int        j;   /* Jour */
  int        m;   /* Mois */
  int        a;   /* Année */

  Date (int jour, int mois, int annee) {
     j = jour;
     m = mois;
     a = annee;
};
```

Un champ peut être déclaré statique. Cela signifie qu'il n'existe qu'à un seul exemplaire dans la classe dont il fait partie. Une donnée statique est donc attachée à une classe et non aux objets de cette classe. Supposons par exemple que l'origine des temps (pour le système Unix) soit une valeur de première importance, ou que nous voulions compter le nombre de dates crées avec notre constructeur. On écrirait :

```
class Date {
  int        j;   /* Jour */
  int        m;   /* Mois */
  int        a;   /* Année */

  static final int Jan=1, Fev=2, Mar=3, Avr=4, Mai=5, Juin=6,
        Juil=7, Aou=8, Sep=9, Oct=10, Nov=11, Dec=12;

  static Date tempsZeroUnix = new Date (1, Jan, 1970);
  static int nbInstances = 0;

  Date (int jour, int mois, int annee) {
     j = jour;
     m = mois;
     a = annee;
     ++ nbInstances;
};
```

Il y a donc deux sortes de données dans une classe, les champs associés à chaque instance d'un objet (ici les jours, mois et années), et les champs uniques pour toute la classe (ici les constantes, la date temps-zéro pour Unix et le nombre d'utilisateurs). Les variables statiques sont initialisées au chargement de la classe, les autres dynamiquement en accédant aux champs de l'objet.

Considérons maintenant les méthodes d'une classe. A nouveau, elles sont de deux sortes : les méthodes dynamiques et les méthodes statiques. Dans notre cours, nous avons fortement privilégié cette deuxième catégorie, car leur utilisation est très proche de celle des fonctions ou procédures de C ou de Pascal. Les méthodes statiques sont précédées du mot-clé `static`, les méthodes dynamiques n'ont pas de préfixe. La syntaxe est celle d'une fonction usuelle. Prenons le cas de l'impression de notre classe Date.

```
class Date {
  ...
  static void imprimer (Date d) {
      System.out.print ("d = " + d.j + ", " +
                        "m = " + d.m + ", " +
                        "a = " + d.a);
  }
}
```

et on pourra imprimer avec des instructions du genre

```
  Date.imprimer (berlin);
  Date.imprimer (bastille);
```

Remarquons qu'on doit qualifier le nom de procédure par le nom de la classe, si on se trouve dans une autre classe. (Cette syntaxe est cohérente avec celle des accès aux champs de données). Dans les langages de programmation usuels, on retrouve cette notation aussi pour accéder aux fonctions de modules différents. On peut aussi écrire de même une fonction qui teste l'égalité de deux dates

```
class Date {
  ...
  static boolean equals (Date d1, Date d2) {
```

```
        return d1.j == d2.j && d1.m == d2.m && d1.a == d2.a;
    }
}
```

Jusqu'à présent, nous avions considéré des objets, des fonctions, et des classes. Les méthodes non statiques sont le béaba de la programmation objet. Quelle est l'idée ? Comme une classe, un objet a non seulement des champs de données, mais il contient aussi un vecteur de méthodes. Pour déclencher une méthode, on passe les paramètres aux méthodes de l'objet. Ces fonctions ont la même syntaxe que les méthodes dynamiques, à une exception près : elles ont aussi le paramètre implicite `this` qui est l'objet dont elles sont la méthode. (Pour accéder aux champs de l'objet, `this` est facultatif). Ce changement, qui rend plus proches les fonctions et les données, peut paraître mineur, mais il est à la base de la programmation objet, car il se combinera à la notion de sous-classe. Prenons l'exemple des deux méthodes statiques écrites précédemment. Nous pouvons les réécrire non statiquement comme suit :

```
class Date {
  ...
  void print () {
      System.out.print ("d = " + this.j + ", " +
                        "m = " + this.m + ", " +
                        "a = " + this.a);
  }

  boolean equals (Date d) {
      return this.j == d.j && this.m == d.m && this.a == d.a;
  }
}
```

ou encore sans le mot-clé `this` non nécessaire ici :

```
class Date {
  ...
  void print () {
      System.out.print ("d = " + j + ", " +
                        "m = " + m + ", " +
                        "a = " + a);
  }

  boolean equals (Date d) {
      return j == d.j && m == d.m && a == d.a;
  }
}
```

et on pourra indistinctement écrire

```
if (!Date.equals(berlin, bastille))
  Date.imprimer (berlin);
```

où

```
if (!berlin.equals(bastille))
  berlin.print ();
```

Dans la deuxième écriture, on passe l'argument (quand il existe) à la méthode correspondante de l'objet auquel appartient cette méthode. Nous avons déjà utilisé cette notation avec les fonctions prédéfinies de la librairie. Par exemple `println` est une méthode associé au flux de sortie `out`, qui lui-même est une donnée statique de la classe `System`. De même pour les chaînes de caractères : la classe `String` définit les méthodes `length`, `charAt` pour obtenir la longueur de l'objet chaîne `s` ou lire le caractère à la position i dans `s` comme suit :

```
String s = "Oh, la belle chaîne";

if (s.length() > 20)
  System.out.println (s.charAt(i));
```

Il faut savoir que la méthode (publique) spéciale `toString` peut être prise en compte par le système d'écriture standard. Ainsi, dans le cas des dates, si on avait déclaré,

```
class Date {
  ...
  public String toString () {
      return ("d = " + j + ", " +
              "m = " + m + ", " +
              "a = " + a);
  }
}
```

on aurait pu seulement écrire

```
if (!berlin.equals(bastille))
  System.out.println (berlin);
```

Enfin, dans une classe, on peut directement mettre entre accolades des instructions, éventuellement précédées par le mot-clé `static` pour initialiser divers champs à chaque création d'un objet ou au chargement de la classe.

Faisons trois remarques supplémentaires sur les objets. Primo, un objet peut être passé comme paramètre d'une procédure, mais alors seule la référence à l'objet est passée. Il n'y a pas de copie de l'objet. Donc, on peut éventuellement modifier le contenu de l'objet dans la procédure. (Pour copier un objet, on peut souvent utiliser la méthode `clone`).

Les objets ne sont jamais copiés implicitement

Deuxièmement, il n'y a pas d'instruction pour détruire des objets. Ce n'est pas grave, car le *garbage collector* (GC) récupère automatiquement l'espace mémoire des objets non utilisés. Cela est fait régulièrement, notamment quand il n'y a plus de place en mémoire. C'est un service de récupération des ordures, comme dans la vie courante. Il n'y a donc pas à se soucier de la dé-allocation des objets. Troisièmement, il est possible de surcharger les méthodes en faisant varier le type de leurs arguments ou leur nombre. Nous avons déjà vu le cas de `println` qui prenait zéro arguments ou un argument de type quelconque (qui était en fait transformé en chaîne de caractères avec la méthode `toString`). On peut déclarer très simplement de telles méthodes surchargées, par exemple dans le cas des constructeurs :

```
class Date {
  int       j;   /* Jour */
  int       m;   /* Mois */
  int       a;   /* Année */

  Date (int jour, int mois, int annee) {
      j = jour; m = mois; a = annee;

  Date (long n) {
      // Un savant calcul du jour , mois et année à partir du nombre
      // de millisecondes depuis l ' origine des temps informatiques , ie
      // le 1 janvier 1970.
  }

  Date () {
```

```
        // idem avec le temps courant \progt{System.currentTimeMillis}
    }
};
```

Le constructeur adéquat sera appelé en fonction du type ou du nombre de ses arguments, ici avec trois entiers désignant les jour, mois et année, ou avec un seul entier long donnant le nombre de millisecondes depuis le début des temps informatiques, ou sans argument pour avoir la date du jour courant. (Remarque : compter le temps avec des millisecondes sur 64 bits reporte le problème du *bug* de l'an 2000 à l'an 10^8, mais ce n'est pas le cas dans certains systèmes Unix où le nombre de secondes sur 32 bits depuis 1970 reporte le problème à l'an 2038 !). Il faut faire attention aux abus de surcharge, et introduire une certaine logique dans son utilisation, sinon les programmes deviennent rapidement incompréhensibles. Pire, cela peut être redoutable lorsqu'on combine surcharge et héritage (cf. plus loin).

La surcharge est résolue statiquement à la compilation.

A.2.7 Sous-classes

Le cours n'utilise pratiquement pas la notion de sous-classe, car cette notion intervient surtout si on fait de la programmation orientée-objet. Nous mentionnons toutefois brièvement cette notion. Une classe peut étendre une autre classe. Par exemple, la classe des dates pourrait être refaite en deux systèmes de dates : grégorien ou révolutionnaire en fonction du nombre de millisecondes (éventuellement négatif) depuis l'origine des temps informatiques; ou bien une classe étend une classe standard déjà fournie comme celle des *applets* pour programmer un navigateur, ou la classe MacLib pour faire le graphique élémentaire de ce cours; etc. Prenons l'exemple ultra classique de la classe des points éventuellement colorés, exemple qui a l'avantage d'être court et suffisant pour expliquer la problématique rencontrée. Un point est représenté par la paire (x, y) de ses coordonnées

```
class Point {
    int x, y;

  Point (int x0, int y0) {
    x = x0; y = y0;
  }

  public String toString () {
    return "(" + x + ", " + y +")";
  }

  void move (int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

On considère deux méthodes pour convertir un point en chaîne de caractères (pour l'impression) et une autre `move` pour bouger un point, cette dernière méthode étant une procédure qui renvoie donc le type vide. On peut utiliser des objets de cette classe en écrivant des instructions de la forme

```
Point p = new Point(1, 2);
System.out.println (p);
```

```
 p.move (-1, -1);
 System.out.println (p);
```

Définissons à présent la classe des points colorés avec un champ supplémentaire `couleur`, en la déclarant comme une extension, ou encore une *sous-classe*, de la classe des points, comme suit :

```
class PointAvecCouleur extends Point {
  int couleur;

  PointAvecCouleur (int x0, int y0, int c) {
    super (x0, y0); couleur = c;
  }

  public String toString () {
    return "(" + x + ", " + y + ", " + couleur + ")";
  }
}
```

Les champs x et y de l'ancienne classe des points existent toujours. Le champ couleur est juste ajouté. S'il existait déjà un champ couleur dans la classe *Point*, on ne ferait que le cacher et l'ancien champ serait toujours accessible par `super.couleur` grâce au mot-clé `super`. Dans la nouvelle classe, nous avons un constructeur qui prend un argument supplémentaire pour la couleur. Ce constructeur doit toujours commencer par un appel à un constructeur de la super-classe (la classe des points). Si on ne met pas d'appel explicite à un constructeur de cette classe, l'instruction `super()` est faite implicitement, et ce constructeur doit alors exister dans la super classe. Enfin, la méthode `toString` est redéfinie pour prendre en compte le nouveau champ pour la couleur. On utilise les points colorés comme suit

```
PointAvecCouleur q = new PointAvecCouleur (3, 4, 0xfff);
System.out.println (q);
q.move(10,-1);
System.out.println (q);
```

La classe des points colorés a donc *hérité* des champs x et y et de la méthode `move` de la classe des points, mais la méthode `toString` a été redéfinie. On n'a eu donc qu'à programmer l'incrément entre les points normaux et les points colorés. C'est le principe de base de la programmation objet : le contrôle des programmes est dirigé par les données et leurs modifications. Dans la programmation classique, on doit changer le corps de beaucoup de fonctions ou de procédures si on change les déclarations des données, car la description d'un programme est donné par sa fonctionnalité.

Une méthode ne peut redéfinir qu'une méthode de même type pour ses paramètres et résultat. Plus exactement, elle peut redéfinir une méthode d'une classe plus générale dont les arguments ont un type plus général. Il est interdit de redéfinir les méthodes préfixées par le mot-clé `final`. On ne peut non plus donner des modificateurs d'accès plus restrictifs, une méthode publique devant rester publique. Une classe hérite d'une seule classe (héritage simple). Des langages comme Smalltalk ou C++ autorisent l'héritage multiple à partir de plusieurs classes, mais les méthodes sont alors un peu plus délicates à implémenter. L'héritage est bien sûr transitif. D'ailleurs toutes les classes Java héritent d'une unique classe `Object`.

Il se pose donc le problème de savoir quelle méthode est utilisée pour un objet donné. C'est toujours la méthode la plus précise qui peut s'appliquer à l'objet et à ses arguments qui est sélectionnée. Remarquons que ceci n'est pas forcément dans le corps du programme chargé au moment de la référence, puisqu'une nouvelle classe peut être chargée, et contenir la méthode qui sera utilisée.

La résolution des méthodes dynamiques est faite à l'exécution.

Enfin, il y a une façon de convertir un objet en objet d'une super-classe ou d'une sous-classe avec la notation des conversions explicites (déjà vue pour le cas des valeurs numériques). Par exemple

```
Point p = new Point (10, 10);
PointAvecCouleur q = new PointAvecCouleur (20, 20, ROUGE);
Point p1 = q;
PointAvecCouleur q1 = (PointAvecCouleur) p;
PointAvecCouleur q2 = (PointAvecCouleur) p1;
```

Aller vers une classe plus générale ne pose pas en général de difficultés, et le faire explicitement peut forcer la résolution des surcharges de noms de fonctions, mais l'objet reste le même. Si on convertit vers une sous-classe plus restrictive, la machine virtuelle Java vérifie à l'exécution et peut lever l'exception `ClassCastException`. Dans l'exemple précédent seule l'avant-dernière ligne lèvera cette exception.

A.2.8 Tableaux

Les tableaux sont des objets comme les autres. Un champ `length` indique leur longueur. L'accès aux éléments du tableau a s'écrit avec des crochets, `a[i-1]` représente i-ème élément. Les tableaux n'ont qu'une seule dimension, un tableau à deux dimensions est considéré comme un tableau dont tous les éléments sont des tableaux à une dimension, etc. Si on accède en dehors du tableau, une exception est levée. La création d'un tableau se fait avec le mot-clé `new` comme pour les objets, mais il existe une facilité syntaxique pour créer des tableaux à plusieurs dimensions. Voici par exemple le calcul de la table de vérité de l'union de deux opérateurs booléens :

```
static boolean[ ][ ] union (boolean[ ][ ] a, boolean[ ][ ] b) {
  boolean[ ][ ] c = new boolean [2][2];
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
        c[i][j] = a[i][j] || b[i][j];
  return c;
}
```

Pour initialiser un tableau, on peut le faire avec des constantes litérales :

```
boolean[ ][ ] intersection = {{true, false},{false, false}};
boolean[ ][ ] ouExclusif = {{false, true},{true, false}};
```

Enfin, on peut affecter des objets d'une sous-classe dans un tableau d'éléments d'une classe plus générale.

A.2.9 Exceptions

Les exceptions sont des objets de toute sous-classe de la classe `Exception`. Il existe aussi une classe `Error` moins utilisée pour les erreurs système. Toutes les deux sont des sous-classes de la classe `Throwable`, dont tous les objets peuvent être appliqués à l'opérateur `throw`, comme suit :

```
throw e;
```

Ainsi on peut écrire en se servant de deux constructeurs de la classe `Exception` :

```
throw new Exception();
throw new Exception ("Accès interdit dans un tableau");
```

Heureusement, dans les classes des librairies standard, beaucoup d'exceptions sont déjà pré-définies, par exemple `IndexOutOfBoundsExeption`. On récupère une exception par l'instruction `try` ...`catch`. Par exemple

```
try {
  // un programme compliqué
} catch ( IOException e) {
  // essayer de réparer cette erreur d'entrée/sortie
}
catch ( Exception e) {
  // essayer de réparer cette erreur plus générale
}
```

Si on veut faire un traitement uniforme en fin de l'instruction `try`, que l'on passe ou non par une exception, que le contrôle sorte ou non de l'instruction par une rupture de séquence comme un `return`, `break`, etc, on écrit

```
try {
  // un programme compliqué
} catch ( IOException e) {
  // essayer de réparer cette erreur d'entrée/sortie
}
catch ( Exception e) {
  // essayer de réparer cette erreur plus générale
}
finally {
  // un peu de nettoyage
}
```

Il y a deux types d'exceptions. On doit déclarer les *exceptions vérifiées* derrière le mot-clé `throws` dans la signature des fonctions qui les lèvent. Ce n'est pas la peine pour les *exceptions non vérifiées* qui se reconnaissent en appartenant à une sous-classe de la classe `RuntimeException`. Ainsi

```
static int lire () throws IOException, ParseException {
  int n;
  BufferedReader in =
      new BufferedReader(new InputStreamReader(System.in));

  System.out.print ("Taille du carré magique, svp?:: ");
  n = Integer.parseInt (in.readLine());
  if ((n <= 0) || (n > N) || (n % 2 == 0))
      erreur ("Taille impossible.");
  return n;
}
```

aurait pu être écrit dans l'exemple du carré magique.

A.2.10 Entrées-Sorties

Nous ne considérons que quelques instructions simples permettant de lire ou écrire dans une fenêtre texte ou avec un fichier. `System.in` et `System.out` sont deux champs statiques (donc uniques) de la classe système, qui sont respectivement des `InputStream` et `PrintStream`. Dans cette dernière classe, il y a notamment les méthodes : `flush` qui vide les sorties non encore effectuées, `print` et `println` qui impriment sur le terminal leur argument avec éventuellement un retour à la ligne. Pour l'impression, l'argument de ces fonctions est quelconque, (éventuellement vide pour la deuxième). Elles sont surchargées sur pratiquement tous les types, et transforment leur argument en chaîne de caractères. Ainsi :

`System.out.println ("x= ")`		`x =` *newline*
`System.out.println (100)`		`100` *newline*
`System.out.print (3.14)`	donnent	`3.14`
`System.out.print ("3.14")`		`3.14`
`System.out.print (true)`		`true`
`System.out.print ("true")`		`true`

Les méthodes des classes `InputStream` et `PrintStream` lisent ou impriment des octets (`byte`). Il vaut mieux faire des opérations avec des caractères Unicode (sur 16 bits, qui comprennent tous les caractères internationaux). Pour cela, au lieu de fonctionner avec les flux d'octets (*Stream* tout court), on utilise les classes des *Reader* ou des *Writer*, comme `InputStreamReader` et `OutputStreamWriter`, qui manipulent des flux de caractères. Dans ces classes, il existe de nombreuses méthodes ou fonctions. Ici, nous considérons les entrées-sorties avec un tampon (*buffer* en anglais), qui sont plus efficaces, car elles regroupent les opérations d'entrées-sorties. C'est pourquoi, on écrit souvent la ligne cryptique :

```
struct Noeud {
   BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
```

qui construit, à partir du flux de caractères `System.in` (désignant la fenêtre d'entrée de texte par défaut), un flux de caractères, puis un flux de caractères avec tampon (plus efficace). Dans cette dernière classe, on lit les caractères par `read()` ou `readLine()`. Par convention, `read()` retourne -1 quand la fin de l'entrée est détectée (comme en C). C'est pourquoi le type de son résultat est un entier (et non un caractère), puisque -1 ne peut pas être du type caractère. Pour lire l'entrée terminal et l'imprimer immédiatement, on fait donc :

```
import java.io.*;

static void copie () throws Exception {

  BufferedReader in =
      new BufferedReader(new InputStreamReader(System.in));
  int c;

  while ((c = in.read()) != -1)
      System.out.println(c);
  System.out.flush();
  }
```

Remarquons l'idiome pour lire un caractère. Il s'agit d'un effet de bord dans le prédicat du `while`. On fait l'affectation `c = in.read()` qui retourne comme résultat la valeur de la partie droite (le caractère lu) et on teste si cette valeur vaut -1.

On peut aussi manipuler des fichiers, grâce aux classes fichiers `File`. Ainsi le programme précédent se réécrit pour copier un fichier source de nom `s` dans un autre destination de nom `d`.

```
import java.io.*;

static void copieDeFichiers (String s, String d) throws Exception {

  File src = new File (s);
  if ( !src.exists() || !src.canRead())
```

```
        erreur ("Lecture impossible de " + s);
    BufferedReader in = new BufferedReader(new FileReader(src));

    File dest = new File (d);
    if ( !dest.canWrite())
        erreur ("Ecriture impossible de " + d);
    BufferedWriter out = new BufferedWriter(new FileWriter(dest));

    maCopie (in, out);
    in.close();
    out.close();
  }

  static void maCopie (BufferedReader in, BufferedWriter out)
    throws IOException {

    int c;
    while ((c = in.read()) != -1)
        out.write(c);
    out.flush();
  }
```

La procédure de copie ressemble au programme précédent, au changement près de `print` en `write`, car nous n'avons pas voulu utiliser la classe *Printer*, ce qui était faisable. Remarquons que les entrées sorties se sont simplement faites avec les fichiers en suivant un schéma quasi identique à celui utilisé pour le terminal. La seule différence vient de l'association entre le nom de fichier et les flux de caractères tamponnés. D'abord le nom de fichier est transformé en objet `File` sur lequel plusieurs opérations sont possibles, comme vérifier l'existence ou les droits d'accès en lecture ou en écriture. Puis on construit un objet de flux de caractères dans les classes `FileReader` et `FileWriter`, et enfin des objets de flux de caractères avec tampon. La procédure de copie est elle identique à celle vue précédemment.

On peut donc dire que l'entrée standard `System.in` (de la fenêtre de texte), la sortie standard `System.out` (dans la fenêtre de texte), et la sortie standard des erreurs `System.err` (qui n'a vraiment de sens que dans le système Unix) sont comme des fichiers particuliers. Les opérations de lecture ou d'écriture étant les mêmes, seule la construction du flux de caractères tamponné varie.

Enfin, on peut utiliser `mark`, `skip` et `reset` pour se positionner à une position précise dans un fichier.

A.2.11 Fonctions graphiques

Les fonctions sont inspirées de la libraire QuickDraw du Macintosh, mais fonctionnent aussi sur les stations Unix. Sur Macintosh, une fenêtre *Drawing* permet de gérer un écran typiquement de 1024×768 points. L'origine du système de coordonnées est en haut et à gauche. L'axe des x va classiquement de la gauche vers la droite, l'axe des y va plus curieusement du haut vers le bas (c'est une vieille tradition de l'informatique, dure à remettre en cause). En QuickDraw, x et y sont souvent appelés `h` (horizontal) et `v` (vertical). Il y a une notion de point courant et de crayon avec une taille et une couleur courantes. On peut déplacer le crayon, en le levant ou en dessinant des vecteurs par les fonctions suivantes

moveTo (x, y) Déplace le crayon aux coordonnées absolues x, y.

move (dx, dy) Déplace le crayon en relatif de *dx*, *dy*.

lineTo (x, y) Trace une ligne depuis le point courant jusqu'au point de coordonnées *x*, *y*.

line (dx, dy) Trace le vecteur (*dx*, *dy*) depuis le point courant.

penPat(pattern) Change la couleur du crayon : *white*, *black*, *gray*, *dkGray* (*dark gray*), *ltGray* (*light gray*).

penSize(dx, dy) Change la taille du crayon. La taille par défaut est (1, 1). Toutes les opérations de tracé peuvent se faire avec une certaine épaisseur du crayon.

penMode(mode) Change le mode d'écriture : *patCopy* (mode par défaut qui efface ce sur quoi on trace), *patOr* (mode Union, i.e. sans effacer ce sur quoi on trace), *patXor* (mode Xor, i.e. en inversant ce sur quoi on trace).

Certaines opérations sont possibles sur les rectangles. Un rectangle `r` a un type prédéfini `Rect`. Ce type est une classe qui a le format suivant

```
public class Rect {

  short left, top, right, bottom;
}
```

Fort heureusement, il n'y a pas besoin de connaître le format internes des rectangles, et on peut faire simplement les opérations graphiques suivantes sur les rectangles

setRect(r, g, h, d, b) fixe les coordonnées (gauche, haut, droite, bas) du rectangle *r*. C'est équivalent à faire les opérations *r.left := g;*, *r.top := h;*, *r.right := d;*, *r.bottom := b*.

unionRect(r1, r2, r) définit le rectangle *r* comme l'enveloppe englobante des rectangles *r1* et *r2*.

frameRect(r) dessine le cadre du rectangle *r* avec la largeur, la couleur et le mode du crayon courant.

paintRect(r) remplit l'intérieur du rectangle *r* avec la couleur courante.

invertRect(r) inverse la couleur du rectangle *r*.

eraseRect(r) efface le rectangle *r*.

fillRect(r,pat) remplit l'intérieur du rectangle *r* avec la couleur *pat*.

drawChar(c), drawString(s) affiche le caractère *c* ou la chaîne *s* au point courant dans la fenêtre graphique. Ces fonctions diffèrent de *write* ou *writeln* qui écrivent dans la fenêtre texte.

frameOval(r) dessine le cadre de l'ellipse inscrite dans le rectangle *r* avec la largeur, la couleur et le mode du crayon courant.

paintOval(r) remplit l'ellipse inscrite dans le rectangle *r* avec la couleur courante.

invertOval(r) inverse l'ellipse inscrite dans *r*.

eraseOval(r) efface l'ellipse inscrite dans *r*.

fillOval(r,pat) remplit l'intérieur l'ellipse inscrite dans *r* avec la couleur *pat*.

frameArc(r,start,arc) dessine l'arc de l'ellipse inscrite dans le rectangle *r* démarrant à l'angle *start* et sur la longueur définie par l'angle *arc*.

frameArc(r,start,arc) peint le camembert correspondant à l'arc précédent …. Il y a aussi des fonctions pour les rectangles avec des coins arrondis.

button est une fonction qui renvoie la valeur vraie si le bouton de la souris est enfoncé, faux sinon.

getMouse(p) renvoie dans p le point de coordonnées $(p.h, p.v)$ courantes du curseur.

getPixel(p) donne la couleur du point p. Répond un booléen : *false* si blanc, *true.* si noir.

hideCursor(), showCursor() cache ou remontre le curseur.

```
class Point {
  short h, v;

  Point(int h, int v) {
    h = (short)h;
    v = (short)v;
  }
}
class MacLib {

  static void setPt(Point p, int h, int v) {..}
  static void addPt(Point src, Point dst) {...}
  static void subPt(Point src, Point dst) {...}
  static boolean equalPt(Point p1, Point p2) {...}
  ...
}
```

Et les fonctions correspondantes (voir page 218)

```
static void setRect(Rect r, int left, int top, int right, int bottom)
static void unionRect(Rect src1, Rect src2, Rect dst)

static void frameRect(Rect r)
static void paintRect(Rect r)
static void eraseRect(Rect r)
static void invertRect(Rect r)

static void frameOval(Rect r)
static void paintOval(Rect r)
static void eraseOval(Rect r)
static void invertOval(Rect r)

static void frameArc(Rect r, int startAngle, int arcAngle)
static void paintArc(Rect r, int startAngle, int arcAngle)
static void eraseArc(Rect r, int startAngle, int arcAngle)
static void invertArc(Rect r, int startAngle, int arcAngle)
static boolean button()
static void getMouse(Point p)
```

Toutes ces définitions sont aussi sur `poly` dans le fichier

`/usr/local/lib/MacLib-java/MacLib.java`

On veillera à avoir cette classe dans l'ensemble des classes chargeables (variable d'environnement `CLASSPATH`). Le programme suivant est un programme qui fait rebondir une balle dans un rectangle, première étape vers un jeu de *pong*.

```
class Pong extends MacLib {

  static final int C =  5, // Le rayon de la  balle
      X0 = 5, X1 = 250,
```

```
      Y0 =  5, Y1 = 180;

  static void getXY (Point p) {
    int N = 2;
    Rect  r = new Rect();
    int   x, y;

    while (!button())       // On attend le bouton enfoncé
      ;
    while (button())        // On attend le bouton relâché
      ;
    getMouse(p);            // On note les coordonnées du pointeur
    x = p.h;
    y = p.v;
    setRect(r, x - N, y - N, x + N, y + N);
    paintOval(r);           // On affiche le point pour signifier la lecture
  }

  public static void main (String[ ] args) {
    int   x, y, dx, dy;
    Rect  r = new Rect();
    Rect  s = new Rect();
    Point p = new Point();
    int  i;

    initQuickDraw();        //  Initialisation  du graphique
    setRect(s, 50, 50, X1 + 100, Y1 + 100);
    setDrawingRect(s);
    showDrawing();
    setRect(s, X0, Y0, X1, Y1);
    frameRect(s);           // Le rectangle de jeu
    getXY(p);               // On note les coordonnées du pointeur
    x = p.h; y = p.v;
    dx = 1;                 // La vitesse  initiale
    dy = 1;                 // de la  balle
    for (;;) {
      setRect(r, x - C, y - C, x + C, y + C);
      paintOval(r);         // On dessine la  balle en $x,y$
      x = x + dx;
      if (x - C <= X0 + 1 || x + C >= X1 - 1)
        dx = -dx;
      y = y + dy;
      if (y - C <= Y0 + 1 || y + C >= Y1 - 1)
        dy = -dy;
      for (i = 0; i < 2500; ++i)
        ;                   // On temporise
      invertOval(r);        // On efface la  balle
    }
  }
}
```

A.3 Syntaxe BNF de Java

Ce qui suit est une syntaxe sous forme BNF (*Backus Naur Form*). Chaque petit paragraphe est la définition souvent récursive d'un fragment de syntaxe dénommée par un nom (malheureusement en anglais). Chaque ligne correspond à différentes définitions possibles. L'indice *optional* sera mis pour signaler l'aspect facultatif de l'objet indicée. Certains objets (*token*) seront supposée préedéfinis : *IntegerLiteral* pour une constante entière, *Identifier* pour tout identificateur, ...La syntaxe du langage ne garantit pas la concordance des types, certaines phrases pouvant être syntaxiquement correctes, mais fausses pour les types.

Goal:
    CompilationUnit

A.3.1 Contantes littérales

Literal:
    IntegerLiteral
    FloatingPointLiteral
    BooleanLiteral
    CharacterLiteral
    StringLiteral
    NullLiteral

A.3.2 Types, valeurs, et variables

Type:
    PrimitiveType
    ReferenceType

PrimitiveType:
    NumericType
    `boolean`

NumericType:
    IntegralType
    FloatingPointType

IntegralType: one of
    `byte short int long char`

FloatingPointType: one of
    `float double`

ReferenceType:
    ClassOrInterfaceType
    ArrayType

ClassOrInterfaceType:
    Name

ClassType:
    ClassOrInterfaceType

InterfaceType:
    ClassOrInterfaceType

ArrayType:
    PrimitiveType `[ ]`
    Name `[ ]`
    ArrayType `[ ]`

A.3.3 Noms

Name:
  SimpleName
  QualifiedName

SimpleName:
  Identifier

QualifiedName:
  Name . Identifier

A.3.4 Packages

CompilationUnit:
  $\text{PackageDeclaration}_{opt}$ $\text{ImportDeclarations}_{opt}$ $\text{TypeDeclarations}_{opt}$

ImportDeclarations:
  ImportDeclaration
  ImportDeclarations ImportDeclaration

TypeDeclarations:
  TypeDeclaration
  TypeDeclarations TypeDeclaration

PackageDeclaration:
  `package` Name ;

ImportDeclaration:
  SingleTypeImportDeclaration
  TypeImportOnDemandDeclaration

SingleTypeImportDeclaration:
  `import` Name ;

TypeImportOnDemandDeclaration:
  `import` Name . * ;

TypeDeclaration:
  ClassDeclaration
  InterfaceDeclaration
  ;

Modifiers:
  Modifier
  Modifiers Modifier

Modifier: one of
  `public protected private`
  `static`
  `abstract final native synchronized transient volatile`

A.3.5 Classes

Déclaration de classe

ClassDeclaration:

Modifiers$_{opt}$ `class` Identifier Super$_{opt}$ Interfaces$_{opt}$ ClassBody

Super:
    `extends` ClassType

Interfaces:
    `implements` InterfaceTypeList

InterfaceTypeList:
    InterfaceType
    InterfaceTypeList `,` InterfaceType

ClassBody:
    `{` ClassBodyDeclarations$_{opt}$ `}`

ClassBodyDeclarations:
    ClassBodyDeclaration
    ClassBodyDeclarations ClassBodyDeclaration

ClassBodyDeclaration:
    ClassMemberDeclaration
    StaticInitializer
    ConstructorDeclaration

ClassMemberDeclaration:
    FieldDeclaration
    MethodDeclaration

Déclarations de champs

FieldDeclaration:
    Modifiers$_{opt}$ Type VariableDeclarators `;`

VariableDeclarators:
    VariableDeclarator
    VariableDeclarators `,` VariableDeclarator

VariableDeclarator:
    VariableDeclaratorId
    VariableDeclaratorId `=` VariableInitializer

VariableDeclaratorId:
    Identifier
    VariableDeclaratorId `[ ]`

VariableInitializer:
    Expression
    ArrayInitializer

Déclarations de méthodes

MethodDeclaration:
    MethodHeader MethodBody

MethodHeader:
    Modifiers$_{opt}$ Type MethodDeclarator Throws$_{opt}$
    Modifiers$_{opt}$ `void` MethodDeclarator Throws$_{opt}$

MethodDeclarator:
  Identifier `(` FormalParameterList$_{opt}$ `)`
  MethodDeclarator `[ ]`

FormalParameterList:
  FormalParameter
  FormalParameterList `,` FormalParameter

FormalParameter:
  Type VariableDeclaratorId

Throws:
  `throws` ClassTypeList

ClassTypeList:
  ClassType
  ClassTypeList `,` ClassType

MethodBody:
  Block
  `;`

Initialieurs statiques

StaticInitializer:
  `static` Block

Déclarations de constructeurs

ConstructorDeclaration:
  Modifiers$_{opt}$ ConstructorDeclarator Throws$_{opt}$ ConstructorBody

ConstructorDeclarator:
  SimpleName `(` FormalParameterList$_{opt}$ `)`

ConstructorBody:
  { ExplicitConstructorInvocation$_{opt}$ BlockStatements$_{opt}$ }

ExplicitConstructorInvocation:
  `this (` ArgumentList$_{opt}$ `) ;`
  `super (` ArgumentList$_{opt}$ `) ;`

A.3.6 Interfaces

InterfaceDeclaration:
  Modifiers$_{opt}$ `interface` Identifier ExtendsInterfaces$_{opt}$ InterfaceBody

ExtendsInterfaces:
  `extends` InterfaceType
  ExtendsInterfaces `,` InterfaceType

InterfaceBody:
  { InterfaceMemberDeclarations$_{opt}$ }

InterfaceMemberDeclarations:
  InterfaceMemberDeclaration

InterfaceMemberDeclarations InterfaceMemberDeclaration

InterfaceMemberDeclaration :
ConstantDeclaration
AbstractMethodDeclaration

ConstantDeclaration :
FieldDeclaration

AbstractMethodDeclaration :
MethodHeader ;

A.3.7 Tableaux

ArrayInitializer :
{ VariableInitializers$_{opt}$,$_{opt}$ }

VariableInitializers :
VariableInitializer
VariableInitializers , VariableInitializer

A.3.8 Blocs et instructions

Block :
{ BlockStatements$_{opt}$ }

BlockStatements :
BlockStatement
BlockStatements BlockStatement

BlockStatement :
LocalVariableDeclarationStatement
Statement

LocalVariableDeclarationStatement :
LocalVariableDeclaration ;

LocalVariableDeclaration :
Type VariableDeclarators

Statement :
StatementWithoutTrailingSubstatement
LabeledStatement
BlockStatementsBlockStatementsIfThenStatement
IfThenElseStatement
WhileStatement
ForStatement

StatementNoShortIf :
StatementWithoutTrailingSubstatement
LabeledStatementNoShortIf
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf

StatementWithoutTrailingSubstatement :
Block

EmptyStatement
ExpressionStatement
SwitchStatement
DoStatement
BreakStatement
ContinueStatement
ReturnStatement
SynchronizedStatement
ThrowStatement
TryStatement

EmptyStatement:
`;`

LabeledStatement:
Identifier `:` Statement

LabeledStatementNoShortIf:
Identifier `:` StatementNoShortIf

ExpressionStatement:
StatementExpression `;`

StatementExpression:
Assignment
PreIncrementExpression
PreDecrementExpression
PostIncrementExpression
PostDecrementExpression
MethodInvocation
ClassInstanceCreationExpression

IfThenStatement:
`if` `(` Expression `)` Statement

IfThenElseStatement:
`if` `(` Expression `)` StatementNoShortIf `else` Statement

IfThenElseStatementNoShortIf:
`if` `(` Expression `)` StatementNoShortIf `else` StatementNoShortIf

SwitchStatement:
`switch` `(` Expression `)` SwitchBlock

SwitchBlock:
{ SwitchBlockStatementGroups$_{opt}$ SwitchLabels$_{opt}$ }

SwitchBlockStatementGroups:
SwitchBlockStatementGroup
SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchBlockStatementGroup:
SwitchLabels BlockStatements

SwitchLabels:
SwitchLabel
SwitchLabels SwitchLabel

SwitchLabel:
    `case` ConstantExpression `:`
    `default :`

WhileStatement:
    `while (` Expression `)` Statement

WhileStatementNoShortIf:
    `while (` Expression `)` StatementNoShortIf

DoStatement:
    `do` Statement `while (` Expression `) ;`

ForStatement:
    `for (` ForInit$_{opt}$ `;` Expression$_{opt}$ `;` ForUpdate$_{opt}$ `)`
        Statement

ForStatementNoShortIf:
    `for (` ForInit$_{opt}$ `;` Expression$_{opt}$ `;` ForUpdate$_{opt}$ `)`
        StatementNoShortIf

ForInit:
    StatementExpressionList
    LocalVariableDeclaration

ForUpdate:
    StatementExpressionList

StatementExpressionList:
    StatementExpression
    StatementExpressionList `,` StatementExpression

BreakStatement:
    `break` Identifier$_{opt}$ `;`

ContinueStatement:
    `continue` Identifier$_{opt}$ `;`

ReturnStatement:
    `return` Expression$_{opt}$ `;`

ThrowStatement:
    `throw` Expression `;`

SynchronizedStatement:
    `synchronized (` Expression `)` Block

TryStatement:
    `try` Block Catches
    `try` Block Catches$_{opt}$ Finally

Catches:
    CatchClause
    Catches CatchClause

CatchClause:

`catch` (FormalParameter) Block

Finally:
`finally` Block

A.3.9 Expressions

Primary:
PrimaryNoNewArray
ArrayCreationExpression

PrimaryNoNewArray:
Literal
`this`
(Expression)
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
ArrayAccess

ClassInstanceCreationExpression:
`new` ClassType (ArgumentList$_{opt}$)

ArgumentList:
Expression
ArgumentList , Expression

ArrayCreationExpression:
`new` PrimitiveType DimExprs Dims$_{opt}$
`new` ClassOrInterfaceType DimExprs Dims$_{opt}$

DimExprs:
DimExpr
DimExprs DimExpr

DimExpr:
[Expression]

Dims:
[ ]
Dims [ ]

FieldAccess:
Primary . Identifier
`super` . Identifier

MethodInvocation:
Name (ArgumentList$_{opt}$)
Primary . Identifier (ArgumentList$_{opt}$)
`super` . Identifier (ArgumentList$_{opt}$)

ArrayAccess:
Name [Expression]
PrimaryNoNewArray [Expression]

PostfixExpression:
Primary

    Name
    PostIncrementExpression
    PostDecrementExpression

PostIncrementExpression:
    PostfixExpression ++

PostDecrementExpression:
    PostfixExpression --

UnaryExpression:
    PreIncrementExpression
    PreDecrementExpression
    + UnaryExpression
    - UnaryExpression
    UnaryExpressionNotPlusMinus

PreIncrementExpression:
    ++ UnaryExpression

PreDecrementExpression:
    -- UnaryExpression

UnaryExpressionNotPlusMinus:
    PostfixExpression
    ~ UnaryExpression
    ! UnaryExpression
    CastExpression

CastExpression:
    (PrimitiveType Dims$_{opt}$) UnaryExpression
    (Expression) UnaryExpressionNotPlusMinus
    (Name Dims) UnaryExpressionNotPlusMinus

MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression

AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression

ShiftExpression:
    AdditiveExpression
    ShiftExpression << AdditiveExpression
    ShiftExpression >> AdditiveExpression
    ShiftExpression >>> AdditiveExpression

RelationalExpression:
    ShiftExpression
    RelationalExpression < ShiftExpression
    RelationalExpression > ShiftExpression
    RelationalExpression <= ShiftExpression
    RelationalExpression >= ShiftExpression

```
        RelationalExpression instanceof ReferenceType

EqualityExpression:
        RelationalExpression
        EqualityExpression == RelationalExpression
        EqualityExpression != RelationalExpression

AndExpression:
        EqualityExpression
        AndExpression & EqualityExpression

ExclusiveOrExpression:
        AndExpression
        ExclusiveOrExpression ^ AndExpression

InclusiveOrExpression:
        ExclusiveOrExpression
        InclusiveOrExpression | ExclusiveOrExpression

ConditionalAndExpression:
        InclusiveOrExpression
        ConditionalAndExpression && InclusiveOrExpression

ConditionalOrExpression:
        ConditionalAndExpression
        ConditionalOrExpression || ConditionalAndExpression

ConditionalExpression:
        ConditionalOrExpression
        ConditionalOrExpression ? Expression : ConditionalExpression

AssignmentExpression:
        ConditionalExpression
        Assignment

Assignment:
        LeftHandSide AssignmentOperator AssignmentExpression

LeftHandSide:
        Name
        FieldAccess
        ArrayAccess

AssignmentOperator: one of
        = *= /= %= += -= <<= >>= >>>= &= ^= |=

Expression:
        AssignmentExpression

ConstantExpression:
        Expression
```

Bibliographie Java

[j1] *Exploring Java*, 2nd edition, Pat Niemeyer et Joshua Peck 628 pages, O'Reilly, ISBN : 1-56592-271-9. 1997.

[j2] *The Java Language Specification*, James Gosling, Bill Joy et Guy Steele, Addison Wesley, ISBN : 0-201-63456-2. 1996.

[j3] *Java in a Nutshell*, David Flanagan, O'Reilly, ISBN : 1-56592-262-X, 1997.

[j4] *Java examples in a Nutshell*, David Flanagan, O'Reilly, ISBN : 1-56592-371-5. 1997.

Annexe B
Objective Caml

Le langage ML[7] a été conçu par Milner à Edimbourg en 1978 comme un langage typé de manipulation symbolique, servant de métalangage au système de démonstration automatique LCF. Caml est un de ses dialectes conçu et développé à l'INRIA à partir de 1984. Les langages de la famille ML ont d'autres descendants, comme comme Haskell, développé à Glasgow et à Yale, Miranda à Kent, et surtout SML/NJ [9], à Bell laboratories et à CMU. Comme Java, ML est fortement typé, il autorise les définitions algébriques des structures de données et la gestion automatique de la mémoire. Les langages de la famille ML sont devenus populaires dans la communauté du calcul symbolique et dans l'enseignement de l'informatique.

Caml est un langage de programmation *fonctionnelle*, c'est-à-dire un langage qui encourage un style de programmation fondé sur la notion de calcul plutôt que sur la notion de modification de l'état de la mémoire de la machine. Ce style, souvent plus proche des définitions mathématiques, repose sur l'utilisation intensive des fonctions, et n'est possible qu'à cause de l'effort porté sur la compilation des fonctions et la gestion de la mémoire. A ce titre, Caml est assez différent de Pascal et de C, quoiqu'il propose aussi des aspects impératifs qui autorisent un style assez proche du style impératif traditionnel. Il partage avec Java son typage fort et la gestion automaituqe de la mémoire, mais il a des types polymorphes paramétrés et des opérations de filtrage sur les structures de données dynamiques. Ici, comme en Java, nous ne ferons pas de programmation fonctionnelle, et nous nous contenterons d'un usage assez impératif.

On trouve une introduction didactique au langage SML/NJ dans les livres de Paulson[6] et d'Ulmann[8]. Pour une introduction à Caml, on consultera les livres de Weis-Leroy [1], Cousineau-Mauny [3], Hardin-Donzeau-Gouge [4], Monasse [5] ou Chailloux-Manoury-Pagano [10]. Le “Manuel de référence du langage Caml” [2] décrit le langage en détail. Cette annexe a été écrite par Pierre Weis.

B.1 Un exemple simple

Exercice imposé, écrivons l'exemple des carrés magiques en Objective Caml :

```
open Printf;;

let magique a =
 let n = Array.length a in
 let i = ref (n - 1) in
 let j = ref (n / 2) in
 for k = 1 to n * n do
  a.(!i).(!j) <- k;
  if k mod n = 0 then decr i else
   begin
    i := (!i + 1) mod n;
```

```
      j := (!j + 1) mod n;
    end
  done;;

let erreur s = printf "Erreur fatale: %s\n" s; exit 1;;

let lire () =
  printf "Taille du carré magique, svp ? ";
  let n = int_of_string (read_line ()) in
  if n <= 0 || n mod 2 = 0 then erreur "Taille impossible" else n;;

let imprimer a =
  for i = 0 to Array.length a - 1 do
    for j = 0 to Array.length a - 1 do
      printf "%4d " a.(i).(j)
    done;
    printf "\n"
  done;;

let main () =
  let n = lire () in
  let a = Array.make_matrix n n 0 in
  magique a;
  imprimer a;
  exit 0;;

main ();;
```

Phrases

On constate qu'un programme Caml est une suite de phrases qui se terminent toutes par `;;`. Ces phrases sont des définitions de valeurs, de procédures ou de fonctions, ou encore des expressions qui sont évaluées dans l'ordre de présentation. Ainsi, la dernière phrase est l'expression `main ();;` qui déclenche l'exécution du programme. On remarque aussi que les définitions des objets précédent toujours leur première utilisation.

Une *définition* est introduite par le mot clé `let` suivi du nom de l'entité définie. Par exemple, `let n = Array.length a` définit la variable `n` comme la longueur du vecteur `a` et `let magique a = ...` définit la fonction `magique` avec `a` pour argument. À l'occasion, on remarque qu'en Caml on évite les parenthèses inutiles (mais le langage admet les parenthèses superflues) ; ainsi, l'application des fonctions est notée par simple juxtaposition, et l'on écrit `Array.length a` plutôt que `Array.length(a)`.

La valeur des variables en Caml est fixée une fois pour toutes lors de leur définition et cette liaison n'est pas modifiable (il est impossible de changer la valeur liée à un nom défini par `let`). Comme en mathématiques, les variables sont des noms liés à des constantes qu'on calcule lors de la définition de ce nom. C'est aussi analogue aux constantes de Pascal, si ce n'est que l'expression liée à une variable Caml est quelconque et qu'il n'y a pas de limite à sa complexité.

En Caml, la valeur d'une variable est fixée lors de sa définition.

Références

Les variables de Caml ne sont donc pas des variables au sens traditionnel des langages de programmation, puisqu'il est impossible de modifier leur valeur. Il est pourtant souvent nécessaire d'utiliser dans les programmes des variables modifiables au sens de Pascal ou de C. En Caml, on utilise pour cela une *référence* modifiable vers une valeur, c'est-à-dire une case mémoire dont on peut lire et écrire le contenu. Pour créer une référence, on applique le constructeur `ref` au contenu initial de la case mémoire. C'est le cas pour la variable `i`, définie par la ligne `let i = ref (n - 1)`, dont la valeur est une référence qui contient `n - 1` à la création. Pour lire le *contenu* d'une référence, on utilise l'opérateur `!`, qu'on lit "contenu de" (ou "deref" car on l'appelle aussi opérateur de *déréférencement*). Pour écrire le contenu d'une référence on utilise l'opérateur d'affectation `:=`. Par exemple, `i := !i + 1` incrémente le contenu de la référence de la variable `i`, ce qui a finalement le même effet que l'affectation `i := i + 1` de Pascal ou l'affectation `i = i + 1` de C. Noter que les références ne contredisent pas le dogme "une variable est toujours liée à la même valeur" : la variable `i` est liée à une unique référence et il est impossible de la changer. Plus précisément, la valeur de `i` est l'adresse de la case mémoire modifiable qui contient la valeur, et cette adresse est une constante. On ne peut que modifier le contenu de l'adresse.

Le connaisseur de Pascal ou de C est souvent troublé par cette distinction explicite entre une référence et son contenu qui oblige à appliquer systématiquement l'opérateur `!` pour obtenir le contenu d'une référence, alors que ce déréférencement est implicite en Pascal et en C. En Caml, quand `i` a été défini comme une référence, la valeur de `i` est la référence elle-même et jamais son contenu : pour obtenir le contenu, il faut appliquer une opération de déréférencement explicite et l'on écrit `!i`. Sémantiquement, `!i` est à rapprocher de `*i` en C ou `i^` en Pascal.

L'opérateur d'affectation `:=` doit être rapproché aussi des opérateurs ordinaires dont il a le statut, `e1 := e2` signifie que le résultat de l'évaluation de `e1` est une référence dont le contenu va devenir la valeur de `e2` (de même que `e1 + e2` renvoie la somme des valeurs de `e1` et `e2`). Évidemment, dans la grande majorité des cas, la partie gauche de l'affectation est réduite à un identificateur, et l'on affecte simplement la référence qui lui est liée. Ainsi, en écrivant `i := !i - 1`, on décrémente le contenu de la référence `i` en y mettant le prédécesseur de son contenu actuel. Cette opération de décrémentation est d'ailleurs prédéfinie sous la forme d'une procédure qui prend une référence en argument et la décrémente :

```
let decr x =
  x := !x - 1;;
```

Dans cet exemple, la distinction référence-contenu est évidente : l'argument de `decr` est la référence elle-même, pas son contenu. Cette distinction référence-contenu s'éclaire encore si l'on considère les références comme des vecteurs à une seule case : c'est alors un prolongement naturel de la nécessaire distinction entre un vecteur et le contenu de ses cases.

L'opérateur d'affectation en Caml pose une petite difficulté supplémentaire aux habitués des langages impératifs : comme nous venons de le voir, l'écriture `e1 := e2` impose que le résultat de l'évaluation de `e1` soit une référence. Pour des raisons de typage, il n'est donc pas question d'utiliser le symbole `:=` pour affecter des cases de vecteurs, ni des caractères de chaînes, ni même des champs d'enregistrement. Chacune de ces opérations possède son propre opérateur (où intervient le symbole `<-`).

En Caml, l'opérateur `:=` est réservé aux références.

Vecteurs et tableaux

Un vecteur est une succession de cases mémoires. Les indices des éléments commencent en 0, si le vecteur est de longueur n les indices vont de 0 à $n-1$. Pour accéder aux éléments d'un vecteur `v`, on utilise la notation `v.(`*indice*`)`. Pour modifier le contenu des cases de vecteur, on utilise le symbole `<-` qu'on lit reçoit. Par exemple `v.(i) <- k` met la valeur `k` dans la case `i` du vecteur `v`.

Pour créer un tableau, on appelle la primitive `Array.make_matrix`. La ligne

```
let c = Array.make_matrix n n 0 in
```

définit donc une matrice $\mathtt{n} \times \mathtt{n}$, dont les éléments sont des entiers, tous initialisés à 0. Chaque élément de la matrice `c` ainsi définie est accédé par la notation `c.(i).(j)`, et modifié par la notation `c.(i).(j) <-` *nouvelle valeur*. Comme la notation le suggère, `c.(i).(j)` signifie en fait `(c.(i)).(j)`, c'est-à-dire accès au $j^{\text{ième}}$ élément du vecteur `c.(i)`. Cela veut dire que la matrice est en réalité un vecteur dont les éléments sont eux-mêmes des vecteurs : les lignes de la matrice. (Mais rien n'empêche évidemment de définir une matrice comme le vecteur de ses colonnes.) Retenons qu'en Caml comme en C, les tableaux sont des vecteurs de vecteurs. D'autre part, la ligne `let c = make_matrix n n 0 in` définit un tableau dont la taille n'est pas une constante connue à la compilation, mais une valeur déterminée à l'exécution (ici n est lue sur l'entrée standard) ; cependant, une fois le tableau créé, sa taille est fixée une fois pour toutes et n'est plus modifiable.

En Caml, la taille des vecteurs est fixée à la création.

Fonctions et procédures

Caml est un langage fonctionnel : comme nous l'avons déjà vu, les fonctions forment les briques de base des programmes. En outre, les fonctions sont des valeurs primitives du langage qu'on manipule au même titre que les autres valeurs. Il est très facile de définir des fonctions qui manipulent des fonctions ou même de fabriquer des structures de données qui comportent des fonctions. Une fonction peut librement être prise en argument ou rendue en résultat, et il n'y a pas de restriction à son usage dans les structures de données.

En Caml, les fonctions sont des valeurs comme les autres.

Comme en mathématiques, une fonction a des arguments et rend un résultat qu'elle calcule avec une expression où intervient la valeur de ses arguments. Comme pour les autres valeurs, la définition d'une fonction est introduite par un mot clé `let` suivi du nom de la fonction et de la liste de ses arguments, ce qui nous donne typiquement `let f x = ...` pour une fonction à un argument et `let f x1 x2 ... xn = ...` pour une fonction à n arguments.

Voici un exemple de fonction des entiers dans les entiers :

```
let prochain x = if x mod 2 = 1 then 3 * x + 1 else x / 2;;
```

La fonction `prochain` renvoie $3x+1$ si x est impair, et $\lfloor x/2 \rfloor$ si x est pair. (On peut s'amuser à itérer cette fonction et à observer ses résultats successifs ; on ne sait toujours pas démontrer qu'on obtient finalement 1, quelque soit l'entier de départ. Par exemple : 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1).

Définissons maintenant le prédicat `even`, qui teste la parité des entiers (c'est une fonction des entiers dans les booléens) :

```
let even x = x mod 2 = 0;;
```

On remarque que les définitions de `prochain` et de `even` ne font pas intervenir de types : la *signature* des fonctions est implicite. Pourtant Caml dispose, comme Pascal, d'un typage *fort*, c'est-à-dire strict et vérifié complètement à la compilation. En effet, les types des arguments et du résultat des fonctions sont automatiquement calculés par le compilateur, sans intervention du programmeur, ni annotations de type dans les programmes. L'habitué de Pascal ou C pourra s'il le désire insérer des types dans ses programmes, à l'aide de *contraintes de type*. Une contrainte de type consiste en un morceau de programme mis entre parenthèses et décoré avec son type. Ainsi `(x : int)` impose que `x` soit un entier. Avec une contrainte sur son argument et son résultat la fonction `even` s'écrirait :

```
let even (x : int) = (x mod 2 = 0 : bool);;
```

Comme on le constate sur le programme du carré magique, les contraintes de type ne sont pas nécessaires, il n'est donc pas d'usage d'en mettre dans les programmes.

En Caml, le typage est à la charge du compilateur.

Lorsque l'argument ou le résultat d'une fonction sont sans intérêt, on le note alors `()`, l'unique valeur du type `unit`. Une telle fonction est souvent qualifiée de *procédure* au lieu de fonction, mais ce n'est qu'une distinction commode qui n'est pas faite par le langage. Par exemple, `main` est une procédure, et l'on constate qu'elle est définie exactement de la même manière que notre fonction `prochain` ou le prédicat `even`.

La fonction `magique` est elle aussi une procédure, elle construit un carré magique d'ordre n impair de la même façon qu'en Pascal ou C.

La fonction `lire` lit la taille du carré magique et fait quelques vérifications sur sa valeur. En cas de taille incorrecte, elle appelle la fonction `erreur` qui affiche un message et arrête le programme en erreur. Pour lire cette taille, elle appelle la procédure de lecture d'une ligne `read_line`, après avoir imprimé un message sur le terminal. La procédure d'impression formatée `printf` a pour premier paramètre une chaîne de caractères, délimitée par des guillemets. C'est le *format* qui indique comment imprimer les arguments qui suivent : on spécifie le type d'impression désiré, à l'aide de caractères symboliques, précédés de l'indicateur `%`. Par exemple `%s` signifie qu'on doit imprimer une chaîne de caractères, et `%d` un entier. L'ordre des indications d'impression dans le format doit être corrélé avec l'ordre des arguments à imprimer. Dans la procédure `imprimer` qui imprime le tableau `a`, l'indication `%4d` indique l'impression d'un entier sur 4 caractères, cadré à droite.

Enfin, la procédure `main` est la procédure principale du programme qui fait appel successivement aux différentes procédures, dans un ordre simple et compréhensible. La méthode qui consiste à définir de petites fonctions, qu'on appelle ensuite dans la fonction principale est un principe de *programmation structurée*. Elle améliore la lisibilité et facilite les modifications, mais n'est pas une obligation du langage. Nous aurions pu définir toutes les fonctions locales à la fonction `main`, mais en général ce style est mauvais, car on mélange le cœur algorithmique du programme (la procédure `magique`) avec des détails annexes comme l'impression et la lecture. Remarquons qu'il faut appeler explicitement le programme principal, ce que fait la ligne `main ();;` À défaut, la procédure `main` serait définie mais pas appelée, et le programme ne ferait rien.

B.2 Quelques éléments de Caml

Appel au compilateur

Sur la plupart des machines, le système Objective Caml propose un compilateur indépendant `ocamlc` qui produit de façon traditionnelle un programme exécutable à partir d'un programme source, contenu dans un fichier ayant l'extension `.ml`. On précise le nom de l'exécutable produit en donnant l'option `-o nom_de_programme` au compilateur ; à défaut, il produit le programme `a.out`. Voici un exemple de compilation obtenue sous le système Unix.

```
poly% cat fact.ml
let rec fact x = if x <= 1 then 1 else x * fact (x - 1);;

print_int (fact 10); print_newline ();;
poly% ocamlc fact.ml
poly% a.out
3628800
```

En dehors du compilateur proprement dit, les systèmes Caml offrent une interface interactive de dialogue avec le compilateur. Dans cette interface, on entre successivement des phrases qui sont compilées, puis exécutées au vol. Voici une session obtenue sur une machine Unix en lançant le système interactif par la commande `ocaml`.

```
poly% ocaml
       Objective Caml version 3.04

#let rec fact x = if x <= 1 then 1 else x * fact (x - 1);;
val fact : int -> int = <fun>
#fact 10;;
- : int = 3628800
##quit;;
poly%
```

Dans cette utilisation, Caml indique le résultat des calculs et le type des objets définis. Il trouve automatiquement ces types par un algorithme d'*inférence de type*. Au lieu de taper directement les programmes, on peut charger les fichiers qui les contiennent par la directive `#use`. Les phrases du fichier sont alors compilées et exécutées à la volée, exactement comme si on les avait tapées dans le système interactif. Ainsi, le chargement du fichier `fact.ml` exécute les phrases dans l'ordre la fin du programme. On peut alors reprendre l'interaction comme avant le chargement :

```
##use "fact.ml";;
fact : int -> int = <fun>
3628800
- : unit = ()
#
```

Le système interactif est donc plutôt réservé à l'apprentissage du langage. Lorsqu'on maîtrise Caml et qu'on développe de gros programmes, on privilégie le compilateur indépendant qui s'intègre plus facilement aux outils de gestion automatique des logiciels. On réserve alors le système interactif au test rapide des programmes, et dans une moindre mesure à la mise au point.

B.2.1 Fonctions

On utilise la notation $A \to B$ pour dénoter les fonctions de l'ensemble A dans l'ensemble B et par exemple `int -> int` est le type de la fonction `fact` ci-dessus. La valeur d'une fonction est une *valeur fonctionnelle*, notée conventionnellement `<fun>`. Remarquons à nouveau que toute définition lie un identificateur à une valeur ; ainsi `fact` possède une valeur et il s'évalue normalement :

```
#fact;;
- : int -> int = <fun>
```

À l'occasion de la définition de `fact`, on observe aussi que les fonctions récursives doivent être introduites par le mot-clé `let rec`, pour signaler au système qu'on fait référence à leur nom avant leur définition effective.

Les fonctions récursives doivent être introduites par le mot-clé `let rec`.

Pour terminer sur les fonctions, citons l'existence des *fonctions anonymes*, c'est-à-dire des valeurs fonctionnelles qui n'ont pas de nom. On les introduit avec le nouveau mot clé `function` et la construction `function x -> ...` Par exemple :

```
#(function x -> x + 1);;
- : int -> int = <fun>
#(function x -> x + 1) 2;;
- : int = 3
```

Lorsqu'on donne un nom à une fonction anonyme, on définit alors très logiquement une fonction “normale” :

```
#let successeur = (function x -> x + 1);;
val successeur : int -> int = <fun>
#successeur 2;;
- : int = 3
```

On utilise la plupart du temps les fonctions anonymes en argument des *fonctionnelles* que nous décrivons maintenant.

Fonctionnelles

Il n'y a pas de contraintes sur les arguments et résultats des procédures et des fonctions : les arguments et résultats fonctionnels sont donc autorisés. Une bonne illustration consiste à écrire une procédure de recherche d'une racine d'une fonction sur un intervalle donné. La procédure `zéro` prend une fonction `f` en argument (et pour cette raison on dit que `zéro` est une *fonctionnelle*). Définissons d'abord une fonction auxiliaire qui calcule la valeur absolue d'un flottant.

```
let abs x = if x >= 0.0 then x else -. x;;
```

Nous notons à l'occasion que les nombres flottants comportent obligatoirement un point, même 0. Les opérations sur les nombres flottants sont également distinctes de celles des nombres entiers, puisqu'elles sont systématiquement suffixées par un point (sauf les comparaisons). Notre fonctionnelle s'écrit alors :

```
let trouve_zéro f a b epsilon max_iterations =
 let rec trouve x y n =
  if abs (y -. x) < epsilon || n >= max_iterations then x
  else
   let m = (x +. y) /. 2.0 in
   if (f m > 0.0) = (f x > 0.0)
   then trouve m y (n + 1)
```

```
      else trouve x m (n + 1) in
    trouve a b 1;;
  let zéro f a b = trouve_zéro f a b 1.0E-7 100;;
```

Remarquons la définition locale de la fonction récursive `trouve`, qu'on applique ensuite avec les arguments `a`, `b` et `1`. Si on a des difficultés à comprendre ce style *fonctionnel*, voici la même fonction en version impérative (avec une boucle `while` que nous verrons plus loin) :

```
let zéro f a b =
 let epsilon = 1.0E-7
 and nmax = 100 in
 let n = ref 1
 and m = ref ((a +. b) /. 2.0) in
 let x = ref a
 and y = ref b in
 while abs (!y -. !x) > epsilon && !n < nmax do
  m := (!x +. !y) /. 2.0;
  if (f !m > 0.0) = (f !x > 0.0)
   then x := !m
   else y := !m;
  n := !n + 1
 done;
 !x;;
```

Le type inféré par Caml pour `zéro` est `(float -> float) -> float -> float -> float` qui indique bien que le premier argument est une fonction des flottants dans les flottants. Utilisons la fonctionnelle `zéro` pour trouver un zéro de la fonction logarithme entre 1/2 et 3/2 :

```
#open Printf;;
#let log10 x = log x /. log 10.0;;
val log10 : float -> float = <fun>
#printf "le zéro de log10 est %f\n" (zéro log10 0.5 1.5);;
le zéro de log10 est 1.000000
- : unit = ()
```

Les arguments fonctionnels sont naturels et rendent souvent l'écriture plus élégante. Il faut noter que l'efficacité du programme n'en souffre pas forcément, surtout dans les langages fonctionnels qui optimisent la compilation des fonctions et de leurs appels.

B.2.2 Symboles, séparateurs, identificateurs

Les identificateurs sont des séquences de lettres, de chiffres, d'apostrophes et de soulignés commençant par une lettre. Les identificateurs sont séparés par des espaces, des caractères de tabulation, des retours à la ligne ou par des caractères spéciaux comme `+`, `-`, `*`. Certains identificateurs sont réservés pour des *mots clés* de la syntaxe, comme `and`, `type`, `begin`, `end`, `while`, ...

Nous abandonnons la convention adoptée en Pascal et C dans ce cours qui consiste à commencer par une majuscule les noms de constantes et par une minuscule les noms de variables. En Caml, toutes les variables ont une valeur constante (au sens de Pascal et de C) et devraient donc commencer par une majuscule. Nous préférons utiliser les minuscules comme première lettre des variables. (Seuls les noms de constructeurs des types somme et les exceptions commenceront par une majuscule.)

B.2.3 Types de base

L'unique valeur *rien* a le type prédéfini `unit` ; elle est notée `()` et est lue "voïde". La valeur rien sert à compléter une conditionnelle à une seule branche (par `else ()`), à déclencher les procédures (`print_newline ()`), et comme instruction vide dans le corps d'une boucle.

Les *booléens* ont le type prédéfini `bool`, qui contient deux constantes `true` et `false`.

Les *entiers* ont le type prédéfini `int`. Les constantes entières sont une suite de chiffres décimaux, éventuellement précédée d'un signe `-`, comme 234, -128, ...Les valeurs extrémales dépendent de l'implémentation.

Les *flottants* ont le type prédéfini `float`. Les constantes flottantes sont une suite de chiffres décimaux comprenant un point, éventuellement suivie d'une indication d'exposant, comme `3.1416` ou `3141.6E-3` pour désigner $3141,6 \times 10^{-3}$.

Les *caractères* ont le type prédéfini `char`. Une constante caractère est une lettre entourée du symbole `'`, comme `'a'`, `'b'`, ..., `'+'`, `':'`. Certains caractères sont codés spécialement comme `'\n'` pour le changement de ligne, `'\r'` pour le retour charriot, `'\t'` pour la tabulation, `'\''` pour le caractère apostrophe, et `'\\'` pour la barre oblique. Enfin, on dénote n'importe quel caractère en le désignant par son numéro décimal dans le code ASCII (*American Standard Codes for Information Interchange*) (ISO-latin), sur trois chiffres et précédé d'une barre oblique. Ainsi `'\032'` désigne le caractère espace et `'\233'` est équivalent à `'é'`. La fonction `int_of_char` donne la valeur entre 0 et 255 dans le code ASCII du caractère. Inversement, la fonction `char_of_int` donne le caractère par son code ASCII. En Objective Caml, on peut aussi utiliser les fonctions `Char.code` et `Char.chr`.

Les *chaînes de caractères* ont le type prédéfini `string`. Ce sont des suites de caractères rangés consécutivement en mémoire. Une constante chaîne est une suite de caractères entourée de guillemets. Dans une chaîne, le caractère guillemet se note `\"` en ajoutant une barre oblique devant le guillemet, et les caractères spéciaux obéissent aux mêmes conventions que pour les constantes caractères. Les éléments d'une chaîne de caractères sont numérotés à partir de 0. On accède à l'élément i de la chaîne s à l'aide de la notation s`.[`i`]`. On remplace l'élément i de la chaîne s par la valeur c, à l'aide de la notation s`.[`i`] <-` c. En évaluant `String.make` l c, on obtient une chaîne de caractères de longueur l, initialisée avec des caractères c. L'opérateur infixe `^` sert à concaténer deux chaînes, la fonction `String.sub` permet d'extraire une sous-chaîne, et la procédure `String.blit` de transférer des caractères d'une chaîne à une autre. Pour plus d'information, voir le module `string` de la librairie.

Les *vecteurs* ont le type prédéfini `array`. Ce sont des suites d'éléments *de même type*, rangés consécutivement en mémoire. Une constante vecteur est une suite d'éléments séparés par des `;` et entourée de "parenthèses" `[|` et `|]`. Par exemple :

```
#let v = [| 1; 2; 3 |];;
val v : int array = [|1; 2; 3|]
```

Remarquons la notation suffixe pour le constructeur de type des vecteurs. Le type d'un vecteur d'entiers s'écrit `int vect`, et le type d'une matrice d'entiers `int vect vect`. Les éléments d'un vecteur sont numérotés à partir de 0. On accède à l'élément i du vecteur v à l'aide de la notation v`.(`i`)`. On remplace l'élément i du vecteur v par la valeur c, à l'aide de la notation v`.(`i`) <-` c. En évaluant `Array.make` l c, on obtient un vecteur de longueur l, initialisé avec l'élément c. On dispose aussi des fonctions `Array.sub` pour extraire des sous-chaînes et `Array.blit` pour transférer des éléments d'un vecteur à un autre. Pour plus d'information, voir le module `vect` de la librairie.

Les *références* ont le type prédéfini `ref`. Comme les vecteurs le constructeur de type des références `ref` utilise la notation suffixe. Une référence est construite par l'application du constructeur (de valeur) `ref` à sa valeur initiale. En évaluant `ref` v, on obtient une référence, initialisée avec la valeur v. On accède au contenu d'une référence r à l'aide de la notation `!`r. On remplace le contenu de la référence r par la valeur c, à l'aide de la notation r `:=` c.

Les *paires* d'éléments de type `t1` et `t2` ont le type `t1 * t2`. On écrit la paire des valeurs v_1 et v_2 de la manière mathématique classique : `(`v_1`,` v_2`)`. La notation s'étend aux *n-uplets*. Il n'y a pas de limitation à l'usage des n-uplets, qui peuvent être librement pris en argument et rendus en résultat des fonctions. Par exemple, la symétrie par rapport à l'origine du repère s'écrit :

```
#let symétrie (x, y) = (-x, -y);;
val symétrie : int * int -> int * int = <fun>
```

Attention, les n-uplets ne sont pas associatifs et $(1,2,3) \neq ((1,2),3)$.

B.2.4 Expressions

Les expressions arithmétiques font intervenir les opérateurs classiques sur les entiers `+` (addition), `-` (soustraction), `*` (multiplication), `/` (division entière), `mod` (modulo). On utilise les parenthèses comme en mathématiques. Ainsi, si `x` et `y` sont deux entiers, on écrit `3 * (x + 2 * y) + 2 * x * x` pour $3(x+2y)+2x^2$.

Les mêmes opérateurs, suffixés par un point, servent pour les expressions flottantes. Donc, si `z` est un flottant, on écrit `3.0 *. (z +. 1) /. 2.0` pour $3(z+1)/2$. Les fonctions `int_of_float` et `float_of_int` autorisent les conversions des flottants dans les entiers : la première donne la partie entière, la seconde convertit un entier en flottant. Contrairement à Pascal ou à C, les conversions ne sont jamais automatiques : par exemple `3.5 + 2` est toujours mal typé.

En Caml, les conversions sont explicites.

Une *expression conditionnelle* ou *alternative* s'écrit :

`if` e `then` e_1 `else` e_2

où la condition e est une expression booléenne du type `bool`, et e_1, e_2 sont deux expressions de même type qui est celui du résultat.

Les *expressions booléennes* sont construites à partir des opérateurs `||`, `&&`, `not`, des booléens et des opérateurs de comparaison. Ainsi, si `b` et `c` sont deux identificateurs de type bool, l'expression

```
(b && not c) || (not b && c)
```

représente le ou-exclusif de `b` et `c`. Les deux opérateurs `||` et `&&` se comportent exactement comme une construction "if then else". Par définition, `a && b` signifie `if a then b else false` et `a || b` signifie `if a then true else b`. Parfois ces opérateurs rendent un résultat sans évaluer certains de leurs arguments. Si a s'évalue en faux, alors `a && b` rend `false` sans que l'expression `b` soit évaluée. Les opérateurs de comparaison `=`, `<>`, `<=`, `<`, `>`, `>=` rendent aussi des valeurs booléennes. On peut comparer des entiers, des flottants, des booléens, des caractères (dans ce dernier cas, l'ordre est celui du code ASCII) et même deux valeurs quelconques, pourvu qu'elles soient du même type.

La précédence des opérateurs est naturelle. Ainsi `*` est plus prioritaire que `+`, lui-même plus prioritaire que `=`. Si un doute existe, il ne faut pas hésiter à mettre des parenthèses. De manière générale, seules les parenthèses vraiment significatives sont nécessaires. Par exemple, dans

```
if (x > 1) && (y = 3) then ...
```

la signification ne change pas si l'on ôte toutes les parenthèses. De même, dans l'expression du ou-exclusif

```
(b && not c) || (not b && c)
```

les parenthèses sont superflues. En effet les précédences respectives de `&&`, `||` et `not` sont analogues à celles de `*`, `+` et `-`. On écrit donc plus simplement `b && not c || not b && c`. (Encore plus simple : `b <> c` !) Évidemment, certaines parenthèses sont impératives pour grouper les expressions. L'exemple des arguments de fonctions est plus particulièrement fréquent : comme en mathématiques `f (x + 1)` est essentiellement différent de `f (x) + 1`. Et comme on omet souvent les parenthèses autour des arguments très simples (variables ou constantes), il faut aussi noter que `f (x) + 1` est synonyme de `f x + 1`. De toutes façons, les parenthèses sont indispensables pour les arguments de fonctions compliqués. Pour la même raison les parnthèses sont nécessaires autour des arguments négatifs `f (-1)` $\neq$ `f -1`, car `f -1` est synonyme de `f - 1` qui est une soustraction.

`f (x + 1)` $\neq$ `f x + 1`.

L'ordre d'évaluation des opérateurs dans les expressions respecte les conventions mathématiques lorsqu'elles existent (priorité de l'addition par rapport à la multiplication par exemple). En ce qui concerne l'application des fonctions, on évalue les arguments avant de rentrer dans le corps de la fonction (appel par valeur). Comme en C, il n'y a pas d'appel par référence mais on peut pratiquement le simuler en utilisant des références (c'est le cas pour la fonction `decr` décrite page 235). L'ordre d'évaluation des arguments des opérateurs et des fonctions n'est pas spécifié par le langage. C'est pourquoi il faut impérativement éviter de faire des effets dans les arguments de fonctions. En règle générale, il ne faut pas mélanger les effets (impressions, lectures ou modification de la mémoire, déclenchement d'exceptions) avec l'évaluation au sens mathématique.

En Caml, l'ordre d'évaluation des arguments n'est pas spécifié.

L'opérateur d'égalité s'écrit avec le symbole usuel =. C'est un opérateur *polymorphe*, c'est-à-dire qu'il s'applique sans distinction à tous les types de données. En outre, c'est une égalité *structurelle*, c'est-à-dire qu'elle parcourt complètement ses arguments pour détecter une différence ou prouver leur égalité. L'habitué de C peut être surpris, si par mégarde il utilise le symbole `==` au lieu de `=`, car il existe aussi un opérateur `==` en Caml (et son contraire `!=`). Cet opérateur teste l'*égalité physique* des valeurs (identité des adresses mémoire en cas de valeurs allouées). Deux objets physiquement égaux sont bien sûr égaux. La réciproque n'est pas vraie :

```
#"ok" = "ok";;
- : bool = true
#"ok" == "ok";;
- : bool = false
```

L'égalité physique est indispensable pour comparer directement les références, plutôt que leur contenu (ce que fait l'égalité structurelle). On s'en sert par exemple dans les algorithmes sur les graphes.

```
#let x = ref 1;;
val x : int ref = ref 1
#let y = ref 1;;
val y : int ref = ref 1
#x = y;;
- : bool = true
#x == y;;
- : bool = false
#x == x;;
- : bool = true
#x := 2;;
- : unit = ()
#x = y;;
- : bool = false
```

B.2.5 Blocs et portée des variables

Dans le corps des fonctions, on définit souvent des valeurs *locales* pour calculer le résultat de la fonction. Ces définitions sont introduites par une construction `let` *ident* `=` *expression* `in ...`. Il n'y a pas de restriction sur les valeurs locales, et les définitions de fonctions sont admises. Ces fonctions locales sont elles-mêmes susceptibles de comprendre de nouvelles définitions de fonctions si nécessaire et ce *ad libitum*.
Lorsqu'on cite un identificateur `x` dans un programme, il fait nécessairement référence au dernier identificateur de nom `x` lié par une définition `let`, ou introduit comme paramètre d'une fonction après le mot-clé `function`. En général, il est plus élégant de garder les variables aussi locales que possible et de minimiser le nombre de variables globales. Ce mode de liaison des identificateurs (qu'on appelle la portée *statique*) est surprenant dans le système interactif. En effet, on ne peut jamais modifier la définition d'un identificateur ; en particulier, la correction d'une fonction incorrecte n'a aucun effet sur les utilisations *antérieures* de cette fonction dans les fonctions déjà définies.

```
let successeur x = x - 1;;
let plus_deux x = successeur (successeur x);;
#plus_deux 1;;
- : int = -1
```

Ici, on constate la bévue dans la définition de `successeur`, on corrige la fonction, mais cela n'a aucun effet sur la fonction `plus_deux`.

```
#let successeur x = x + 1;;
val successeur : int -> int = <fun>
#plus_deux 1;;
- : int = -1
```

La solution à ce problème est de recharger complètement les fichiers qui définissent le programme. En cas de doute, quitter le système interactif et recommencer la session.

B.2.6 Correction des programmes

Le suivi de l'exécution des fonctions est obtenu à l'aide du mode *trace* qui permet d'afficher les arguments d'une fonction à l'entrée dans la fonction et le résultat à la sortie. Dans l'exemple du paragraphe précédent, le mécanisme de trace nous renseigne utilement : en traçant la fonction `successeur` on constate qu'elle n'est jamais appelée pendant l'évaluation de `plus_deux 1` (puisque c'est l'ancienne version de `successeur` qui est appelée).

```
#trace successeur;;
successeur is now traced.
- : unit = ()
#successeur 1;;
successeur <-- 1
successeur --> 2
- : int = 2
#plus_deux 1;;
- : int = -1
```

Le mode trace est utile pour suivre le déroulement des calculs, mais moins intéressant pour pister l'évolution de la mémoire. En ce cas, on imprime des messages pendant le déroulement du programme.

B.2.7 Instructions

Caml n'a pas à proprement parler la notion d'instructions, puisqu'en dehors des définitions, toutes les constructions syntaxiques sont des expressions qui donnent lieu à une évaluation et produisent un résultat. Parmi ces expressions, la *séquence* joue un rôle particulier : elle permet d'évaluer successivement les expressions. Une séquence est formée de deux expressions séparées par un `;`, par exemple e_1 `;` e_2. La valeur d'une séquence est celle de son dernier élément, les résultats intermédiaires sont ignorés. Dans e_1 `;` e_2, on évalue e_1, on oublie le résultat obtenu, puis on évalue e_2 qui donne le résultat final de la séquence. Comme en Pascal, on peut entourer une séquence des mots-clé `begin` et `end`.

`begin` e_1`;` e_2`; ...;` e_n `end`

Dans une séquence, on admet les alternatives sans partie `else` :

`if` e `then` e_1

qui permet d'évaluer e_1 seulement quand e est vraie, alors que l'alternative complète

`if` e `then` e_1 `else` e_2

évalue e_2 quand e est faux. En réalité, la conditionnelle partielle est automatiquement complétée en une alternative complète avec `else ()`. Cela explique pourquoi le système rapporte des erreurs concernant le type `unit`, en cas de conditionnelle partielle dont l'unique branche n'est pas de type `unit` :

```
#if true then 1;;
              ^
This expression has type int but is here used with type unit
#if true then printf "Hello world!\n";;
Hello world!
- : unit = ()
```

On lève les ambiguités dans les cascades de conditionnelles en utilisant `begin` et `end`. Ainsi

`if` e `then if` e'`then` e_1 `else` e_2

équivaut à

`if` e `then begin if` e'`then` e_1`else` e_2 `end`

Filtrage

Caml fournit d'autres méthodes pour aiguiller les calculs : `match` permet d'éviter une cascade d'expressions `if` et opère un aiguillage selon les différentes valeurs possibles d'une expression. Ce mécanisme s'appelle le *filtrage* ; il est plus riche qu'une simple comparaison avec l'égalité, car il fait intervenir la forme de l'expression et opère des liaisons de variables. À la fin d'un filtrage, un cas _ se comporte comme un cas par défaut. Ainsi

```
match e with
| v1 -> e1
| v2 -> e2
  ...
| vn -> en
| _ -> défaut
```

permet de calculer l'expression e_i si $e = v_i$, ou *défaut* si $e \neq v_i$ pour tout i. Nous reviendrons sur ce mécanisme plus tard.

Boucles

Les autres constructions impératives servent à l'itération, ce sont les boucles `for` et `while`. Dans les deux cas, le corps de la boucle est parenthésé par les mot clés `do` et `done`. La boucle `for` permet d'itérer, sans risque de non terminaison, avec un *indice de boucle*, un identificateur dont la valeur augmente automatiquement de 1 ou de -1 à chaque itération. Ainsi

```
for i = e1 to e2 do e done
```

évalue l'expression e avec `i` valant successivement e_1, $e_1 + 1$, ... jusqu'à e_2 compris. Si e_1 est supérieur à e_2, le corps de la boucle n'est jamais évalué. De même

```
for i = e1 downto e2 do e done
```

itère de e_1 à e_2 en décroissant. Dans les deux cas, l'indice de boucle est introduit par la boucle et disparaît à la fin de celle-ci. En outre, cet indice de boucle n'est pas lié à une référence mais à un entier : sa valeur ne peut être modifiée par une affectation dans le corps de la boucle. Les seuls pas d'itération possibles sont 1 et -1. Pour obtenir d'autres pas, il faut multiplier la valeur de la variable de contrôle ou employer une boucle `while`.

On ne peut pas affecter l'indice de boucle d'une boucle `for`.

La construction `while`,

```
while e1 do e done
```

évalue répétitivement l'expression e tant que la condition booléenne e_1 est vraie. (Si e_1 est toujours fausse, le corps de la boucle n'est jamais exécuté.) Lorsque le corps d'une boucle `while` est vide, on la remplace par la valeur rien, `()`, qui joue alors le rôle d'une instruction vide. Par exemple,

```
while not button_down () do () done;
```

attend que le bouton de la souris soit enfoncé.

B.2.8 Exceptions

Il existe un dispositif de gestion des cas exceptionnels. En appliquant la primitive `raise` à une valeur *exceptionnelle*, on déclenche une erreur. De façon symétrique, la construction syntaxique `try` *calcul* `with` *filtrage* permet de récupérer les erreurs qui se produiraient lors de l'évaluation de *calcul*. La valeur renvoyée s'il n'y a pas d'erreur doit être du même type que celle renvoyée en cas d'erreur dans la partie `with` de la construction. Ainsi, le traitement des situations exceptionnelles (dans la partie `with`) est disjoint du déroulement normal du calcul. En cas de besoin, le programmeur définit ses propres exceptions à l'aide de la construction `exception` *nom-de-l'exception*`;;` pour les exceptions sans argument ; ou `exception` *nom-de-l'exception* `of` *type-de-l'argument*`;;` pour les exceptions avec argument. L'argument des exceptions permet de véhiculer une valeur, de l'endroit où se produit l'erreur à celui où elle est traitée (voir plus loin).

B.2.9 Entrées – Sorties

On lit sur le terminal (ou la fenêtre texte) à l'aide de la fonction prédéfinie `read_line` qui renvoie la chaîne de caractères tapée.

Pour les impressions simples, on dispose de primitives pour les types de base : `print_int`, `print_char`, `print_float` et `print_string`; la procédure `print_newline` permet de passer à la ligne. Pour des impressions plus sophistiquées, on emploie la fonction d'impression formatée `printf` (de la bibliothèque `printf`).

La lecture et l'écriture sur fichiers a lieu par l'intermédiaire de *canaux d'entrées-sorties*. Un canal est ouvert par l'une des primitives `open_in` ou `open_out`. L'appel `open_in` *nom_de_fichier* crée un *canal d'entrée* sur le fichier *nom_de_fichier*, ouvert en lecture. L'appel `open_out` *nom_de_fichier* crée un *canal de sortie* sur le fichier *nom_de_fichier*, ouvert en écriture. La lecture s'opère principalement par les primitives `input_char` pour lire un caractère, ou `input`, `input_line` pour les chaînes de caractères. En sortie, on utilise `output_char`, `output_string` et `output`. Il ne faut pas oublier de fermer les canaux ouverts lorsqu'ils ne sont plus utilisés (à l'aide de `close_in` ou `close_out`). En particulier, pour les fichier ouverts en écriture, la fermeture du canal assure l'écriture effective sur le fichier (sinon les écritures sont réalisées en mémoire, dans des tampons).

Copie de fichiers

Le traitement des fichiers nous permet d'illustrer le mécanisme d'exception. Par exemple, l'ouverture d'un fichier inexistant se solde par le déclenchement d'une erreur par le système d'exploitation : l'exception `Sys_error` est lancée avec pour argument la chaîne de caractères "*fichier*`: No such file or directory`".

```
#open_in "essai";;
Uncaught exception: Sys_error "essai: No such file or directory".
```

On remarque qu'une exception qui n'est pas rattrapée interrompt complètement les calculs.

Supposons maintenant que le fichier de nom `essai` existe. Après avoir ouvert un canal sur ce fichier et l'avoir lu entièrement, toute tentative de lecture provoque aussi le déclenchement d'une exception, l'exception prédéfinie `End_of_file` :

```
#let ic = open_in "essai" in
  while true do input_line ic done;;
Uncaught exception: End_of_file.
```

À l'aide d'une construction `try`, on récupèrerait facilement l'erreur pour l'imprimer (et éventuellement continuer autrement le programme). Nous illustrons ce mécanisme en écrivant une procédure qui copie un fichier dans un autre. On utilise une boucle infinie qui copie ligne à ligne le canal d'entrée et ne s'arrête que lorsqu'on atteint la fin du fichier à copier, qu'on détecte par le déclenchement de l'exception prédéfinie `End_of_file`. Ici le déclenchement de l'exception n'est pas une erreur, c'est au contraire l'indication attendue de la fin du traitement.

```
open Printf;;

let copie_channels ic oc =
 try
  while true do
   let line = input_line ic in
   output_string oc line;
   output_char oc '\n'
  done
 with
 | End_of_file -> close_in ic; close_out oc;;
```

La procédure de copie elle-même se contente de créer les canaux sur les fichiers d'entrée et de sortie, puis d'appeler la procédure `copie_channels`. Comme les ouvertures des fichiers d'entrée et de sortie sont susceptibles d'échouer, la procédure utilise deux `try` imbriqués pour assurer la bonne gestion des erreurs. En particulier, le canal d'entrée n'est pas laissé ouvert quand il y a impossibilité d'ouvrir le canal de sortie. Dans le `try` intérieur qui protège l'ouverture du fichier de sortie et la copie, on remarque le traitement de deux exceptions. La deuxième, `Sys.Break`, est déclenchée quand on interrompt le programme. En ce cas un message est émis, les canaux sont fermés et l'on déclenche à nouveau l'interruption pour prévenir la fonction appelante que la copie ne s'est pas déroulé normalement.

```
let copie origine copie =
 try
  let ic = open_in origine in
   try
    let oc = open_out copie in
    copie_channels ic oc
   with
   | Sys_error s ->
      close_in ic;
      printf "Impossible d'ouvrir le ficher %s \n" copie;
      raise (Sys_error s)
   | Sys.Break ->
      close_in ic;
      close_out oc;
      printf "Interruption pendant la copie de %s dans %s\n" origine copie;
      raise (Sys.Break)
 with
 | Sys_error s ->
    printf "Le ficher %s n'existe pas\n" origine;
    raise (Sys_error s);;
```

B.2.10 Définitions de types

Types sommes : types énumérés

L'utilisateur peut définir ses propres types de données. Par exemple, les types énumérés `couleur` et `sens` définissent un ensemble de constantes qui désignent des objets symboliques.

```
type couleur = Bleu | Blanc | Rouge
and sens = Gauche | Haut | Droite | Bas;;
```

```
let c = Bleu
and s = Droite in
...
end;
```

couleur est l'énumération des trois valeurs Bleu, Blanc, Rouge. On aura remarqué que le symbole | signifie "ou". Le type bool est aussi un type énuméré prédéfini tel que :

```
type bool = false | true;;
```

Par exception à la règle et pour la commodité du programmeur, les constructeurs du type bool ne commencent pas par une majuscule.

Types sommes : unions

Les types sommes sont une généralisation des types énumérés : au lieu de définir des constantes, on définit des constructeurs qui prennent des arguments pour définir une valeur du nouveau type. Considérons par exemple un type d'expressions contenant des constantes (définies par leur valeur entière), des variables (définies par leur nom), des additions et des multiplications (définies par le couple de leurs deux opérandes), et des exponentiations définies par le couple d'une expression et de son exposant. On définira le type expression par :

```
type expression =
 | Const of int
 | Var of string
 | Add of expression * expression
 | Mul of expression * expression
 | Exp of expression * int;;
```

On crée des valeurs de type somme en appliquant leurs constructeurs à des arguments du bon type. Par exemple le polynôme $1 + 2x^3$ est représenté par l'expression :

```
let p = Add (Const 1, Mul (Const 2, Exp (Var "x", 3)));;
```

Les types somme permettent de faire du *filtrage* (*pattern matching*), afin de distinguer les cas en fonction d'une valeur filtrée. Ainsi la dérivation par rapport à une variable x se définirait simplement par :

```
let rec dérive x e =
 match e with
  | Const _ -> Const 0
  | Var s -> if x = s then Const 1 else Const 0
  | Add (e1, e2) -> Add (dérive x e1, dérive x e2)
  | Mul (Const i, e2) -> Mul (Const i, dérive x e2)
  | Mul (e1, e2) -> Add (Mul (dérive x e1, e2), Mul (e1, dérive x e2))
  | Exp (e, i) -> Mul (Const i, Mul (dérive x e, Exp (e, i - 1)));;
```

Nous ne donnerons ici que la signification intuitive du filtrage sur notre exemple particulier. La construction match du corps de dérive signifie qu'on examine la valeur de l'argument e et selon que c'est :

- Const _ : une constante quelconque (_), alors on retourne la valeur 0.
- Var s : une variable que nous nommons s, alors on retourne 1 ou 0 selon que c'est la variable par rapport à laquelle on dérive.
- Add (e1, e2) : une somme de deux expressions que nous nommons respectivement e1 et e2, alors on retourne la somme des dérivée des deux expressions e1 et e2.
- les autres cas sont analogues au cas de la somme.

On constate sur cet exemple la puissance et l'élégance du mécanisme. Combiné à la récursivité, il permet d'obtenir une définition de `dérive` qui se rapproche des définitions mathématiques usuelles.On obtient la dérivée du polynôme $\mathtt{p} = 1 + 2x^3$ en évaluant :

```
#dérive "x" p;;
- : expression =
 Add
  (Const 0,
   Mul (Const 2, Mul (Const 3, Mul (Const 1, Exp (Var "x", 2)))))
```

On constate que le résultat obtenu est grossièrement simplifiable. On écrit alors un simplificateur (naïf) par filtrage sur les expressions :

```
let rec puissance i j =
 match j with
 | 0 -> 1
 | 1 -> i
 | n -> i * puissance i (j - 1);;

let rec simplifie e =
 match e with
 | Add (Const 0, e) -> simplifie e
 | Add (Const i, Const j) -> Const (i + j)
 | Add (e, Const i) -> simplifie (Add (Const i, e))
 | Add (e1, e2) -> Add (simplifie e1, simplifie e2)
 | Mul (Const 0, e) -> Const 0
 | Mul (Const 1, e) -> simplifie e
 | Mul (Const i, Const j) -> Const (i * j)
 | Mul (e, Const i) -> simplifie (Mul (Const i, e))
 | Mul (e1, e2) -> Mul (simplifie e1, simplifie e2)
 | Exp (Const 0, j) -> Const 0
 | Exp (Const 1, j) -> Const 1
 | Exp (Const i, j) -> Const (puissance i j)
 | Exp (e, 0) -> Const 1
 | Exp (e, 1) -> simplifie e
 | Exp (e, i) -> Exp (simplifie e, i)
 | e -> e;;
```

Pour comprendre le filtrage de la fonction `simplifie`, il faut garder à l'esprit que l'ordre des clauses est significatif puisqu'elles sont essayées dans l'ordre. Un exercice intéressant consiste aussi à prouver formellement que la fonction `simplifie` termine toujours. On obtient maintenant la dérivée du polynôme $\mathtt{p} = 1 + 2x^3$ en évaluant :

```
#simplifie (dérive "x" p);;
- : expression = Mul (Const 2, Mul (Const 3, Exp (Var "x", 2)))
```

Types produits : enregistrements

Les enregistrements (*records* en anglais) permettent de regrouper des informations hétérogènes. Ainsi, on déclare un type `date` comme suit :

```
type mois =
 | Janvier | Février | Mars | Avril | Mai | Juin | Juillet
 | Aout | Septembre | Octobre | Novembre | Décembre;;

type date = {j: int; m: mois; a: int};;

let berlin = {j = 10; m = Novembre; a = 1989}
and bastille = {j = 14; m = Juillet; a = 1789};;
```

Un enregistrement contient des champs de type quelconque, donc éventuellement d'autres enregistrements. Supposons qu'une personne soit représentée par son nom, et sa date de naissance ; le type correspondant comprendra un champ contenant un enregistrement de type `date` :

```
type personne = {nom: string; naissance: date};;

let poincaré =
 {nom = "Poincaré"; naissance = {j = 29; m = Avril; a = 1854}};;
```

Les champs d'un enregistrement sont éventuellement *mutables*, c'est-à-dire modifiables par affectation. Cette propriété est spécifique à chaque champ et se déclare lors de la définition du type, en ajoutant le mot clé `mutable` devant le nom du champ. Pour modifier le champ `label` du record `r` en y déposant la valeur `v`, on écrit `r.label <- v`.

```
#type point = {mutable x : int; mutable y : int};;
Le type t est défini.
#let origine = {x = 0; y = 0};;
origine : point = {x = 0; y = 0}
#origine.x <- 1;;
- : unit = ()
#origine;;
- : point = {x = 1; y = 0}
```

En combinaison avec les types somme, les enregistrements modélisent des types de données complexes :

```
type complexe =
   | Cartésien of cartésiennes
   | Polaire of polaires

and cartésiennes = {re: float; im: float}
and polaires = {rho: float; theta: float};;

let pi = 4.0 *. atan 1.0;;

let x = Cartésien {re = 0.0; im = 1.0}
and y = Polaire {rho = 1.0; theta = pi /. 2.0};;
```

Une rotation de $\pi/2$ s'écrit alors :

```
let rotation_pi_sur_deux = function
  | Cartésien x -> Cartésien {re = -. x.im; im = x.re}
  | Polaire x -> Polaire {rho = x.rho; theta = x.theta +. pi /. 2.0};;
```

Types abréviations

On donne un nom à une expression de type à l'aide d'une définition d'abréviation. C'est quelquefois utile pour la lisibilité du programme. Par exemple :

```
type compteur = int;;
```

définit un type `compteur` équivalent au type `int`.

Types abstraits

Si, dans l'interface d'un module (voir ci-dessous), on exporte un type sans rien en préciser (sans donner la liste de ses champs s'il s'agit d'un type enregistrement, ni la liste de ses constructeurs s'il s'agit d'un type somme), on dit qu'on a *abstrait* ce type, ou qu'on l'a exporté abstraitement. Pour exporter abstraitement le type `t`, on écrit simplement

```
type t;;
```

L'utilisateur du module qui définit ce type abstrait n'a aucun moyen de savoir comment le type `t` est implémenté s'il n'a pas accès au source de l'implémentation du module. Cela permet de changer cette implémentation (par exemple pour l'optimiser) sans que l'utilisateur du module n'ait à modifier ses propres programmes. C'est le cas du type des piles dans l'interface du module `stack` décrit ci-dessous.

Égalité de types

La concordance des types se fait par nom. Les définitions de type sont qualifiées de *génératives*, c'est-à-dire qu'elles introduisent toujours de nouveaux types (à la manière des `let` emboîtés qui introduisent toujours de nouveaux identificateurs). Ainsi, deux types sont égaux s'ils font référence à la même définition de type.

Attention : ce phénomène implique que deux types *de même nom* coexistent parfois dans un programme. Dans le système interactif, cela arrive quand on redéfinit un type qui était erroné. Le compilateur ne confond pas les deux types, mais il énonce éventuellement des erreurs de type bizarres, car il n'a pas de moyen de nommer différemment les deux types. Étrangement, il indique alors qu'un type `t` (l'ancien) n'est pas compatible avec un type `t` (mais c'est le nouveau). Considérons les définitions

```
#type t = C of int;;
type t = C of int
#let int_of_t x =
  match x with C i -> i;;
val int_of_t : t -> int = <fun>
```

Jusque là rien d'anormal. Mais définissons `t` à nouveau (pour lui ajouter un nouveau constructeur par exemple) : l'argument de la fonction `int_of_t` est de l'*ancien* type `t` et on ne peut pas l'appliquer à une valeur du nouveau type `t`. (Voir aussi l'URL `http ://pauillac.inria.fr/caml/FAQ/FAQ_EXPERT-fra.html`.)

```
#type t = C of int | D of float;;
type t = C of int | D of float
#int_of_t (C 2);;
          ^^^
This expression has type t but is here used with type t
```

Ce phénomène se produit aussi avec le compilateur indépendant (en cas de gestion erronée des dépendances de modules). Si l'on rencontre cet étrange message d'erreur, il faut tout recommencer ; soit quitter le système interactif et reprendre une nouvelle session ; soit recompiler entièrement tous les modules de son programme.

Structures de données polymorphes

Toutes les données ne sont pas forcément d'un type de base. Certaines sont *polymorphes* c'est-à-dire qu'elles possèdent un type dont certaines composantes ne sont pas complètement déterminées mais comporte des *variables de type*. L'exemple le plus simple est la liste vide, qui est évidemment une liste d'entiers (`int list`) ou une liste de booléens (`bool list`) et plus généralement une liste de "n'importe quel type", ce que Caml symbolise par `'a` (et la liste vide polymorphe est du type `'a list`).

On définit des structures de données polymorphes en faisant précéder le nom de leur type par la liste de ses paramètres de type. Par exemple :

```
type 'a liste =
   | Nulle
   | Cons of 'a * 'a liste;;
```

ou encore pour des tables polymorphes :

```
type ('a, 'b) table = {nb_entrées : int; contenu : ('a * 'b) vect};;
```

Les listes prédéfinies en Caml forment le type `list` et sont équivalentes à notre type `liste`. La liste vide est symbolisée par `[]` et le constructeur de liste est noté `::`, et correspond à notre constructeur `Cons`. Pour plus de commodité, l'opérateur `::` est infixe : `x :: l` représente la liste qui commence par `x`, suivi des éléments de la liste `l`. En outre, on dispose d'une syntaxe légère pour construire des listes littérales : on écrit les éléments séparés par des point-virgules, le tout entre crochets `[` et `]`.

```
#let l = [1; 2; 3];;
val l : int list = [1; 2; 3]
#let ll = 0 :: l;;
val ll : int list = [0; 1; 2; 3]
```

Les listes sont munies de nombreuses opérations prédéfinies, dont les fonctionnelles de parcours ou d'itération, `map`, `do_list` ou `it_list`, ou encore la concaténation `@`. À titre d'exemple emblématique de fonction définie sur les listes, nous redéfinissons la fonctionnelle `map` qui applique une fonction sur tous les éléments d'une liste.

```
#let print_list l = do_list print_int l;;
val print_list : int list -> unit = <fun>
#print_list l;;
123- : unit = ()
#let rec map f l =
  match l with
  | [] -> []
  | x :: rest -> f x :: map f rest;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
#let succ_l = map (function x -> x + 1) ll;;
val succ_l : int list = [1; 2; 3; 4]
#print_list succ_l;;
1234- : unit = ()
#let somme l = it_list (function x -> function y -> x + y) 0 l;;
val somme : int list -> int = <fun>
#somme ll;;
- : int = 6
#somme succ_l;;
- : int = 10
```

La manipulation des listes est grandement facilitée par la gestion automatique de la mémoire, puisque l'allocation et la désallocation sont prises en charge par le gestionnaire de mémoire et son programme de récupération automatique des structures devenues inutiles (le ramasse-miettes, ou glaneur de cellules, ou GC (*garbage collector*)).

En Caml, la gestion mémoire est automatique.

B.2.11 Modules

On dispose d'un système de modules simple qui définit un module comme un couple de fichiers. Le *fichier d'interface* spécifie les fonctionalités offertes par le module, et le *fichier d'implémentation* contient le code source qui crée ces fonctionalités. Le fichier d'interface a l'extension `.mli` (ml interface) et le fichier d'implémentation l'extension `.ml`. Prenons l'exemple d'un module `Stack` implémentant les piles. Son fichier d'interface est le fichier stack.mli suivant :

```
(* Stacks *)

(* This module implements stacks (LIFOs), with in-place modification. *)

type 'a t;;
       (* The type of stacks containing elements of type ['a]. *)

exception Empty;;
       (* Raised when [pop] is applied to an empty stack. *)

val create: unit -> 'a t
       (* Return a new stack, initially empty. *)
val push: 'a -> 'a t -> unit
       (* [push x s] adds the element [x] at the top of stack [s]. *)
val pop: 'a t -> 'a
       (* [pop s] removes and returns the topmost element in stack [s],
          or raises [Empty] if the stack is empty. *)
val clear : 'a t -> unit
       (* Discard all elements from a stack. *)
val length: 'a t -> int
       (* Return the number of elements in a stack. *)
val iter: ('a -> 'b) -> 'a t -> unit
       (* [iter f s] applies [f] in turn to all elements of [s], from the
          element at the top of the stack to the element at the
          bottom of the stack. The stack itself is unchanged. *)
;;
```

L'interface déclare les signatures des objets fournis par le module, types, exceptions ou valeurs. Une implémentation répondant à cette spécification est :

```
type 'a t = { mutable c : 'a list };;

let create () = { c = [] };;

let clear s = s.c <- [];;

let push x s = s.c <- x :: s.c;;

let pop s =
  match s.c with
  | hd::tl -> s.c <- tl; hd
  | []     -> raise Empty;;

let length s = List.length s.c;;

let iter f s = List.iter f s.c;;
```

La compilation de l'interface du module Stack produit un fichier stack.cmi et celle de l'implémentation le fichier stack.cmo. Quand le module Stack est compilé, on dispose de ses fonctionalités en écrivant la ligne

```
open Stack;;
```

en tête du fichier qui l'utilise. L'appel direct des identificateurs fournis par le module utilise la *notation qualifiée*, qui consiste à suffixer le nom du module par le symbole . suivi de l'identificateur. Ainsi Stack.pop désigne la fonction pop du module Stack. Nous avons déjà utilisé la notation dans nos programmes pour désigner les exceptions Sys.Break. De même, il est courant de ne pas ouvrir le module Printf pour un simple appel à la fonction d'impression formatée : on appelle directement la fonction Printf par son nom qualifié Printf.printf.

Pour qu'on puisse accéder aux identificateurs du module `Stack`, il faut que ce module soit accessible, c'est-à-dire résidant dans le répertoire de travail actuel ou bien dans la bibliothèque standard de Objective Caml, ou bien encore dans l'un des répertoires indiqués sur la ligne de commande du compilateur avec l'option `-I`. Lors de la création de l'exécutable, il faut également demander l'édition des liens de tous les modules utilisés dans l'application (autres que les modules de la bibliothèque standard).

Les modules permettent évidemment la compilation séparée, ce qui sous le système Unix s'accompagne de l'utilisation de l'utilitaire `make`. Nous donnons donc un *makefile* minimum pour gérer un programme découpé en modules. On s'inspirera de ce fichier pour créer ses propres makefiles. Dans ce squelette de fichier make, la variable `OBJS` est la liste des modules, et `EXEC` contient le nom de l'exécutable à fabriquer. Pour simplifier, on suppose qu'il y a deux modules seulement `module1` et `module2`, et que l'exécutable s'appelle `prog`.

```
CAMLC=ocamlc -W -g -I .

OBJS= module1.zo module2.zo
EXEC= prog

all: $(OBJS)
	$(CAMLC) -o $(EXEC) $(OBJS)

clean:
	rm -f *.cm[io] *.cmix *~ #*#

depend:
	mv Makefile Makefile.bak
	(sed -n -e '1,/^### DO NOT DELETE THIS LINE/p' Makefile.bak; \
	 camldep *.mli *.ml) > Makefile
	rm Makefile.bak

.SUFFIXES:
.SUFFIXES: .ml .mli .zo .zi

.mli.cmi:
	$(CAMLC) -c $<
.ml.cmo:
	$(CAMLC) -c $<

### EVERYTHING THAT GOES BEYOND THIS COMMENT IS GENERATED
### DO NOT DELETE THIS LINE
```

La commande `make all` ou simplement `make`, refabrique l'exécutable et `make clean` efface les fichiers compilés. Les dépendances entre modules sont automatiquement recalculées en lançant `make depend`. La commande `camldep` est un `perl` script qui se trouve à l'adresse
`http ://pauillac.inria.fr/caml/FAQ/FAQ_EXPERT-fra.html#make`.

Bibliothèques

Les bibliothèques du système Caml résident dans le répertoire `lib` de l'installation pour les bibliothèques de base indispensables. Les bibliothèques auxiliaires sont installées au même endroit ; leur source se trouve dans le répertoire `contrib` de la distribution. Une documentation minimale est incluse dans les fichiers d'interface, sous la forme de commentaires. Sous Unix, il est très utile d'utiliser la commande `camlbrowser` (d'habitude créée lors de l'intallation du système), pour parcourir les librairies ou ses propres sources. Parmi les bibliothèques, nous citons simplement la bibliothèque du générateur de nombres aléatoires, celle de traitement de grammaires et celle des utilitaires graphiques.

Nombres aléatoires

Pour les essais, il est souvent pratique de disposer d'un générateur de nombres aléatoires. Le module `Random` propose la fonction `Random.int` qui renvoie un nombre aléatoire compris entre 0 inclus et son argument exclus. `Random.float` est analogue mais renvoie un nombre flottant. La fonction `Random.init` permet d'initialiser le générateur avec un nombre entier.

Analyse syntaxique et lexicale

Il existe une interface avec les générateurs d'analyseurs syntaxiques et lexicaux d'Unix (Yacc et Lex). Les fichiers d'entrée de camllex et camlyacc ont les extensions `.mll` et `.mly`. Sur le fonctionnement de ces traits avancés, consulter la documentation du langage.

B.2.12 Fonctions graphiques

Les primitives graphiques sont indépendantes de la machine. Sur les micros-ordinateurs le graphique est intégré à l'application ; sur les machines Unix, il faut appeler une version interactive spéciale `ocamlg` (avec toutes les fonctions graphiques préchargées), qui s'obtient par la commande `ocamlmktop -o ocamlg graphics.cma`. On accède aux primitives graphiques en ouvrant le module `Graphics`. On crée la fenêtre de dessin en appelant la fonction `open_graph` qui prend en argument une chaîne de description de la géométrie de la fenêtre (par défaut, une chaîne vide assure un comportement raisonnable). La description de la fenêtre dépend de la machine et n'est pas obligatoirement prise en compte par l'implémentation de la bibliothèque.

Un programme qui utilise le graphique commence donc par ces deux lignes :

```
open Graphics;;
open_graph "";;
```

La taille de l'écran graphique dépend de l'implémentation, mais l'origine du système de coordonnées est toujours en bas et à gauche. L'axe des abscisses va classiquement de la gauche vers la droite et l'axe des ordonnées du bas vers le haut. Il y a une notion de point courant et de crayon avec une taille et une couleur courantes. On déplace le crayon, sans dessiner ou en dessinant des segments de droite par les fonctions suivantes :

`moveto x y` déplace le crayon aux coordonnées absolues (`x`, `y`).

`lineto x y` trace une ligne depuis le point courant jusqu'au point de coordonnées (`x`, `y`).

`plot x y` trace le point (`x`, `y`).

`set_color c` fixe la couleur du crayon. (Les couleurs sont obtenues par la fonction `rgb` ; on dispose aussi des constantes `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan` et `magenta`.)

`set_line_width n` change la largeur du trait du crayon.

`draw_arc x y rx ry a1 a2` trace un arc d'ellipse de centre (`x`, `y`) de rayon horizontal `rx`, vertical `ry`, de l'angle `a1` à l'angle `a2` (en degrés).

`draw_ellipse x y rx ry` trace une ellipse de centre (`x`, `y`) de rayon horizontal `rx`, et de rayon vertical `ry`.

`draw_circle x y r` trace un cercle centre (`x`, `y`) et de rayon `r`.

On peint l'intérieur de ces courbes avec `fill_rect`, `fill_arc`, `fill_ellipse`, `fill_circle`. Dans la fenêtre graphique, on imprime avec les fonctions :

`draw_char c` affiche le caractère `c` au point courant dans la fenêtre graphique.

`draw_string s` affiche la chaîne de caractères `s`.

`close_graph ()` détruit la fenêtre graphique.

`clear_graph ()` efface l'écran.

`size_x ()` renvoie la taille de l'écran en abscisses.

`size_y ()` renvoie la taille de l'écran en ordonnées..

`background` et `foreground` sont respectivement les couleurs du fond et du crayon.

`point_color x y` renvoie la couleur du point (`x`, `y`).

`current_point ()` renvoie la position du point courant.

`button_down` renvoie vrai si le bouton de la souris est enfoncé, faux sinon.

`mouse_pos ()` renvoie les cordonnées du curseur de la souris.

`read_key ()` attend l'appui sur une touche, la renvoie.

`key_pressed ()` renvoie vrai si une touche est enfoncée, faux sinon.

Plus généralement, un certain nombre d'événements observent l'interaction entre la machine et l'utilisateur :

`wait_next_event evl` attend jusqu'à ce que l'un des événements de la liste `evl` se produise, et renvoie le statut de la souris et du clavier à ce moment-là. Les événements possibles sont :
- `Button_down` : le boutton de la souris a été pressé.
- `Button_up` : le boutton de la souris a été relaché.
- `Key_pressed` : une touche a été appuyée.
- `Mouse_motion` : la souris a bougé.
- `Poll` : ne pas attendre, retourner de suite.

Nous ne décrirons pas ici les primitives de manipulation des images qu'on trouve dans le module `graphics`.

Un exemple

Nous faisons rebondir une balle dans un rectangle, première étape vers un jeu de *ping-pong*.

```
open Graphics;;
open_graph "";;

let c = 5;;           (* Le rayon de la balle  *)

let draw_balle x y =
 set_color foreground; fill_circle x y c;;

let clear_balle x y =
 set_color background; fill_circle x y c;;

let get_mouse () =
 while not (button_down ()) do () done;
 mouse_pos();;

let wait () = for i = 0 to 1000 do () done;;

let v_x = 2           (* Vitesse de déplacement de la balle *)
and v_y = 3;;

let x_min = 2 * c + v_x;; (* Cadre autorisé*)
let x_max = size_x () - x_min;;
let y_min = 2 * c + v_y;;
let y_max = size_y () - y_min;;

let rec pong_aux x y dx dy =
 draw_balle x y;
 let new_x = x + dx
 and new_y = y + dy in
 let new_dx =
  if new_x <= x_min || new_x >= x_max then (- dx) else dx
 and new_dy =
  if new_y <= y_min || new_y >= y_max then (- dy) else dy in
 wait ();
 clear_balle x y;
 pong_aux new_x new_y new_dx new_dy;;

let pong () = clear_graph(); pong_aux 20 20 v_x v_y;;

pong ();;
```

Les adeptes du style impératif écriraient plutôt la procédure `pong` ainsi :

```
let dx = ref v_x
and dy = ref v_y;;

let pong () =
 clear_graph();
 let x = ref 20
 and y = ref 20 in
 while true do
  let cur_x = !x
  and cur_y = !y in
  draw_balle cur_x cur_y;
  x := cur_x + !dx;
  y := cur_y + !dy;
  if !x <= x_min || !x >= x_max then dx := - !dx;
  if !y <= y_min || !y >= y_max then dy := - !dy;
  wait ();
  clear_balle cur_x cur_y
 done;;
```

Documentation

La documentation de Caml se trouve évidemment dans le manuel de référence [2]. On trouve aussi de la documentation en anglais, sous forme d'aide en ligne sur les micros ordinateurs Macintosh et PC. Beaucoup d'informations sont disponibles directement sur la *toile* ou *World Wide Web* :

`http ://caml.inria.fr/index-fra.html` : site principal de Caml.

`http ://caml.inria.fr/ocaml/htmlman/index.html` : manuel en anglais.

`http ://caml.inria.fr/hump.html` : bibliothèque et outils.

`http ://caml.inria.fr/FAQ/index-fra.html` : la *FAQ* de Caml, les questions et réponses fréquemment posées au sujet du langage.

B.3 Syntaxe BNF de Caml

Nous donnons une syntaxe BNF (*Backus Naur Form*) étendue. Chaque règle de la grammaire définit un non-terminal par une production. Le non-terminal défini est à gauche du symbole ::=, la production à droite. Un non-terminal est écrit *ainsi*, les mots clés et symboles terminaux `ainsi`. Dans les membres droits de règles, le symbole | signifie l'alternative, les crochets indiquent une partie optionnelle [ainsi], les accolades une partie qui peut être répétée un nombre quelconque de fois { ainsi }, tandis qu'un symbole + en exposant des accolades indique une partie répétée au moins une fois, { ainsi }$^+$. Dans les règles lexicales les trois points ..., situés entre deux caractères a et b, indiquent l'ensemble des caractères entre a et b dans l'ordre du code ASCII. Finalement, les parenthèses servent au groupement (ainsi).

Règles de grammaires

implementation ::= {*impl-phrase* `;;`}

impl-phrase ::= *expression* | *value-definition*
  | *type-definition* | *exception-definition* | *directive*

value-definition ::= `let` [`rec`] *let-binding* {`and` *let-binding*}

let-binding ::= *pattern* `=` *expression* | *variable pattern-list* `=` *expression*

interface ::= {*intf-phrase* `;;`}

intf-phrase ::= *value-declaration* | *type-definition* | *exception-definition* | *directive*

value-declaration ::= `value` *ident* `:` *type-expression* {`and` *ident* `:` *type-expression*}

expression ::= *primary-expression*
  | *construction-expression*
  | *nary-expression*
  | *sequencing-expression*

primary-expression ::= *ident* | *variable* | *constant* | `(` *expression* `)`
  | `begin` *expression* `end` | `(` *expression* `:` *type-expression* `)`

construction-expression ::= *ncconstr expression*
  | *expression* `,` *expression* {`,` *expression*}
  | *expression* `::` *expression* | `[` *expression* {`;` *expression*} `]`
  | `[|` *expression* {`;` *expression*} `|]`
  | `{` *label* `=` *expression* {`;` *label* `=` *expression*} `}`
  | `function` *simple-matching* | `fun` *multiple-matching*

nary-expression ::= *expression expression*
| *prefix-op expression* | *expression infix-op expression*
| *expression* & *expression* | *expression* or *expression*
| *expression* . *label* | *expression* . *label* <- *expression*
| *expression* .(*expression*) | *expression* .(*expression*) <- *expression*
| *expression* .[*expression*] | *expression* .[*expression*] <- *expression*

sequencing-expression ::= *expression* ; *expression*
| if *expression* then *expression* [else *expression*]
| match *expression* with *simple-matching*
| try *expression* with *simple-matching*
| while *expression* do *expression* done
| for *ident* = *expression* (to | downto) *expression* do *expression* done
| let [rec] *let-binding* {and *let-binding*} in *expression*

simple-matching ::= *pattern* -> *expression* {| *pattern* -> *expression*}

multiple-matching ::= *pattern-list* -> *expression* {| *pattern-list* -> *expression*}

pattern-list ::= *pattern* {*pattern*}

prefix-op ::= - | -. | !

infix-op ::= + | - | * | / | mod | +. | -. | *. | /. | @ | ^ | ! | :=
| = | <> | == | != | < | <= | > | <= | <. | <=. | >. | <=.

pattern ::= *ident* | *constant* | (*pattern*) | (*pattern* : *type-expression*)
| *ncconstr pattern*
| *pattern* , *pattern* {, *pattern*}
| *pattern* :: *pattern* | [*pattern* {; *pattern*}]
| { *label* = *pattern* {; *label* = *pattern*} }
| *pattern* | *pattern*
| _ | *pattern* as *ident*

exception-definition ::= exception *constr-decl* {and *constr-decl*}

type-definition ::= type *typedef* {and *typedef*}

typedef ::= *type-params ident* = *constr-decl* {| *constr-decl*}
| *type-params ident* = { *label-decl* {; *label-decl*} }
| *type-params ident* == *type-expression*
| *type-params ident*

type-params ::= *nothing* | ' *ident* | (' *ident* {, ' *ident*})

constr-decl ::= *ident* | *ident* of *type-expression*

label-decl ::= *ident* : *type-expression* | mutable *ident* : *type-expression*

type-expression ::= ' *ident* | (*type-expression*)
| *type-expression* -> *type-expression* | *type-expression* {* *type-expression*}$^+$
| *typeconstr* | *type-expression typeconstr*
| (*type-expression* {, *type-expression*}) *typeconstr*

constant ::= *integer-literal* | *float-literal* | *char-literal* | *string-literal* | *cconstr*

global-name ::= *ident* | *ident* __ *ident*

variable ::= *global-name* | prefix *operator-name*

operator-name ::= + | - | * | / | mod | +. | -. | *. | /.
| @ | ^ | ! | := | = | <> | == | != | !
| < | <= | > | <= | <. | <=. | >. | <=.

cconstr ::= *global-name* | [] | ()

ncconstr ::= *global-name* | prefix ::

typeconstr ::= *global-name*

label ::= *global-name*

directive ::= # open *string* | # close *string* | # *ident string*

Précédences

Les précédences relatives des opérateurs et des constructions non fermées sont données dans l'ordre décroissant. Chaque nouvelle ligne indique une précédence décroissante. Sur chaque ligne les opérateurs de même précédence sont cités séparés par des blancs ; ou séparés par une virgule suivie du mot *puis*, en cas de précédence plus faible. La décoration [droite] ou [gauche] signifie que le ou les opérateurs qui précédent possèdent l'associativité correspondante.
Les précédences relatives des opérations dans les expressions sont les suivantes :

Accès : `!`, *puis* `.  .(  .[`
Applications : *application de fonction* [droite], *puis application de constructeur*
Arithmétique : `- -.` (*unaire*), *puis* `mod` [gauche], *puis* `* *. / /.` [gauche], *puis* `+ +. - -.` [gauche]
Opérations non arithmétiques : `::` [droite], *puis* `@ ^` [droite]
Conditions : (`= == <` etc.) [gauche], *puis* `not`, *puis* `&` [gauche], *puis* `or` [gauche]
Paires : `,`
Affectations : `<- :=` [droite]
Constructions : `if`, *puis* `;` [droite], *puis* `let match fun function try`.

Les précédences relatives des opérations dans les filtres sont les suivantes :
application de constructeur, *puis* `::` [droite], *puis* `,`, *puis* `|` [gauche], *puis* `as`.
Les précédences relatives des opérations dans les types sont les suivantes :
application de constructeur de type, *puis* `*`, *puis* `->` [droite].

Règles lexicales

ident ::= *letter* {*letter* | `0...9` | `_`}
letter ::= `A...Z` | `a...z`
integer-literal ::= [`-`] {`0...9`}$^+$
| [`-`] (`0x` | `0X`) {`0...9` | `A...F` | `a...f`}$^+$
| [`-`] (`0o` | `0O`) {`0...7`}$^+$
| [`-`] (`0b` | `0B`) {`0...1`}$^+$
float-literal ::= [`-`] {`0...9`}$^+$ [`.` {`0...9`}] [(`e` | `E`) [`+` | `-`] {`0...9`}$^+$]
char-literal ::= `` ` `` *regular-char* `` ` ``
| `` ` `` `\` (`\` | `` ` `` | `n` | `t` | `b` | `r`) `` ` ``
| `` ` `` `\` (`0...9`) (`0...9`) (`0...9`) `` ` ``
string-literal ::= `"` {*string-character*} `"`
string-character ::= *regular-char*
| `\` (`\` | `"` | `n` | `t` | `b` | `r`)
| `\` (`0...9`) (`0...9`) (`0...9`)

Ces identificateurs sont des mots-clés réservés :

and	as	begin	do	done	downto
else	end	exception	for	fun	function
if	in	let	match	mutable	not
of	or	prefix	rec	then	to
try	type	value	where	while	with

Les suites de caractères suivantes sont aussi des mots clés :

```
#    !    !=   &    (    )    *    *.   +    +.
,    -    -.   ->   .    .(   .[   /    /.   :
::   :=   ;    ;;   <    <.   <-   <=   <=.  <>
<>.  =    =.   ==   >    >.   >=   >=.  @    [
[|   ]    ^    _    __   {    |    |]   }    '
```

Les ambiguités lexicales sont résolues par la règle du plus long préfixe (ou "longest match") : quand une suite de caractères permet de trouver plusieurs lexèmes, le plus long est choisi.

Bibliographie

[1] *Le langage Caml*, Pierre Weis et Xavier Leroy, InterEditions, 1993, ISBN 2-7296-0493-6.

[2] *Manuel de Référence du langage Caml*, Xavier Leroy et Pierre Weis, InterEditions, 1993, ISBN 2-7296-0492-8.

[3] *Approche fonctionnelle de la programmation*, Guy Cousineau et Michel Mauny, Ediscience (Collection Informatique), 1995, ISBN 2-84074-114-8.

[4] *Concepts et outils de programmation – le style fonctionnel, le style impératif avec CAML et Ada* Thérèse Accart Hardin et Véronique Donzeau-Gouge Viguié, InterEditions, 1991, ISBN 2-7296-0419-7.

[5] *Option informatique*, Denis Monasse, Vuibert (Enseignement supérieur et Informatique), 1996, ISBN 2-7117-8831-8

[6] Lawrence C. Paulson. ML for the working programmer. Cambridge University Press, 1991.

[7] *Edinburgh LCF*, M.J. Gordon, R. Milner, C. Wadsworth, LNCS 78, 1979.

[8] *Elements of ML programming*, Jeffrey D. Ullman, Prentice Hall, 1994.

[9] *The definiton of Standard ML*, Robin Milner, Mads Tofte, Robert Harper, The MIT Press, 1990.

[10] *Développement d'applications avec Objective Caml*, Emmanuel Chailloux, Pascal Manoury, et Bruno Pagano, Éditions O'Reilly, Paris, avril 2000, ISBN 2-84177-121-0.

Bibliographie

[1] Harold Abelson, Gerald J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[2] Jon Bentley, *Programming Pearls*, Addison Wesley, 1986.

[3] Robert Cori, Jean-Jacques Lévy, *Algorithmes et programmation*, Ecole polytechnique, 1992. Sur le Web : `www.enseignement.polytechnique.fr/informatique/TC/polycopie-1.6`

[4] Patrick Cousot, *Introduction à l'algorithmique et à la programmation*, Ecole polytechnique, Cours d'Informatique, 1986.

[5] David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmeticc*, Computing Surveys, 23(1), Mars 1991.

[6] Eric Raymond, *The New Hacker's Dictionary*, dessins de Guy L. Steele Jr., MIT Press 1991.

Algorithmes et graphes

[7] Claude Berge, *La théorie des graphes et ses applications*, Dunod, Paris, 1966.

[8] Jean Berstel, Jean-Eric Pin, Michel Pocchiola, *Mathématiques et Informatique*, McGraw-Hill, 1991.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Algorithms*, MIT Press, 1990.

[10] Shimon Even, *Graph Algorithms*, Computer Science Press, Potomac, Md, 1979.

[11] Gaston H. Gonnet, Riccardo Baeza-Yates, *Handbook of Algorithms and Data Structures, In Pascal and C*, Addison Wesley, 1991.

[12] Ron L. Graham, Donald E. Knuth, Oren Patashnik, *Concrete mathematics : a foundation for computer science*, Addison Wesley, 1989.

[13] Donald E. Knuth, *Fundamental Algorithms. The Art of Computer Programming*, vol 1, Addison Wesley, 1968.

[14] Donald E. Knuth, *Seminumerical algorithms, The Art of Computer Programming*, vol 2, Addison Wesley, 1969.

[15] Donald E. Knuth, *Sorting and Searching. The Art of Computer Programming*, vol 3, Addison Wesley, 1973.

[16] M. Lothaire, *Combinatorics on Words*, Encyclopedia of Mathematics, Cambridge University Press, 1983.

[17] Udi Manber, *Introduction to Algorithms, A creative approach*, Addison Wesley, 1989

[18] A. Sainte-Laguë, *Les réseaux (ou graphes)*, Mémoire des Sciences Mathématiques (18), 1926.

[19] Bob Sedgewick, *Algorithms*, 2nd edition, Addison-Wesley, 1988. En français : *Algorithmes en langage C*, trad. par Jean-Michel Moreau, InterEditions, 1991.

[20] Robert E. Tarjan, *Depth First Search and linear graph algorithms*, Siam Journal of Computing, 1, pages 146-160, 1972.

Analyse syntaxique et compilation

[21] Al V. Aho, Ravi Sethi, Jeff D. Ullman, *Compilers : Principles, Techniques, and Tools*, Addison Wesley, 1986. En français : *Compilateurs : principes, techniques et outils*, trad. par Pierre Boullier, Philippe Deschamp, Martin Jourdan, Bernard Lorho, Monique Mazaud, InterÉditions, 1989.

[22] Andrew W. Appel, *Modern Compiler Implementation in Java*, Cambridge University Press 1998.

[23] Gerry Kane, *Mips, RISC Architecture*, MIPS Computer Systems, Inc., Prentice Hall, 1987.

[24] Brian Randell, L. J. Russel, *Algol 60 Implementation*, Academic Press, New York, 1964.

Modules et objets

[25] Martín Abadi et Luca Cardelli, *A Theory of Objects*, Springer Verlag, 1996.

[26] Adele Goldberg, *Smalltalk-80 : the language and its implementation*, Addison-Wesley 1983.

Correction de programmes et logique

[27] Henk Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, North Holland, 1981.

[28] Richard B. Kieburtz, *Structured Programming And Problem Solving With Algol W*, Prentice Hall, 1975.

[29] John C. Reynolds, *Theories of Programming Languages*, Cambridge University Press, 1998.

[30] Jan van Leeuwen, *Handbook of theoretical computer science*, volumes A et B, MIT press, 1990.

Concurrence

[31] Mordechai Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall, 1982.

[32] Greg Nelson, *Systems Programming with Modula-3*, Prentice Hall, 1991.

Automates et calculabilité

[33] Noam Chomsky, Marcel Paul Schützenberger, *The algebraic theory of context free languages* dans *Computer Programming and Formal Languages*, P. Braffort, D. Hirschberg ed. North Holland, Amsterdam, 1963

[34] Stephen C. Kleene, *Introduction to Metamathematics*, North Holland, 6ème édition, 1971. (1ère en 1952).

[35] Hartley Rogers, *Theory of recursive functions and effective computability*, MIT press, 1987, (édition originale McGraw-Hill, 1967).

Graphique

[36] Adobe Systems Inc., *PostScript Language, Tutorial and Cookbook*, Addison Wesley, 1985.

[37] J.D. Foley , A. van Dam , S.K. Feiner, J.F. Hughes, *Computer Graphics - principles and practice*, 2nd ed., Addison Wesley , 1990.

[38] Brian W. Kernighan, *PIC–a language for typesetting graphics*, Software Practice & Experience 12 (1982), 1-20.

[39] Donald E. Knuth, *The Metafont book*, Addison Wesley, 1986.

Langages de programmation

[40] Guy Cousineau, Michel Mauny, *Approche fonctionnelle de la programmation*, Ediscience International, Collection Informatique, 1995. ISBN 2-84074-114-8

[41] Mike J. C. Gordon, Robin Milner, Lockwood Morris, Malcolm C. Newey, Chris P. Wadsworth, *A metalanguage for interactive proof in LCF*, In 5th ACM Symposium on Principles of Programming Languages, 1978, ACM Press, New York.

[42] Samuel P. Harbison, *Modula-3*, Prentice Hall, 1992.

[43] Kathleen Jensen, Niklaus Wirth, *PASCAL user manual and report : ISO PASCAL standard*, Springer, 1991. (1ère édition en 1974).

[44] Brian W. Kernighan, Dennis M.Ritchie, *The C programming language*, Prentice Hall, 1978. En français : *Le Langage C*, trad. par Thierry Buffenoir, Manuels informatiques Masson, 8ème tirage, 1990.

[45] Robin Milner, *A proposal for Standard ML*, In ACM Symposium on LISP and Functional Programming, pp 184-197, 1984, ACM Press, New York.

[46] Robin Milner, Mads Tofte, Robert Harper, *The definiton of Standard ML*, The MIT Press, 1990.

[47] Martin Richards, *The portability of the BCPL compiler*, Software Practice and Experience 1 :2, pp. 135-146, 1971.

[48] Martin Richards, Colin Whitby-Strevens, *BCPL : The Language and its Compiler*, Cambride University Press, 1979.

[49] Ravi Sethi, *Programming Languages, Concepts and Constructs*, Addison Wesley, 1989.

[50] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, 1986.

[51] Pierre Weis, Xavier Leroy, *Le langage Caml*, InterEditions, 1993.

Général

[52] John H. Hennessy, David A. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishers, Inc. , 1990.

[53] Donald E. Knuth, *The TEXbook*, Addison Wesley, 1984.

[54] Leslie Lamport, *LaTeX, User's guide & Reference Manual*, Addison Wesley, 1986.

[55] Butler W. Lampson et Ken A. Pier, *A Processor for a High-Performance Personal Computer*, Xerox Palo Alto Research Center Report CSL-81-1. 1981 (aussi dans *Proceedings of Seventh Symposium on Computer Architecture*, SigArch/IEEE, La Baule, Mai 1980, pp. 146–160.

[56] Bob Metcalfe, D. Boggs, *Ethernet : Distributed Packet Switching for Local Computer Networks*, Communications of the ACM 19,7, Juillet 1976, pp 395–404.

[57] Denis M. Ritchie et Ken Thompson, *The UNIX Time-Sharing System*, Communications of the ACM, 17, 7, Juillet 1974, pp 365–375 (aussi dans The Bell System Technical Journal, 57,6, Juillet-Aout 1978).

[58] Chuck P. Thacker, Ed M. McCreight, Butler W. Lampson, R. F. Sproull, D. R. Boggs, *Alto : A Personal Computer*, Xerox-PARC, CSL-79-11, 1979 (aussi dans *Computer Structures : Readings and Examples, 2nd edition*, par Siewoiorek, Bell et Newell).

Table des figures

Index