Algorithmes et Programmation

Ecole Polytechnique

Robert Cori Jean-Jacques Lévy

Table des matières

Avant-propos

Nous remercions Serge Abiteboul, François Anceau, Jean Berstel, Thierry Besançon, Jean Betréma, François Bourdoncle, Philippe Chassignet, Georges Gonthier, Florent Guillaume, Martin Jourdan, François Morain, Dominique Perrin, Jean-Eric Pin, Nicolas Pioch, Bruno Salvy, Michel Weinfeld, Paul Zimmermann pour avoir relu avec attention ce cours, Michel Mauny et Didier Rémy pour leurs macros TEX, Ravi Sethi pour nous avoir donné les sources `pic` des dessins de ses livres [3, 48], Martin Jourdan pour avoir fourni ces sources, Georges Gonthier pour sa connaissance de C, de `pic` et du reste, Damien Doligez pour ses talents de metteur au point, pour les dessins, les scripts en Perl et sa grande connaissance du Macintosh, Xavier Leroy pour ses magnifiques *shell-scripts*, Bruno Salvy pour sa grande connaissance de Maple, Pierre Weis pour son aide en CAML, Paul Zimmermann pour sa rigueur, Philippe Chassignet pour son expertise en graphique Macintosh ... et tous ceux que nous avons oubliés involontairement.

Nous devons spécialement mentionner Pierre Weis, auteur de l'annexe Caml, et Damien Doligez et Xavier Leroy, auteurs de l'annexe sur Unix, ainsi que Dominique Moret et Gérard Guillerm du Centre Informatique pour l'adaptation de cette annexe au contexte de l'X.

Bruno Blanchet, François Bourdoncle, Philippe Chassignet, et Georges Gonthier ont également contribué à l'adaptation du polycopié au langage Java.

Enfin, Pascal Brisset et à Luc Maranget ont traduit le polycopié en document HTML, maintenant consultable sur le *World Wide Web* à l'adresse

`http://w3.edu.polytechnique.fr/informatique/info1.html/`

Polycopié, version 1.7

Introduction

L'informatique qui est au coeur de la révolution due à l'essor des technologies de l'information est directement responsable de la croissance actuelle des pays occidentaux. Les indices de chiffre d'affaires des secteurs de services dans l'informatique avaient déjà très vivement augmenté depuis 40 ans; ils ont connu ces deux dernières années une progession spectaculaire dépassant 30%. Une manière de pensée et de communication a découlé de cette rapide évolution. L'informatique apparaît comme une discipline scientifique introduisant des problèmes nouveaux ou faisant revivre d'autres. Certains pensent que l'informatique est la partie constructive des mathématiques, puisqu'on ne s'y contente pas de théorèmes d'existence, mais du calcul des objets. D'autres y voient les mathématiques concrètes, puisque le domaine a l'exactitude et la rigueur des mathématiques, et que la confrontation des idées abstraites à la réalisation de programmes interdit le raisonnement approximatif. Une autre communauté est intéressée surtout par la partie physique du domaine, car une bonne part des progrès de l'informatique est due au développement foudroyant de la micro-électronique.

La jeunesse de l'informatique permet à certains de nier son aspect scientifique: "les ordinateurs ne sont que des outils pour faire des calculs ou des machines traitement de texte". Malheureusement, beaucoup de "fous de la programmation" étayent l'argument précédent en ignorant toute considération théorique qui puisse les aider dans leurs constructions souvent très habiles. Regardons la définition du mot *hacker* fournie dans *The New Hacker's Dictionary* [40], dictionnaire relu et corrigé électroniquement par une bonne partie de la communauté informatique.

> **hacker** *[originally, someone who makes furniture with an axe] n. 1. A person who enjoys exploring the details of programmable systems and how to stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary. 2. One who programs enthusiastically (even obsessively) or who enjoys programming rather than just theorizing about programming. 3. A person capable of appreciating* hack value. *4. A person who is good at programming quickly. 5. An expert at a particular program, or one who frequently does work using it or on it; as in "a Unix hacker". 6. An expert or enthusiast of any kind. One might be an astronomy hacker, for example. 7. One who enjoys the intellectual challenge of creatively overcoming or circumventing limitations. 8. [deprecated] A malicious meddler who tries to discover sensitive information by poking around. Hence* password hacker, network hacker. *See* **cracker**.

Hacker est un mot courant pour désigner un programmeur passionné. On peut constater toutes les connotations contradictoires recouvertes par ce mot. La définition la plus infamante est "*2. One who … enjoys programming rather than just theorizing about programming.*". Le point de vue que nous défendrons dans ce cours sera quelque peu différent. Il existe une théorie informatique (ce que ne contredit pas la définition précédente) et nous essaierons de démontrer qu'elle peut aussi se révéler utile. De manière assez extraordinaire, peu de disciplines offrent la possibilité de passer des connaissances théoriques à l'application pratique aussi rapidement. Avec les systèmes informatiques modernes, toute personne qui a l'idée d'un algorithme peut s'asseoir derrière un terminal, et sans passer par une lourde expérimentation peut mettre en pratique son idée. Bien sûr, l'apprentissage d'une certaine gymnastique est nécessaire au début, et la construction de gros systèmes informatiques est bien plus longue, mais il s'agit alors d'installer un nouveau service, et non pas d'expérimenter une idée. *En informatique, théorie et pratique sont très proches.*

D'abord, il n'existe pas une seule théorie de l'informatique, mais plusieurs théories imbriquées: logique et calculabilité, algorithmique et analyse d'algorithmes, conception et sémantique des langages de programmation, bases de données, principes des systèmes d'exploitation, architectures des ordinateurs et évaluation de leurs performances, réseaux et protocoles, langages formels et compilation, codes et cryptographie, apprentissage et *zero-knowledge algorithms*, calcul formel, démonstration automatique, conception et vérification de circuits, vérification et validation de programmes, temps réel et logiques temporelles, traitement d'images et vision, synthèse d'image, robotique, ...

Chacune des approches précédentes a ses problèmes ouverts, certains sont très célèbres. Un des plus connus est de savoir si $P = NP$, c'est-à-dire si tous les algorithmes déterministes en temps polynomial sont équivalents aux algorithmes non déterministes polynomiaux. Plus concrètement, peut-on trouver une solution polynomiale au problème du voyageur de commerce. (Celui-ci a un ensemble de villes à visiter et il doit organiser sa tournée pour qu'elle ait un trajet minimal en kilomètres). Un autre est de trouver une sémantique aux langages pour la programmation objet. Cette technique de programmation incrémentale est particulièrement utile dans les gros programmes graphiques. A ce jour, on cherche encore un langage de programmation objet dont la sémantique soit bien claire et qui permette de programmer proprement. Pour résumer, *en informatique, il y a des problèmes ouverts.*

Quelques principes généraux peuvent néanmoins se dégager. D'abord, en informatique, il est très rare qu'il y ait une solution unique à un problème donné. Tout le monde a sa version d'un programme, contrairement aux mathématiques où une solution s'impose relativement facilement. Un programme ou un algorithme se trouve très souvent par raffinements successifs. On démarre d'un algorithme abstrait et on évolue lentement par optimisations successives vers une solution détaillée et pouvant s'exécuter sur une machine. C'est cette diversité qui donne par exemple la complexité de la correction d'une composition ou d'un projet en informatique (à l'Ecole Polytechnique par exemple). C'est aussi cette particularité qui fait qu'il y a une grande diversité entre les programmeurs. *En informatique, la solution unique à un problème n'existe pas.*

Ensuite, les systèmes informatiques représentent une incroyable construction, une cathédrale des temps modernes. Jamais une discipline n'a si rapidement fait un tel empilement de travaux. Si on essaie de réaliser toutes les opérations nécessaires pour déplacer un curseur sur un écran avec une souris, ou afficher l'écho de ce qu'on frappe sur un clavier, on ne peut qu'être surpris par les différentes constructions intellectuelles que cela a demandées. On peut dire sans se tromper que l'informatique sait utiliser la transitivité. Par exemple, ce polycopié a été frappé à l'aide du traitement de texte LaTeX [31] de L. Lamport (jeu de macros TeX de D. Knuth [26]), les caractères ont été calculés en METAFONT de D. Knuth [27], les dessins en `gpic` version Gnu (R. Stallman) de `pic` de B. Kernighan [23]. On ne comptera ni le système Macintosh (qui dérive fortement de l'Alto et du Dorado faits à Xerox PARC [51, 32]), ni OzTeX (TeX sur Macintosh), ni les éditeurs QED, Emacs, Alpha, ni le système Unix fait à Bell laboratories [44], les machines Dec Stations 3100, 5000, Sun Sparc2, ni leurs composants MIPS 2000/3000

[21] ou Sparc, ni le réseau Ethernet (Xerox-PARC [36]) qui permet d'atteindre les serveurs fichiers, ni les réseaux Transpac qui permettent de travailler à la maison, ni les imprimantes Laser et leur langage PostScript, successeur d'InterPress (J. Warnock à Xerox-PARC, puis Adobe Systems) [2], qui permettent l'impression du document. On peut évaluer facilement l'ensemble à quelques millions de lignes de programme et également à quelques millions de transistors. Rien de tout cela n'existait en 1960. Tout n'est que gigantesque mécano, qui s'est assemblé difficilement, mais dont on commence à comprendre les critères nécessaires pour en composer les différents éléments. *En informatique, on doit composer beaucoup d'objets.*

Une autre remarque est la rapidité de l'évolution de l'informatique. Une loi due à B. Joy dit que les micro-ordinateurs doublent de vitesse tous les deux ans, et à prix constant. Cette loi n'a pas été invalidée depuis 1978! (cf. le livre de Hennessy et Patterson [19]). Si on regarde les densités des mémoires ou des disques, on constate que des machines de taille mémoire honorable de 256 kilo-Octets en 1978 ont au minimum 32 Méga-Octets aujourd'hui. De même, on est passé de disques de 256 MO de 14 pouces de diamètre et de 20 cm de hauteur à des disques 3,5 pouces de 4 GO (Giga-Octets) de capacité et de 5cm de hauteur. Une machine portable de 3,1 kg peut avoir un disque de 500 MO. Il faut donc penser tout projet informatique en termes évolutifs. Il est inutile d'écrire un programme pour une machine spécifique, puisque dans deux ans le matériel ne sera plus le même. Tout programme doit être pensé en termes de portabilité, il doit pouvoir fonctionner sur toute machine. C'est pourquoi les informaticiens sont maintenant attachés aux standards. Il est frustrant de ne pouvoir faire que des tâches fugitives. Les standards (comme par exemple le système Unix ou le système de fenêtres X-Window qui sont des standards *de facto*) assurent la pérennité des programmes. *En informatique, on doit programmer en utilisant des standards.*

Une autre caractéristique de l'informatique est le côté instable des programmes. Ils ne sont souvent que gigantesques constructions, qui s'écroulent si on enlève une petite pierre. Le 15 janvier 1990, une panne téléphonique a bloqué tous les appels longue distance aux Etats-Unis pendant une après-midi. Une instruction `break` qui arrêtait le contrôle d'un `for` dans un programme C s'est retrouvée dans une instruction `switch` qui avait été insérée dans le `for` lors d'une nouvelle version du programme de gestion des centraux d'AT&T. Le logiciel de test n'avait pas exploré ce recoin du programme, qui n'intervenait qu'accidentellement en cas d'arrêt d'urgence d'un central. Le résultat de l'erreur fut que le programme marcha très bien jusqu'à ce qu'intervienne l'arrêt accidentel d'un central. Celui-ci, à cause de l'erreur, se mit à avertir aussitôt tous ses centraux voisins, pour leur dire d'appliquer aussi la procédure d'arrêt d'urgence. Comme ces autres centraux avaient tous aussi la nouvelle version du programme, ils se mirent également à parler à leurs proches. Et tout le système s'écroula en quelques minutes. Personne ne s'était rendu compte de cette erreur, puisqu'on n'utilisait jamais la procédure d'urgence et, typiquement, ce programme s'est effondré brutalement. Il est très rare en informatique d'avoir des phénomènes continus. Une panne n'est en général pas le résultat d'une dégradation perceptible. Elle arrive simplement brutalement. C'est ce côté exact de l'informatique qui est très attrayant. En informatique, il y a peu de solutions approchées. *En informatique, il y a une certaine notion de l'exactitude.*

Notre cours doit être compris comme une initiation à l'informatique, et plus exactement à l'algorithmique et à la programmation. Il s'adresse à des personnes dont nous tenons pour acquises les connaissances mathématiques: nulle part on n'expliquera ce qu'est une récurrence, une relation d'équivalence ou une congruence. Il est orienté vers l'algorithmique, c'est-à-dire la conception d'algorithmes (et non l'analyse très précise de leur performance), et la programmation, c'est-à-dire leur implantation pratique sur ordinateur (et non l'étude d'un — ou de plusieurs — langage(s) de programmations).

Les cours d'algorithmique sont maintenant bien catalogués, et la référence principale sur laquelle nous nous appuierons est le livre de Sedgewick [47]. Une autre référence intéressante par sa complétude et sa présentation est le livre de Cormen, Leiserson et Rivest [10]. Il existe bien d'autres ouvrages: Gonnet et Baeza-Yates [15], Berstel-Pin-Pocchiola [8], Manber [35], Graham-Knuth-Patashnik [17].... Il faut aussi bien sûr signaler les livres de Knuth [28, 29, 30]. Les cours de programmation sont moins clairement définis. Ils dépendent souvent du langage de programmation ou du type de langage de programmation choisi. Nous nous appuierons sur le polycopié de P. Cousot [11], sur le livre de Kernighan et Ritchie pour C [22], sur le livre de Wirth pour Pascal [20], sur le livre de Nelson [39] ou Harbison pour Modula-3[18].

Le cours est organisé selon les structures de données et les méthodes de programmation, utilisées dans différents algorithmes. Ainsi seront considérés successivement les tableaux, les listes, les piles, les arbres, les files de priorité, les graphes. En parallèle, on regardera différentes méthodes de programmation telles que la récursivité, les interfaces ou modules, la compilation séparée, le *backtracking*, la programmation dynamique. Enfin, nous essaierons d'enchaîner sur différents thèmes. Le premier d'entre eux sera la programmation symbolique. Un autre sera la validation des programmes, notamment en présence de phénomènes asynchrones. Enfin, on essaiera de mentionner les différentes facettes des langages de programmation (*garbage collection*, exceptions, modules, polymorphisme, programmation incrémentale, surcharge). Nous fournirons pour chaque algorithme deux versions du programme (quand nous en montrerons son implémentation): une dans le langage Java et une autre en Caml (langage enseigné en classes préparatoires). Une introduction à Java figurera en annexe, ainsi qu'un rappel de Caml. La lecture des chapitres et des annexes peut se mener en parallèle. Le lecteur qui connaît les langages de programmation n'aura pas besoin de consulter les annexes. Celui qui démarre en Java ou en Caml devra plutôt commencer par les appendices.

Le choix de considérer les langages Java et Caml est dicté par deux considérations. Ce sont d'abord des langages fortement typés. Il est impossible de finir un programme par une violation de la mémoire ou une erreur système irrémissible. Les deux langages vérifient la validité des index dans les accès aux tableaux ou des références dans les accès aux structures de données dynamiques. Par ailleurs, les deux langages ont une gestion automatique de la mémoire, car ils disposent d'un glaneur de cellules (*garbage collector* en anglais) permettant de récupérer automatiquement l'espace mémoire perdu. Cela simplifie notablement l'écriture des programmes et permet de se concentrer sur leur essence. L'allocation mémoire, elle, reste toujours explicite et on pourra toujours comprendre l'espace mémoire exigé par un programme. Autrefois, le cours étais fait en Pascal et en C; il sera possible de disposer d'appendices donnant les programmes de ce cours dans ces deux langages.

Historiquement, Java est un langage orienté-objet, quoique nous utiliserons peu cette particularité dans notre cours qui doit garder un aspect élémentaire. Il provient de Simula-67, Smalltalk, Modula-3[39] et C++[49]. Caml est un langage plutôt fonctionnel, dialecte de ML, dont il existe plusieurs implémentations: Caml et SML/NJ développés à l'INRIA et à Bell laboratories. Il provient de Lisp et de Scheme[1]. Depuis peu de temps, Caml dispose d'une très efficace extention orientée objet, Objective Caml (Ocaml). Les deux langages n'ont pas l'efficacité de C ou C++, quoique ceci reste à démontrer dans la manipulation des structures de données dynamiques. Mais à l'ère des machines RISC, il vaut mieux essayer d'écrire des programmes simples et compréhensibles. C'est le style que nous voulons adopter ici. La structure de nos programmes sera très souvent indépendante du langage de programmation, comme en attesteront les quatre versions de chacun de nos programmes en Pascal, C, Caml ou Java.

Pour être bien clair, prenons l'exemple du calcul de π. Un programme C esthétiquement correct, qui utilise la formule $\pi/4 = \arctan(1) = \sum_{n=0}^{n=\infty} (-1)^n/(2n+1)$, est

```
#include <stdio.h>

#define N 10000

main()
{
    float pi = 0;
    int sign = 1;
    int n = 1;
    int m = 1;

    while (n < N) {
        pi = pi + (float)sign / m;
        m = m + 2;
        sign = -sign;
        ++n;
    }
    pi = 4 * pi;
    printf ("%.6f\n", pi);
}
```

Mais le programme C suivant [40] est interdit dans notre cours.

```
#define _ -F<00 || --F-OO--;
int F=00,OO=00;
main(){F_OO();printf("%1.3f\n", 4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_
            _-_-_-_
 }
```

Ce programme fait un nombre inconsidéré d'appels au préprocesseur et beaucoup trop d'effets de bord (cf. page 197). Il est illisible. Il faut avoir une conception simple et donc esthétique des programmes. On n'est plus à l'époque où tout se joue à l'instruction machine près. La puissance des ordinateurs a rendu très viable la programmation sans optimisations excessives. Tout l'art du programmeur est de comprendre les quelques endroits critiques d'un programme, en général très peu nombreux, où les optimisations sont nécessaires. Donc le programme cryptique précédent, ou tout autre bourré d'optimisations inutiles sera mauvais et donc interdit.

Dans cet exemple, typiquement, la convergence du calcul de π est mauvaise. Inutile donc de chercher à optimiser ce programme. Il faut changer de formule pour le calcul. La formule de John Machin (1680-1752) fait converger plus vite en calculant $\pi/4 =$

$4\arctan(1/5) - \arctan(1/239)$. Une autre technique consiste à taper sur la commande suivante sur une machine de l'Ecole Polytechnique

```
% maple
    |\^/|      MAPLE V
._|\|   |/|_.  Copyright (c) 1981-1990 by the University of Waterloo.
 \  MAPLE  /   All rights reserved.  MAPLE is a registered trademark of
 <____ ____>   Waterloo Maple Software.
      |        Type ? for help.
> evalf(Pi,1000);

3.1415926535897932384626433832795028841971693993751058209749445923078164\
0628620899862803482534211706798214808651328230664709384460955058223172\
5359408128481117450284102701938521105559644622948954930381964428810975\
6659334461284756482337867831652712019091456485669234603486104543266482\
1339360726024914127372458700660631558817488152092096282925409171536436\
7892590360011330530548820466521384146951941511609433057270365759591953\
0921861173819326117931051185480744623799627495673518857527248912279381\
8301194912983367336244065664308602139494639522473719070217986094370277\
0539217176293176752384674818467669405132000568127145263560827785771342\
7577896091736371787214684409012249534301465495853710507922796892589235\
4201995611212902196086403441815981362977477130996051870721134999999837\
2978049951059731732816096318595024459455346908302642522308253344685035\
2619311881710100031378387528865875332083814206171776691473035982534904\
2875546873115956286388235378759375195778185778053217122680661300192787\
66111959092164201989

> quit;
%
```

Complexité des algorithmes

Dans ce cours, il sera question de complexité d'algorithmes, c'est-à-dire du nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par l'algorithme. Elle s'exprime en fonction de la taille n des données. On dit que la complexité de l'algorithme est $O(f(n))$ où f est d'habitude une combinaison de polynômes, logarithmes ou exponentielles. Ceci reprend la notation mathématique classique, et signifie que le nombre d'opérations effectuées est borné par $cf(n)$, où c est une constante, lorsque n tend vers l'infini.

Considérer le comportement à l'infini de la complexité est justifié par le fait que les données des algorithmes sont de grande taille et qu'on se préoccupe surtout de la croissance de cette complexité en fonction de la taille des données. Une question systématique à se poser est: que devient le temps de calcul si on multiplie la taille des données par 2?

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité.

- Les algorithmes sub-linéaires, dont la complexité est en général en $O(\log n)$. C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de cardinal n.
- Les algorithmes linéaires en complexité $O(n)$ ou en $O(n \log n)$ sont considérés comme rapides, comme l'évaluation de la valeur d'une expression composée de n symboles ou les algorithmes optimaux de tri.
- Plus lents sont les algorithmes de complexité située entre $O(n^2)$ et $O(n^3)$, c'est le cas de la multiplication des matrices et du parcours dans les graphes.
- Au delà, les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$ sont considérés comme lents, sans parler des algorithmes exponentiels (dont la complexité est supérieure à tout polynôme en n) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

La recherche de l'algorithme ayant la plus faible complexité, pour résoudre un problème donné, fait partie du travail régulier de l'informaticien. Il ne faut toutefois pas tomber dans certains excès, par exemple proposer un algorithme excessivement alambiqué, développant mille astuces et ayant une complexité en $O(n^{1,99})$, alors qu'il existe un algorithme simple et clair de complexité $O(n^2)$. Surtout, si le gain de l'exposant de n s'accompagne d'une perte importante dans la constante multiplicative: passer d'une complexité de l'ordre de $n^2/2$ à une complexité de $10^{10} n \log n$ n'est pas vraiment une amélioration. Les critères de clarté et de simplicité doivent être considérés comme aussi importants que celui de l'efficacité dans la conception des algorithmes.

Scalaires

Faisons un rappel succinct du calcul scalaire dans les programmes. Il y a deux grandes catégories d'objets scalaires dans les programmes: les entiers en virgule fixe, les réels en virgule flottante, on peut aussi utiliser ces scalaires à l'aide de représentations symboliques utilisant différents types de bases

Les entiers

Le calcul entier est relativement simple. Les nombres sont en fait calculés dans une arithmétique modulo $N = 2^n$ où n est le nombre de bits des mots machines.

n	N		Exemple
16	65 536	$= 6 \times 10^4$	Macintosh SE/30
32	4 294 967 296	$= 4 \times 10^9$	Pentium
64	18 446 744 073 709 551 616	$= 2 \times 10^{19}$	Alpha

FIG. 1 – *Bornes supérieures des nombres entiers*

Ainsi, les processeurs de 1999 peuvent avoir une arithmétique précise sur quelques milliards de milliards, 100 fois plus rapide qu'une machine 16 bits! Il faut toutefois faire attention à la multiplication ou pire à la division qui a tendance sur les machines RISC (*Reduced Instruction Set Computers*) à prendre beaucoup plus de temps qu'une simple addition, typiquement 10 fois plus sur le processeur R3000 de Mips. Cette précision peut être particulièrement utile dans les calculs pour accéder aux fichiers. En effet, on a des disques de plus de 4 Giga Octets actuellement, et l'arithmétique 32 bits est insuffisante pour adresser leurs contenus.

Il faut pouvoir aussi utiliser des nombres négatifs. La notation couramment utilisée est celle du complément à 2. En notation binaire, le bit le plus significatif est le bit de signe. Au lieu de parler de l'arithmétique entre 0 et $2^n - 1$, les nombres sont pris entre -2^{n-1} et $2^{n-1} - 1$. Soit, pour $n = 16$, sur l'intervalle $[-32768, 32767]$. En Java, `Integer.MAX_VALUE` $= 2^{n-1} - 1$ est le plus grand entier positif et `Integer.MIN_VALUE` $= -2^{n-1}$ le plus petit entier négatif. En Caml, `max_int` $= 2^{n-2} - 1$ et `min_int` $= -2^{n-2}$.

Une opération entre 2 nombres peut créer un débordement, c'est-à-dire dépasser les bornes de l'intervalle. Mais elle respecte les règles de l'arithmétique modulo N. Selon le langage de programmation et son implémentation sur une machine donnée, ce débordement est testé ou non. Ni Java, ni Caml ne font ces tests.

Les nombres flottants

La notation flottante sert à représenter les nombres réels afin d'atteindre des valeurs impossibles à obtenir en notation fixe. Un nombre flottant a une partie significative, *la mantisse*, et une partie *exposant*. Un nombre flottant tient souvent sur le même nombre de bits n qu'un nombre entier. En flottant, on décompose $n = s + p + q$ en trois champs pour le signe, la mantisse et l'exposant, qui sont donnés par la machine. Ainsi tout nombre réel écrit "*signe decimal* e *exposant*" en Java ou Caml, vaut

$$\textit{signe decimal} \times 10^{exposant}$$

Les nombres flottants sont une approximation des nombres réels, car leur partie significative f ne tient que sur un nombre p de bits fini. Par exemple, $p = 23$ en simple précision. Le nombre de bits pour l'exposant e est $q = 8$ en simple précision, ce qui fait que le nombre de bits total, avec le bit de signe, est bien 32.

Pour rendre portables des programmes utilisant les nombres flottants, une norme IEEE 754 (*the Institute of Electrical and Electronics Engineers*) a été définie. Non seulement elle décrit les bornes des nombres flottants, mais elle donne une convention pour représenter des valeurs spéciales: $\pm\infty$, NaN (*Not A Number*) qui permettent de donner des valeurs à des divisions par zéro, ou à des racines carrées de nombres négatifs par exemple. Les valeurs spéciales permettent d'écrire des programmes de calcul de racines de fonctions éventuellement discontinues.

La norme IEEE est la suivante

Exposant	Mantisse	Valeur
$e = e_{min} - 1$	$f = 0$	± 0
$e = e_{min} - 1$	$f \neq 0$	$0, f \times 2^{e_{min}}$
$e_{min} \leq e \leq e_{max}$		$1, f \times 2^{e}$
$e = e_{max} + 1$	$f = 0$	$\pm\infty$
$e = e_{max} + 1$	$f \neq 0$	NaN

FIG. 2 – *Valeurs spéciales pour les flottants IEEE*

et les formats en bits simple et double précision sont

Paramètre	Simple	Double
p	23	52
q	8	11
e_{max}	+127	+1023
e_{min}	−126	−1022
Taille totale du mot	32	64

FIG. 3 – *Formats des flottants IEEE*

On en déduit donc que la valeur absolue de tout nombre flottant x vérifie en simple précision

$$10^{-45} \simeq 2^{-150} \leq |x| < 2^{128} \simeq 3 \times 10^{38}$$

et en double précision

$$2 \times 10^{-324} \simeq 2^{-1075} \leq |x| < 2^{1024} \simeq 10^{308}$$

Par ailleurs, la précision d'un nombre flottant est $2^{-23} \simeq 10^{-7}$ en simple précision et $2^{-52} \simeq 2 \times 10^{-16}$ en double précision. On perd donc 2 à 4 chiffres de précision par rapport aux opérations entières. Il faut comprendre aussi que deux nombres flottants sont alignés avant toute addition ou soustraction, ce qui entraîne des pertes de précision. Par exemple, l'addition d'un très petit nombre à un grand nombre va laisser ce dernier inchangé. Il y a alors dépassement de capacité vers le bas (*underflow*). Un bon exercice est de montrer que la série harmonique converge en informatique flottante, ou que l'addition flottante n'est pas associative! Il y a aussi des débordements de capacité vers le haut (*overflows*). Ces derniers sont en général plus souvent testés que les dépassements vers le bas.

Enfin, pour être complet, la représentation machine des nombres flottants est légèrement différente en IEEE. En effet, on s'arrange pour que le nombre 0 puisse être représenté par le mot machine dont tous les bits sont 0, et on additionne la partie exposant du mot machine flottant de $e_{min} - 1$, c'est-à-dire de 127 en simple précision, ou de 1023 en double précision.

Chapitre 1

Tableaux

Les tableaux sont une des structures de base de l'informatique. Un tableau représente selon ses dimensions, un vecteur ou une matrice d'éléments d'un même type. Un tableau permet l'accès direct à un élément, et nous allons nous servir grandement de cette propriété dans les algorithmes de tri et de recherche en table que nous allons considérer.

1.1 Le tri

Qu'est-ce qu'un tri? On suppose qu'on se donne une suite de N nombres entiers $\langle a_i \rangle$, et on veut les ranger en ordre croissant au sens large. Ainsi, pour $N = 10$, la suite

$$\langle 18, 3, 10, 25, 9, 3, 11, 13, 23, 8 \rangle$$

devra devenir

$$\langle 3, 3, 8, 9, 10, 11, 13, 18, 23, 25 \rangle$$

Ce problème est un classique de l'informatique. Il a été étudié en détail, cf. la moitié du livre de Knuth [30]. En tant qu'algorithme pratique, on le rencontre souvent. Par exemple, il faut établir le classement de certains élèves, mettre en ordre un dictionnaire, trier l'index d'un livre, faire une sortie lisible d'un correcteur d'orthographe, ... Il faudra bien faire la distinction entre le tri d'un grand nombre d'éléments (plusieurs centaines), et le tri de quelques éléments (un paquet de cartes). Dans ce dernier cas, la méthode importe peu. Un algorithme amusant, *bogo*-tri, consiste à regarder si le paquet de cartes est déjà ordonné. Sinon, on le jette par terre. Et on recommence. Au bout d'un certain temps, on risque d'avoir les cartes ordonnées. Bien sûr, le *bogo*-tri peut ne pas se terminer. Une autre technique fréquemment utilisée avec un jeu de cartes consiste à regarder s'il n'y a pas une transposition à effectuer. Dès qu'on en voit une à faire, on la fait et on recommence. Cette méthode marche très bien sur une bonne distribution de cartes.

Plus sérieusement, il faudra toujours avoir à l'esprit que le nombre d'objets à trier est important. Ce n'est pas la peine de trouver une méthode sophistiquée pour trier 10 éléments. Pourtant, les exemples traités dans un cours sont toujours de taille limitée, pour des raisons pédagogiques il n'est pas possible de représenter un tri sur plusieurs milliers d'éléments. Le tri, par ses multiples facettes, est un très bon exemple d'école. En général, on exigera que le tri se fasse *in situ*, c'est-à-dire que le résultat soit au même endroit que la suite initiale. On peut bien sûr trier autre chose que des entiers. Il suffit de disposer d'un domaine de valeurs muni d'une relation d'ordre total. On peut donc trier des caractères, des mots en ordre alphabétique, des enregistrements selon un certain champ. On supposera, pour simplifier, qu'il existe une opération d'échange ou plus simplement d'affectation sur les éléments de la suite à trier. C'est pourquoi nous prendrons le cas de valeurs entières.

1.1.1 Méthodes de tri élémentaires

Dans tout ce qui suit, on suppose que l'on trie des nombres entiers et que ceux-ci se trouvent dans un tableau a. L'algorithme de tri le plus simple est le *tri par sélection*. Il consiste à trouver l'emplacement de l'élément le plus petit du tableau, c'est-à-dire l'entier m tel que $a_i \geq a_m$ pour tout i. Une fois cet emplacement m trouvé, on échange les éléments a_1 et a_m. Puis on recommence ces opérations sur la suite $\langle a_2, a_3, \ldots, a_N \rangle$, ainsi on recherche le plus petit élément de cette nouvelle suite et on l'échange avec a_2. Et ainsi de suite ... jusqu'au moment où on n'a plus qu'une suite composée d'un seul élément $\langle a_N \rangle$.

La recherche du plus petit élément d'un tableau est un des premiers exercices de programmation. La détermination de la position de cet élément est très similaire, elle s'effectue à l'aide de la suite d'instructions:

```
m = 0;
for (int j = 1; j < N; ++j)
    if (a[j] < a[m])
        m = i;
```

L'échange de deux éléments nécessite une variable temporaire t et s'effectue par:

```
t = a[m]; a[m] = a[1]; a[1] = t;
```

Il faut refaire cette suite d'opérations en remplacant 1 par 2, puis par 3 et ainsi de suite jusqu'à N. Ceci se fait par l'introduction d'une nouvelle variable i qui prend toutes les valeurs entre 1 et N. Ces considérations donnent lieu au programme présenté en détail ci-dessous. Pour une fois, nous l' écrivons pour une fois en totalité; les procédures d'acquisition des données et de restitution des résultats sont aussi fournies. Pour les autres algorithmes, nous nous limiterons à la description de la procédure effective de tri.

```
class TriSelection {

  final static int N = 10;

  static int[] a = new int[N];    // Le tableau à trier

  static void initialisation() { // On tire au sort des nombres
                                 // entre 0 et 127, en initialisant
                                 // le tirage au sort sur l'heure
      for (int i = 0; i < N; ++i)
          a[i] = (int) (Math.random() * 128);
  }

  static void impression() {
      for (int i = 0; i < N; ++i)
          System.out.print (a[i] + " ");
      System.out.println ("");
  }

  static void triSelection() {
      int   min, t;

      for (int i = 0; i < N - 1; ++i) {
          min = i;
          for (int j = i+1; j < N; ++j)
```

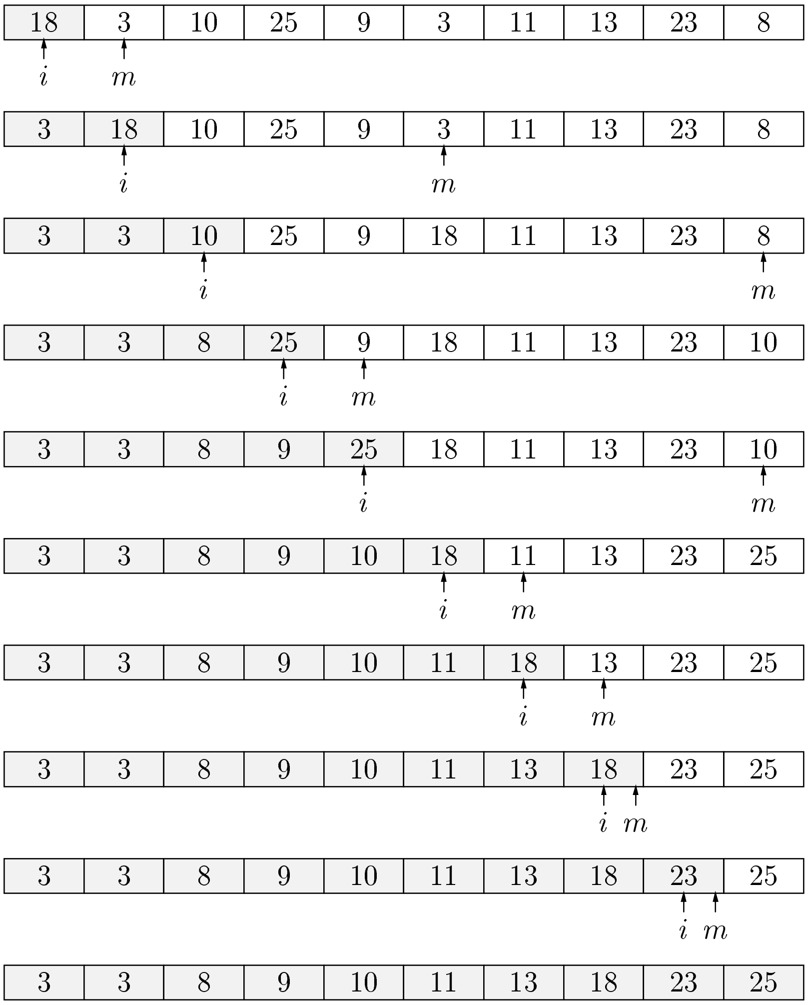


FIG. 1.1 – *Exemple de tri par sélection*

```
                if (a[j] < a[min])
                    min = j;
            t = a[min]; a[min] = a[i]; a[i] = t;
        }
    }

    public static void main (String args[]) {

        initialisation();    // On lit le tableau
        impression();        // et on l'imprime
        triSelection();      // On trie
        impression();        // On imprime le résultat
    }
}
```

Il est facile de compter le nombre d'opérations nécessaires. A chaque itération, on démarre à l'élément a_i et on le compare successivement à a_{i+1}, a_{i+2}, ..., a_N. On fait donc $N-i$ comparaisons. On commence avec $i = 1$ et on finit avec $i = N-1$. Donc on fait $(N-1)+(N-2)+\cdots+2+1 = N(N-1)/2$ comparaisons, et $N-1$ échanges. Le tri par sélection fait donc de l'ordre de N^2 comparaisons. Si $N = 100$, il y a 5000 comparaisons, soit 5 ms si on arrive à faire une comparaison et l'itération de la boucle `for` en 1μs, ce qui est tout à fait possible sur une machine plutôt rapide actuellement. On écrira que le tri par sélection est en $O(N^2)$. Son temps est quadratique par rapport aux nombres d'éléments du tableau.

Une variante du tri par sélection est le *tri bulle*. Son principe est de parcourir la suite $\langle a_1, a_2, \ldots, a_N \rangle$ en intervertissant toute paire d'éléments consécutifs (a_{j-1}, a_j) non ordonnés. Ainsi après un parcours, l'élément maximum se retrouve en a_N. On recommence avec le préfixe $\langle a_1, a_1, \ldots, a_{N-1} \rangle$, ... Le nom de tri bulle vient donc de ce que les plus grands nombres se déplacent vers la droite en poussant des bulles successives de la gauche vers la droite. L'exemple numérique précédent est donné avec le tri bulle dans la figure 1.2.

La procédure correspondante utilise un indice `i` qui marque la fin du préfixe à trier, et l'indice `j` qui permet de déplacer la bulle qui monte vers la borne `i`. On peut compter aussi très facilement le nombre d'opérations et se rendre compte qu'il s'agit d'un tri en $O(N^2)$ comparaisons et éventuellement échanges (si par exemple le tableau est donné en ordre strictement décroissant).

```
static void triBulle() {

  int   t;

  for (int i = N-1; i >= 0; --i)
      for (int j = 1; j <= i; ++j)
          if (a[j-1] > a[j]) {
              t = a[j-1]; a[j-1] = a[j]; a[j] = t;
          }
}
```

1.1.2 Analyse en moyenne

Pour analyser un algorithme de tri, c'est à dire déterminer le nombre moyen d'opérations qu'il effectue, on utilise le modèle des permutations. On suppose dans ce modèle que la suite des nombres à trier est la suite des entiers $1, 2, \ldots n$ et l'on admet que toutes les permutations de ces entiers sont équiprobables. On peut noter que le nombre de comparaisons à effectuer pour un tri ne dépend pas des éléments à trier mais de l'ordre dans lequel ils apparaissent. Les supposer tous compris entre 1 et N n'est donc

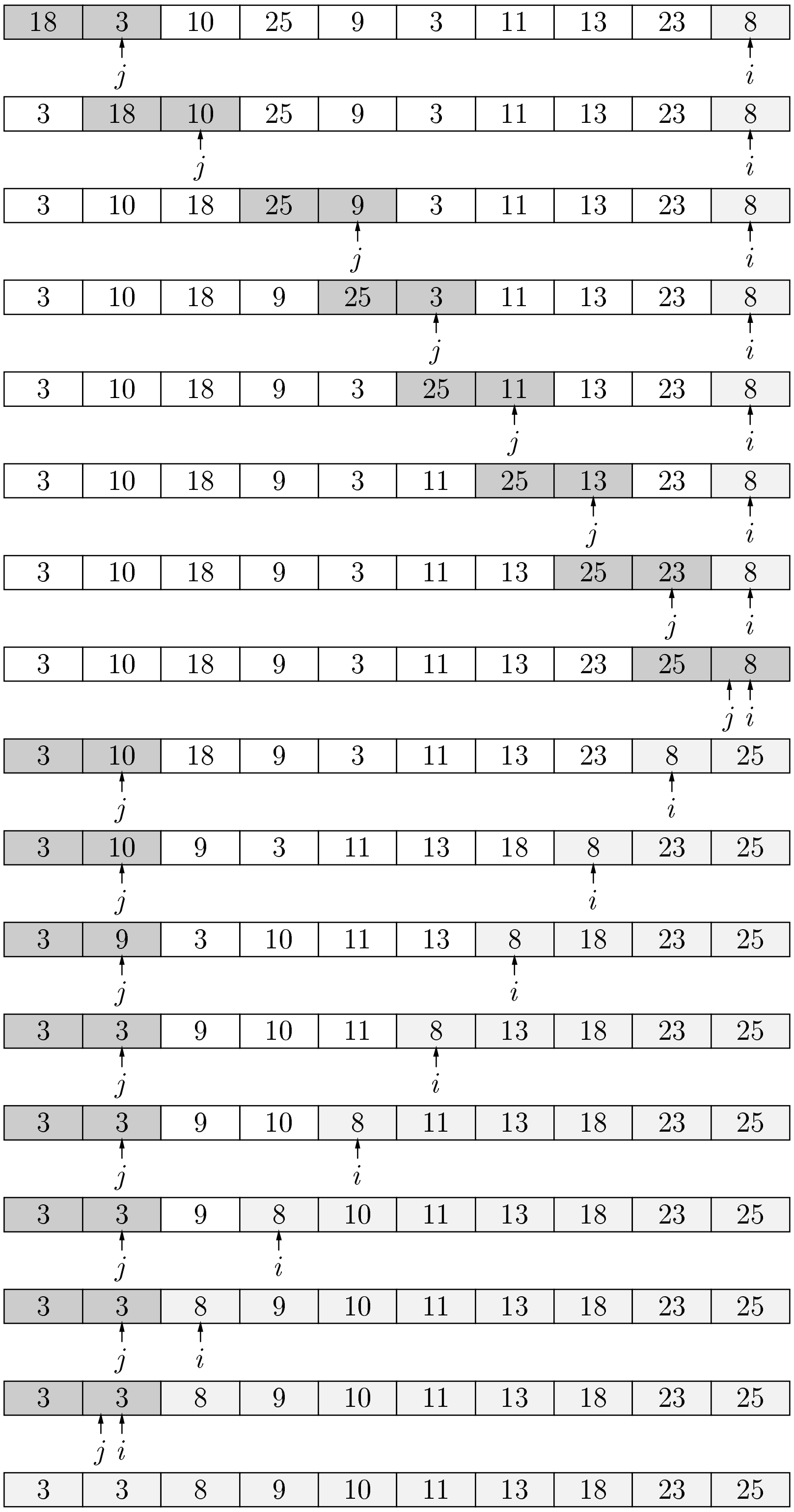


FIG. 1.2 – *Exemple de tri bulle*

pas une hypothèse restrictive si on ne s'intéresse qu'à l'analyse et si l'on se place dans un modèle d'algorithme dont l'opération de base est la comparaison.

Pour une permutation α de $\{1, 2 \dots n\}$ dans lui-même, une inversion est un couple (a_i, a_j) tel que $i < j$ et $a_i > a_j$. Ainsi, la permutation

$$\langle 8, 1, 5, 10, 4, 2, 6, 7, 9, 3\rangle$$

qui correspond à l'ordre des éléments de la figure 1.1, comporte 21 inversions. Chaque échange d'éléments de la procédure `TriBulle` supprime une et une seule inversion et, une fois le tri terminé, il n'y a plus aucune inversion. Ainsi le nombre total d'échanges effectués est égal au nombre d'inversions dans la permutation. Calculer le nombre moyen d'échanges dans la procédure de tri bulle revient donc à compter le nombre moyen d'inversions de l'ensemble des permutations sur N éléments. Un moyen de faire ce calcul consiste à compter le nombre d'inversions dans chaque permutation à faire la somme de tous ces nombres et à diviser par $N!$. Ceci est plutôt fastidieux, une remarque simple permet d'aller plus vite.

L'image miroir de toute permutation $\alpha = \langle a_1, a_2 \dots a_N\rangle$ est la permutation $\beta = \langle a_N, \dots, a_2, a_1\rangle$. Il est clair que (a_i, a_j) est une inversion de α si et seulement si ce n'est pas une inversion de β. La somme du nombre d'inversions de α et de celles de β est $N(N-1)/2$. On regroupe alors deux par deux les termes de la somme des nombres d'inversions des permutations sur N éléments et on obtient que le nombre moyen d'inversions sur l'ensemble des permutations est donné par:

$$\frac{N(N-1)}{4}$$

ce qui est donc le nombre moyen d'échanges dans la procédure `TriBulle`. On note toutefois que le nombre de comparaisons effectuées par `TriBulle` est le même que celui de `TriSelection` soit $N(N-1)/2$.

1.1.3 Le tri par insertion

Une méthode complètement différente est le *tri par insertion*. C'est la méthode utilisée pour trier un paquet de cartes. On prend une carte, puis 2 et on les met dans l'ordre si nécessaire, puis 3 et on met la $3^{\text{ème}}$ carte à sa place dans les 2 premières, ... De manière générale on suppose les $i-1$ premières cartes triées. On prend la $i^{\text{ème}}$ carte, et on essaie de la mettre à sa place dans les $i-1$ cartes déjà triées. Et on continue jusqu'à $i = N$. Ceci donne le programme suivant

```
static void triInsertion() {

  int   j, v;
  for (int i = 1; i < N; ++i) {
      v = a[i]; j = i;
      while (j > 0  &&  a[j-1] > v) {
          a[j] = a[j-1];
          --j;
      }
      a[j] = v;
  }
}
```

Pour classer le $i^{\text{ème}}$ élément du tableau `a`, on regarde successivement en marche arrière à partir du $i-1^{\text{ième}}$. On décale les éléments visités vers la droite pour pouvoir mettre `a[i]` à sa juste place. Le programme précédent contient une légère erreur, si `a[i]` est le plus petit élément du tableau, car on va sortir du tableau par la gauche. On peut

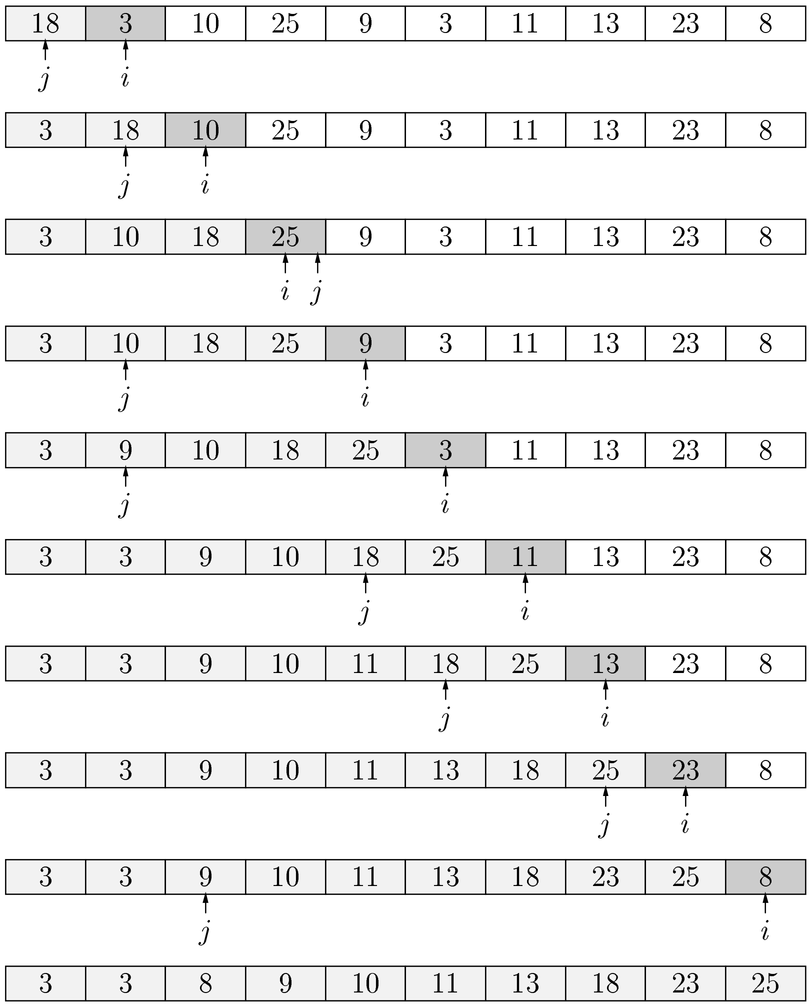


FIG. 1.3 – *Exemple de tri par insertion*

toujours y remédier en supposant qu'on rajoute un élément `a[0]` valant `-maxint`. On dit alors que l'on a mis une *sentinelle* à gauche du tableau `a`. Ceci n'est pas toujours possible, et il faudra alors rajouter un test sur l'indice `j` dans la boucle `while`. Ainsi, pour le tri par insertion, l'exemple numérique précédent est dans la figure 1.3.

Le nombre de comparaisons pour insérer un élément dans la suite triée de ceux qui le précédent est égal au nombre d'inversions qu'il présente avec ceux-ci augmenté d'une unité. Soit c_i ce nombre de comparaisons. On a

$$c_i = 1 + \mathrm{card}\{a_j \mid a_j > a_i, j < i\}$$

Pour la permutation α correspondant à la suite à trier, dont le nombre d'inversions est $inv(\alpha)$, le nombre total de comparaisons pour le tri par insertion est

$$C_\alpha = \sum_{i=2}^{N} c_i = N - 1 + inv(\alpha)$$

D'où le nombre moyen de comparaisons

$$C_N = \frac{1}{N!}\sum_{\alpha} C_\alpha = N - 1 + \frac{N(N-1)}{4} = \frac{N(N+3)}{4} - 1$$

Bien que l'ordre de grandeur soit toujours N^2, ce tri est plus efficace que le tri par sélection. De plus, il a la bonne propriété que le nombre d'opérations dépend fortement de l'ordre initial du tableau. Dans le cas où le tableau est presque en ordre, il y a peu d'inversions et très peu d'opérations sont nécessaires, contrairement aux deux méthodes de tri précédentes. Le tri par insertion est donc un bon tri si le tableau à trier a de bonnes chances d'être presque ordonné.

Une variante du tri par insertion est un tri dû à D. L. Shell en 1959, c'est une méthode de tri que l'on peut sauter en première lecture. Son principe est d'éviter d'avoir à faire de longues chaînes de déplacements si l'élément à insérer est très petit. Nous laissons le lecteur fanatique en comprendre le sens, et le mentionnons à titre anecdotique. Au lieu de comparer les éléments adjacents pour l'insertion, on les compare tous les ..., 1093, 364, 121, 40, 13, 4, et 1 éléments. (On utilise la suite $u_{n+1} = 3u_n + 1$). Quand on finit par comparer des éléments consécutifs, ils ont de bonnes chances d'être déjà dans l'ordre. On peut montrer que le tri Shell ne fait pas plus que $O(N^{3/2})$ comparaisons, et se comporte donc bien sur des fichiers de taille raisonnable (5000 éléments). La démonstration est compliquée, et nous la laissons en exercice difficile. On peut prendre tout autre générateur que 3 pour générer les séquences à explorer. Pratt a montré que pour des séquences de la forme 2^p3^q, le coût est $O(n \log^2 n)$ dans le pire cas (mais il est coûteux de mettre cette séquence des 2^p3^q dans l'ordre). Dans le cas général, le coût (dans le cas le pire) du tri Shell est toujours un problème ouvert. Le tri Shell est très facile à programmer et très efficace en pratique (c'est le tri utilisé dans le noyau Maple).

```
static void triShell() {

  int h = 1; do
     h = 3*h + 1;
  while ( h <= N );
  do {
      h = h / 3;
      for (int i = h; i < N; ++i)
          if (a[i] < a[i-h]) {
              int   v = a[i], j = i;
              do {
```

nom	tel
paul	2811
roger	4501
laure	2701
anne	2702
pierre	2805
yves	2806

FIG. 1.4 – *Un exemple de table pour la recherche en table*

```
                a[j] = a[j-h];
                j = j - h;
            } while (j >= h  &&  a[j-h] > v);
            a[j] = v;
        }
    } while ( h > 1);
}
```

1.2 Recherche en table

Avec les tableaux, on peut aussi faire des tables. Une table contient des informations sur certaines clés. Par exemple, la table peut être un annuaire téléphonique. Les clés sont les noms des abonnés. L'information à rechercher est le numéro de téléphone. Une table est donc un ensemble de paires $\langle \text{nom}, \text{numéro} \rangle$. Il y a plusieurs manières d'organiser cette table: un tableau d'enregistrement, une liste ou un arbre (comme nous le verrons plus tard). Pour l'instant, nous supposons la table décrite par deux tableaux `nom` et `tel`, indicés en parallèle, le numéro de téléphone de `nom[i]` étant `tel[i]`.

1.2.1 La recherche séquentielle

La première méthode pour rechercher un numéro de téléphone consiste à faire une recherche séquentielle (ou linéaire). On examine successivement tous les éléments de la table et on regarde si on trouve un abonné du nom voulu. Ainsi

```
static int recherche (String x) {

  for (int i = 0; i < N; ++i)
      if (x.equals(nom[i]))
          return tel[i];
  return -1;
}
```

qui peut aussi s'écrire

```
static int recherche (String x) {

  int   i = 0;
  while (i < N  &&  !x.equals(nom[i]))
      ++i;
  if (i < N)
      return tel[i];
  else
```

```
        return -1;
}
```

Si on a la place, une autre possibilité est de mettre une *sentinelle* au bout de la table.

```
static int recherche (String x) {

  int  i = 0;
  nom[N] = x; tel[N] = -1;
  while (! x.equals(nom[i]))
      ++i;
  return tel[i];
}
```

L'écriture de la procédure de recherche dans la table des noms est alors plus efficace, car on peut remarquer que l'on ne fait plus qu'un test là où on en faisait deux. La recherche séquentielle est aussi appelée recherche linéaire, car il est facile de montrer que l'on fait $N/2$ opérations en moyenne, et N opérations dans le pire cas. Sur une table de 10000 éléments, la recherche prend 5000 opérations en moyenne, soit 5ms.

Voici un programme complet utilisant la recherche linéaire en table.

```
class Table {

  final static int N = 6;
  static String nom[] = new String[N+1];
  static int tel[] = new int[N+1];

  static void initialisation() {
    nom[0] = "paul";  tel[0] = 2811;
    nom[1] = "roger";  tel[1] = 4501;
    nom[2] = "laure";  tel[2] = 2701;
    nom[3] = "anne";  tel[3] = 2702;
    nom[4] = "pierre";  tel[4] = 2805;
    nom[5] = "yves";  tel[5] = 2806;
  }

  static int recherche (String x) {

    for (int i = 0; i < N; ++i)
        if (x.equals(nom[i]))
            return tel[i];
    return -1;
  }

  public static void main (String args[]) {

    Initialisation();
    if (args.length == 1)
        System.out.println (Recherche(args[0]));
  }
}
```

1.2.2 La recherche dichotomique

Une autre technique de recherche en table est la *recherche dichotomique*. Supposons que la table des noms soit triée en ordre alphabétique (comme l'annuaire des PTT). Au lieu de rechercher séquentiellement, on compare la clé à chercher au nom qui se trouve au milieu de la table des noms. Si c'est le même, on retourne le numéro de téléphone du milieu, sinon on recommence sur la première moitié (ou la deuxième) si le nom recherché est plus petit (ou plus grand) que le nom rangé au milieu de la table. Ainsi

```
static void initialisation() {

  nom[0] = "anne"; tel[0] = 2702;
  nom[1] = "laure"; tel[1] = 2701;
  nom[2] = "paul"; tel[2] = 2811;
  nom[3] = "pierre"; tel[3] = 2805;
  nom[4] = "roger"; tel[4] = 4501;
  nom[5] = "yves"; tel[5] = 2806;
}

static int rechercheDichotomique (String x) {

  int   i, g, d, cmp;

  g = 0; d = N-1;
  do {
      i = (g + d) / 2;
      cmp = x.compareTo(nom[i]);
      if (cmp == 0)
          return tel[i];
      if (cmp < 0)
          d = i - 1;
      else
          g = i + 1;
  } while (g <= d);
  return -1;
}
```

Le nombre C_N de comparaisons pour une table de taille N est tel que $C_N = 1 + C_{\lfloor N/2 \rfloor}$ et $C_0 = 1$. Donc $C_N \simeq \log_2(N)$. (Dorénavant, $\log_2(N)$ sera simplement écrit $\log N$.) Si la table a 10000 éléments, on aura $C_N \simeq 14$. C'est donc un gain sensible par rapport aux 5000 opérations nécessaires pour la recherche linéaire. Bien sûr, la recherche linéaire est plus simple à programmer, et sera donc utilisée pour les petites tables. Pour des tables plus importantes, la recherche dichotomique est plus intéressante.

On peut montrer qu'un temps sub-logarithmique est possible si on connaît la distribution des objets. Par exemple, dans l'annuaire du téléphone, ou dans un dictionnaire, on sait *a priori* qu'un nom commençant par la lettre V se trouvera plutôt vers la fin. En supposant la distribution uniforme, on peut faire une règle de trois pour trouver l'indice de l'élément de référence pour la comparaison, au lieu de choisir le milieu, et on suit le reste de l'algorithme de la recherche dichotomique. Cette méthode est la *recherche par interpolation*. Alors le temps de recherche est en $O(\log \log N)$, c'est-à-dire 4 opérations pour une table de 10000 éléments, et 5 opérations jusqu'à 10^9 entrées dans la table!

1.2.3 Insertion dans une table

Dans la recherche linéaire ou par dichotomie, on ne s'est pas préoccupé de l'insertion dans la table d'éléments nouveaux. C'est par exemple très peu souvent le cas pour un annuaire téléphonique. Mais cela peut être fréquent dans d'autres utilisations, comme la table des usagers d'un système informatique. Essayons de voir comment organiser l'insertion d'éléments nouveaux dans une table, dans le cas des recherches séquentielle et dichotomique.

Pour le cas séquentiel, il suffit de rajouter au bout de la table l'élément nouveau, s'il y a la place. S'il n'y a pas de place, on appelle une procédure `erreur` qui imprimera le message d'erreur donné en paramètre et arrêtera le programme (cf. page 215). Ainsi

```
void insertion (String x, int val) {

    ++n;
    if (n >= N) then
        erreur ("De'bordement de la table");
    nom[n] = x;
    tel[n] = val;
}
```

L'insertion se fait donc en temps constant, en $O(1)$. Dans le cas de la recherche par dichotomie, il faut maintenir la table ordonnée. Pour insérer un nouvel élément dans la table, il faut d'abord trouver son emplacement par une recherche dichotomique (ou séquentielle), puis pousser tous les éléments derrière lui pour pouvoir insérer le nouvel élément au bon endroit. Cela peut donc prendre $\log n + n$ opérations. L'insertion dans une table ordonnée de n éléments prend donc un temps $O(n)$.

1.2.4 Hachage

Une autre méthode de recherche en table est le *hachage*. On utilise une fonction h de l'ensemble des clés (souvent des chaînes de caractères) dans un intervalle d'entiers. Pour une clé x, $h(x)$ est l'endroit où l'on trouve x dans la table. Tout se passe parfaitement bien si h est une application injective. Pratiquement, on ne peut arriver à atteindre ce résultat. On tente alors de s'en approcher et on cherche aussi à minimiser le temps de calcul de $h(x)$. Ainsi un exemple de fonction de hachage est

$$h(x) = (x[1] \times B^{l-1} + x[2] \times B^{l-2} + \cdots + x[l]) \bmod N$$

On prend d'habitude $B = 128$ ou $B = 256$ et on suppose que la taille de la table N est un nombre premier. Pourquoi? D'abord, il faut connaître la structure des ordinateurs pour comprendre le choix de B comme une puissance de 2. En effet, les multiplications par des puissances de 2 peuvent se faire très facilement par des décalages, puisque les nombres sont représentés en base 2. En général, dans les machines "modernes", cette opération est nettement plus rapide que la multiplication par un nombre arbitraire. Quant à prendre N premier, c'est pour éviter toute interférence entre les multiplications par B et la division par N. En effet, si par exemple $B = N = 256$, alors $h(x) = x[l]$ et la fonction h ne dépendrait que du dernier caractère de x. Le but est donc d'avoir une fonction h de hachage simple à calculer et ayant une bonne distribution sur l'intervalle $[0, N-1]$. (Attention: il sera techniquement plus simple dans cette section sur le hachage de supposer que les indices des tableaux varient sur $[0, N-1]$ au lieu de $[1, N]$). Le calcul de la fonction h se fait par la fonction `h(x, l)`, où `l` est la longueur de la chaîne `x`,

```
static int h (String x) {

  int   r = 0;
  for (int i = 0; i < x.length(); ++i)
      r = ((r * B) + x.charAt(i)) % N;
  return r;
}
```

Donc la fonction h donne pour toute clé x une entrée possible dans la table. On peut alors vérifier si $x = \texttt{nom}[h(x)]$. Si oui, la recherche est terminée. Si non, cela signifie que la table contient une autre clé x' telle que $h(x') = h(x)$. On dit alors qu'il y a une *collision*, et la table doit pouvoir gérer les collisions. Une méthode simple est de lister

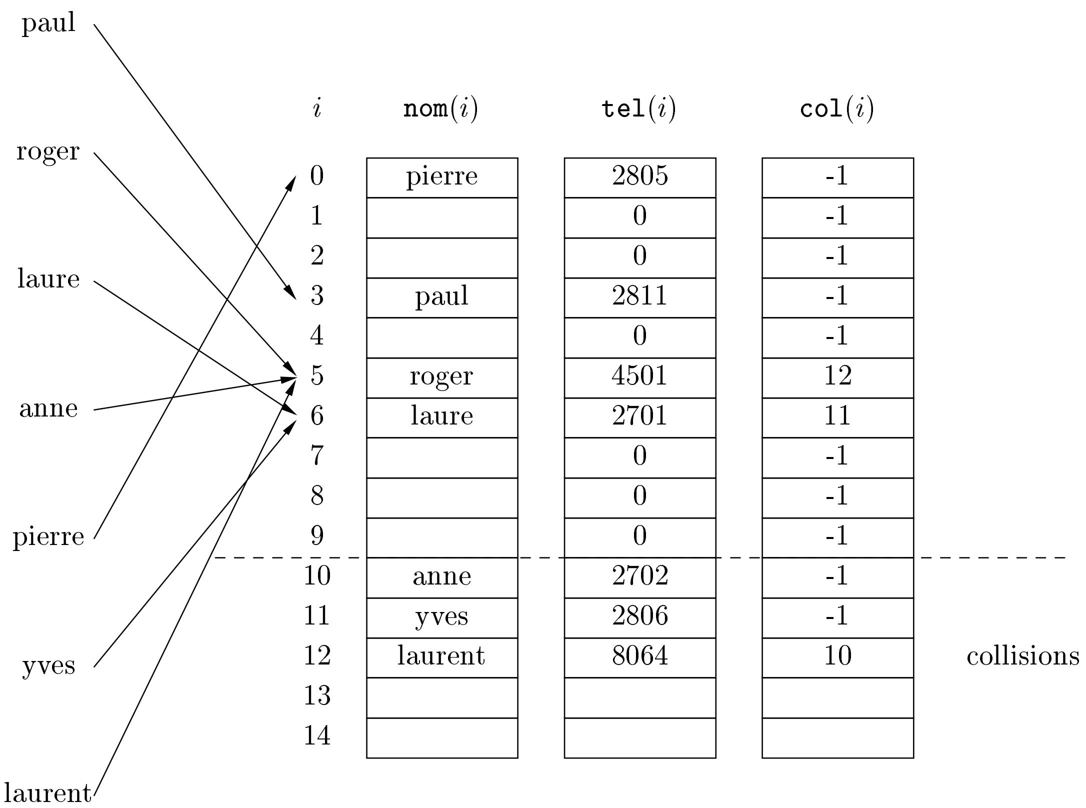


FIG. 1.5 – *Hachage par collisions séparées*

les collisions dans une table `col` parallèle à la table `nom`. La table des collisions donnera une autre entrée i dans la table des noms où peut se trouver la clé recherchée. Si on ne trouve pas la valeur x à cette nouvelle entrée i, on continuera avec l'entrée i' donnée par $i' = \texttt{col}[i]$. Et on continue tant que $\texttt{col}[i] \neq -1$. La recherche est donnée par

```
static int recherche (String x) {

  for (int i = h(x); i != -1; i = col[i])
      if (x.equals(nom[i]))
          return tel[i];
  return -1;
}
```

Ainsi la procédure de recherche prend un temps au plus égal à la longueur moyenne des classes d'équivalence définies sur la table par la valeur de $h(x)$, c'est-à-dire à la longueur moyenne des listes de collisions. Si la fonction de hachage est parfaitement uniforme, il n'y aura pas de collision et on atteindra tout élément en une comparaison. Ce cas est très peu probable. Il y a des algorithmes compliqués pour trouver une fonction de hachage parfaite sur une table donnée et fixe. Mais si le nombre moyen d'éléments ayant même valeur de hachage est $k = N/M$, où M est grosso modo le nombre de classes d'équivalences définies par h, la recherche prendra un temps N/M. Le hachage ne fait donc que réduire d'un facteur constant le temps pris par la recherche séquentielle. L'intérêt du hachage est qu'il est souvent très efficace, tout en étant simple à programmer.

L'insertion dans une table avec le hachage précédent est plus délicate. En effet, on devrait rapidement fusionner des classes d'équivalences de la fonction de hachage, car il faut bien mettre les objets à insérer à une certaine entrée dans la table qui correspond elle-même à une valeur possible de la fonction de hachage. Une solution simple est de supposer la table de taille n telle que $\texttt{N} \leq n \leq \texttt{Nmax}$. Pour insérer un nouvel élément, on regarde si l'entrée calculée par la fonction h est libre, sinon on met le nouvel élément au bout de la table, et on chaîne les collisions entre elles par un nouveau tableau `col`. (Les tableaux `nom`, `tel` et `col` sont maintenant de taille `Nmax`). On peut choisir de mettre le nouvel élément en tête ou à la fin de la liste des collisions; ici on le mettra en tête. Remarque: à la page 67, tous les outils seront développés pour enchaîner les collisions par des listes; comme nous ne connaissons actuellement que les tableaux comme structure de donnée, nous utilisons le tableau `col`. L'insertion d'un nouvel élément dans la table s'écrit

```
static void insertion (String x, int val) {

  int   i = h(x);

  if (nom[i] == null) {
      nom[i] = x;
      tel[i] = val;
  } else
  if (n >= Nmax)
      erreur ("Débordement de la table");
  else {
      nom[n] = x;
      tel[n] = val;
      col[n] = col[i];       // On met la nouvelle entrée en tête
      col[i] = n;            // de la liste des collisions de sa
      ++n;                   // classe d'équivalence.
  }
}
```

Au début, on suppose $\texttt{n} = \texttt{N}$, $\texttt{nom}[i] =$ `""` (chaîne vide) et $\texttt{col}[i] = -1$ pour $0 \leq i < \texttt{N}$. La procédure d'insertion est donc très rapide et prend un temps constant $O(1)$.

Une autre technique de hachage est de faire un hachage *à adressage ouvert*. Le principe en est très simple. Au lieu de gérer des collisions, si on voit une entrée occupée lors de l'insertion, on range la clé à l'entrée suivante (modulo la taille de la table). On suppose une valeur interdite dans les clés, par exemple la chaîne vide `''`, pour désigner une entrée libre dans la table. Les procédures d'insertion et de recherche s'écrivent très simplement comme suit

```
static int recherche (String x) {

  int   i = h(x);

  while (nom[i] != null) {
      if (x.equals(nom[i]))
          return tel[i];
      i = (i+1) % N;
  }
  return -1;
}

static void insertion (String x, int val) {
  int   i;
```

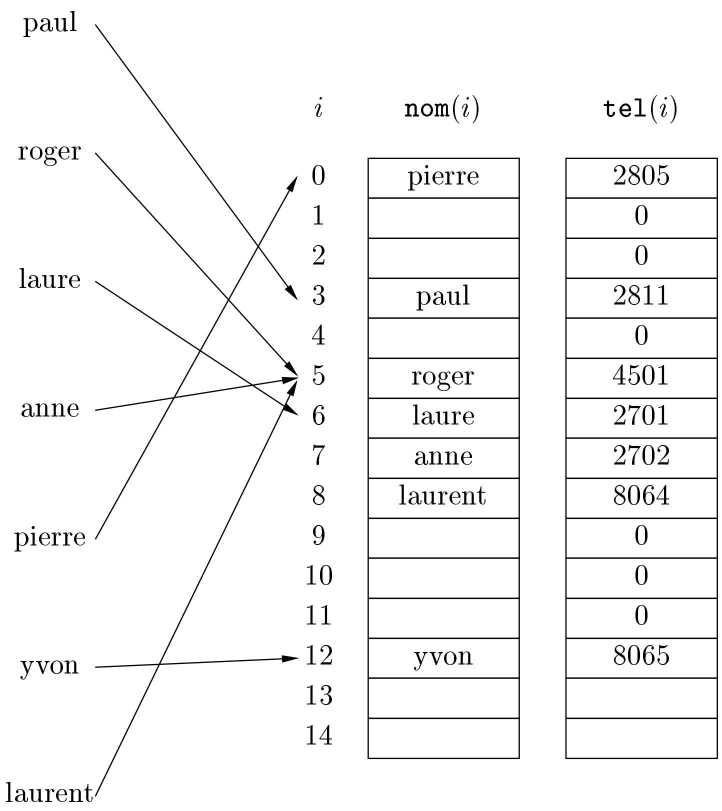


FIG. 1.6 – *Hachage par adressage ouvert*

```
    if (n >= N)
        erreur ("De'bordement de la table");
    ++n;
    i = h(x);
    while ((nom[i] != null) && ! x.equals(nom[i]))
        i = (i+1) % N;
    nom[i] = x;
    tel[i] = val;
}
```

Dans le cas où la clé à insérer se trouverait déjà dans la table, l'ancien numéro de téléphone est écrasé, ce qui est ce que l'on veut dans le cas présent. Il est intéressant de reprendre les méthodes de recherche en table déjà vues et de se poser ce problème. Plus intéressant, on peut se demander si cette méthode simple de hachage linéaire est très efficace, et si on ne risque pas, en fait, d'aboutir à une recherche séquentielle standard. Notre problème est donc de comprendre la contiguïté de la fonction de hachage, et la chance que l'on peut avoir pour une valeur donnée de $h(x)$ d'avoir aussi les entrées $h(x)+1$, $h(x)+2$, $h(x)+3$ ... occupées. On peut démontrer que, si la fonction de hachage est uniforme et si $\alpha =$ `n/Nmax` est le taux d'occupation de la table, le nombre d'opérations est:

- $1/2 + 1/(2(1-\alpha))$ pour une recherche avec succès,
- $1/2 + 1/(2(1-\alpha)^2)$ pour une recherche avec échec.

Donc si $\alpha = 2/3$, on fait 2 ou 5 opérations, si $\alpha = 90\%$, on en fait 5 ou 50. La conclusion est donc que, si on est prêt à grossir la table de 50%, le temps de recherche est très bon, avec une méthode de programmation très simple.

Une méthode plus subtile, que l'on peut ignorer en première lecture, est d'optimiser le hachage à adressage ouvert précédent en introduisant un deuxième niveau de hachage. Ainsi au lieu de considérer l'élément suivant dans les procédures de recherche et d'insertion, on changera les instructions

```
i := (i+1) mod Nmax;
```

en

```
i := (i+u) mod Nmax;
```

où `u` $= h_2(x, l)$ est une deuxième fonction de hachage. Pour éviter des phénomènes de périodicité, il vaut mieux prendre `u` et `Nmax` premiers entre eux. Une méthode simple, comme `Nmax` est déjà supposé premier, est de faire `u` $<$ `Nmax`. Par exemple, $h_2(x,l) = 8 - (x[l] \bmod 8)$ est une fonction rapide à calculer, et qui tient compte des trois derniers bits de x. On peut se mettre toujours dans le cas de distributions uniformes, et de fonctions h et h_2 "indépendantes". Alors on montre que le nombre d'opérations est en moyenne:

- $(1/\alpha) \times \log(1/(1-\alpha))$ pour une recherche avec succès,
- $1/(1-\alpha)$ pour une recherche avec échec,

en fonction du taux d'occupation α de la table. Numériquement, pour $\alpha = 80\%$, on fait 3 ou 5 opérations, pour $\alpha = 99\%$, on fait 7 ou 100. Ce qui est tout à fait raisonnable.

Le hachage est très utilisé pour les correcteurs d'orthographe. McIlroy[1] a calculé que, dans un article scientifique typique, il y a environ 20 erreurs d'orthographe (c'est-à-dire des mots n'apparaissant pas dans un dictionnaire), et qu'une collision pour 100

1. Doug McIlroy était le chef de l'équipe qui a fait le système Unix à Bell laboratories.

papiers est acceptable. Il suffit donc de faire un correcteur probabiliste qui ne risque de se tromper qu'un cas sur 2000. Au lieu de garder tout le dictionnaire en mémoire, et donc consommer beaucoup de place, il a utilisé un tableau de n bits. Il calcule k fonctions h_i de hachage indépendantes pour tout mot w, et regarde si les k bits positionnés en $h_i(w)$ valent simultanément 1. On est alors sûr qu'un mot n'est pas dans le dictionnaire si la réponse est négative, mais on pense qu'on a de bonnes chances qu'il y soit si la réponse est oui. Un calcul simple montre que la probabilité P pour qu'un mot d'un dictionnaire de d entrées ne positionne pas un bit donné est $P = e^{-dk/n}$. La probabilité pour qu'une chaîne quelconque soit reconnue comme un mot du dictionnaire est $(1-P)^k$. Si on veut que cette dernière vaille $1/2000$, il suffit de prendre $P = 1/2$ et $k = 11$. On a alors $n/d = k/\ln 2 = 15,87$. Pour un dictionnaire de 25000 mots, il faut donc 400000 bits (50 kO), et, pour un de 200000 mots, if faut 3200000 bits (400 kO). McIlroy avait un pdp11 et 50 kO était insupportable. Il a compressé la table en ne stockant que les nombres de 0 entre deux 1, ce qui a ramené la taille à 30 kO. Actuellement, la commande `spell` du système Unix utilise $k = 11$ et une table de 400000 bits.[2]

1.3 Programmes en Caml

```
(* Tri par sélection, page 24 *)
#open "printf";;

let nMax = 10;;
let a = make_vect nMax 0;;

let initialisation () =
  for i = 0 to nMax - 1 do
    a.(i) <- random__int 100
  done;;

let impression () =
  for i = 0 to nMax - 1 do
    printf "%3d " a.(i)
  done;
  printf "\n";;

let tri_sélection () =
  let min = ref 0 in
  for i = 0 to nMax - 2 do
    min := i;
    for j = i + 1 to nMax - 1 do
      if a.(j) < a.(!min) then min := j
    done;
    let t = a.(!min) in
    a.(!min) <- a.(i); a.(i) <- t
  done;;

let main () =
  (* On initialise le tableau *)
  initialisation ();
```

2. Paul Zimmermann a remarqué que les dictionnaires français sont plus longs que les dictionnaires anglais à cause des conjugaisons des verbes. Il a reprogrammé la commande `spell` en français en utilisant des arbres digitaux qui partagent les préfixes et suffixes des mots d'un dictionnaire français de 200000 mots Le dictionnaire tient alors dans une mémoire de 500 kO. Son algorithme est aussi rapide et exact (non probabiliste).

```
  (* On trie *)
  tri_sélection* ();
  (* On imprime le résultat *)
  impression ();
  exit 0;;
```

```
(* Style plus Caml *)

let initialisation a =
  for i = 0 to vect_length a - 1 do
    a.(i) <- random__int 100
  done;;

let impression a =
  for i = 0 to vect_length a - 1 do
    print_int a.(i); print_string " ";
  done;
  print_newline();;

let main () =
  let a = make_vect nMax 0 in
  initialisation (a);
  tri_sélection (a);
  impression (a);
  exit 0;;
```

```
(* Tri bulle, voir page 26 *)
let tri_bulle a =
  let j = ref 0 in
  let v = ref 0 in
  for i = vect_length a - 1 downto 0 do
    for j = 1 to i do
      if a.(j-1) > a.(j) then let t = ref a.(j-1) in begin
        a.(j-1) <- a.(j); a.(j) <- !t
        end
    done
  done;;
```

```
(* Tri par insertion, voir page 28 *)
let tri_insertion a =
  let j = ref 0 in
  let v = ref 0 in
  for i = 1 to vect_length a - 1 do
    v := a.(i); j := i;
    while !j > 0 && a.(!j - 1) > !v do
      a.(!j) <- a.(!j - 1);
      decr j
    done;
    a.(!j) <- !v;
  done;;
```

```
(* Tri Shell, voir page 30 *)
let tri_shell a =
  let h = ref 1 in
   while !h <= vect_length a - 1 do
    h := 3 * !h + 1
   done;
   while !h > 1 do
     h := !h / 3;
     for i = !h to vect_length a - 1 do
       if a.(i) < a.(i - !h) then begin
         let v = a.(i) in
         let j = ref i in
         while !j >= !h && a.(!j - !h) > v do
           a.(!j) <- a.(!j - !h);
           j := !j - !h
         done;
         a.(!j) <- v
       end
     done
   done;;
```

```
(* Recherche 1, voir page 31 *)
exception Found of int;;

let recherche s bottin =
  try
    for i = 0 to vect_length bottin - 1 do
      let nom, tel = bottin.(i) in
      if nom = s then raise (Found tel)
    done;
    raise Not_found
  with Found tel -> tel;;
```

```
(* Recherche 3, voir page 31 *)
let recherche s bottin =
  nom.(vect_length nom - 1) <- (s, 0);
  let i = ref 0 in
  while fst (bottin.(!i)) <> s do
    incr i
  done;
  if !i = vect_length bottin
  then raise Not_found
  else
    let _, tel = bottin.(!i) in tel;;
```

```
exception Found of int;;

let bottin =
 [| "paul",   2811;
    "roger",  4501;
    "laure",  2701;
```

```
    "anne",   2702;
    "pierre", 2805;
    "yves",   2806
 |];;

// Encore une autre manière d'écrire la fonction recherche
let recherche s bottin = begin
  try
  do_vect
    (function nom, tel ->
       if nom = s then raise (Found tel))
    bottin;
  raise Not_found
  with Found t -> t
  end;;

(* Lecture des données *)
while true do
  printf "%d\n"
    (recherche (input_line stdin) bottin)
done;;
```

```
(* Recherche dichotomique, voir page 33 *)
let nom =
  [| "anne"; "laure"; "paul";
     "pierre"; "roger"; "yves" |];;
let tel =
  [| 2702;   2701;    2811;
     2805;     4501;    2806 |];;

let recherche_dichotomique x =
  let g = ref 0
  and d = ref (vect_length nom - 1)
  and i = ref 0 in
  while x <> nom.(!i) && !g <= !d do
    i := (!g + !d) / 2;
    if x < nom.(!i) then d := !i - 1
    else g := !i + 1;
  done;
  if x = nom.(!i) then tel.(!i) else (-1);;
```

```
(* Insertion 1, voir page 34 *)
let n = ref 0;;

let insertion x val =
  if !n >= vect_length nom
    then failwith "Débordement de la table";
  nom.(n) <- x;
  tel.(n) <- val;
  incr n;;
```

```
(* Fonction de hachage, voir page 34 *)
```

```
let N = vect_length nom - 1
and B = 128;;

let h s =
  let result = ref 0 in
  for i = 0 to string_length s - 1 do
    result :=
      (!result * B + (int_of_char (s.[i]))) mod N
  done;
  !result;;
```

```
(* Recherche avec hachage, voir page 35 *)
let col = make_vect (vect_length nom)  (-1);;

let recherche x =
  let i = ref (h x) in
  while nom.(!i) <> x &&
        col.(!i) <> -1
  do i := col.(!i) done;
  if x = nom.(!i) then tel.(!i) else -1;;
```

```
(* Insertion avec hachage, voir page 34 *)
let nom = make_vect nMax "";;
let tel = make_vect nMax 0;;
let col = make_vect nMax (-1);;

let n = ref 0;;

let insertion x val =
  let i = h x in
  if nom.(i) = "" then begin
    nom.(i) <- x;
    tel.(i) <- val
  end else begin
  if !n >= nMax
    then failwith "Débordement de la table"
    else begin
      nom.(!n) <- x;
      tel.(!n) <- val;
      col.(!n) <- col.(i);   (* On met la nouvelle entrée en tête *)
      col.(i) <- !n;         (* de la liste des collisions *)
      incr n                 (* de sa classe d'équivalence *)
    end;
  end;;
```

```
(* Hachage avec adressage ouvert, voir page 36 *)

let recherche x =
  let i = ref (h x) in
  while nom.(!i) <> x && nom.(!i) <> "" do
    i := (!i + 1) mod N
  done;
  if nom.(!i) = x then tel.(!i) else -1;;
```

```
let insertion x val =
  if !n > N
    then failwith "Débordement de la table";
  incr n;
  let i = ref (h x) in
  while nom.(!i) <> x && nom.(!i) <> "" do
    i := (!i + 1) mod N
  done;
  nom.(!i) <- x;
  tel.(!i) <- val;;
```

Chapitre 2

Récursivité

Les définitions par récurrence sont assez courantes en mathématiques. Prenons le cas de la suite de Fibonacci, définie par

$u_0 = u_1 = 1$
$u_n = u_{n-1} + u_{n-2}$ pour $n > 1$

On obtient donc la suite 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946,... Nous allons voir que ces définitions s'implémentent très simplement en informatique par les définitions récursives.

2.1 Fonctions récursives

2.1.1 Fonctions numériques

Pour calculer la suite de Fibonacci, une transcription littérale de la formule est la suivante:

```
static int fib (int n) {

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

`fib` est une fonction qui utilise son propre nom dans la définition d'elle-même. Ainsi, si l'argument n est plus petit que 1, on retourne comme valeur 1. Sinon, le résultat est `fib`$(n-1)$ + `fib`$(n-2)$. Il est donc possible en Java, comme en beaucoup d'autres langages (sauf Fortran), de définir de telles fonctions *récursives*. D'ailleurs, toute suite $\langle u_n \rangle$ définie par récurrence s'écrit de cette manière en Java, comme le confirment les exemples numériques suivants: factorielle et le triangle de Pascal.

```
static int fact(int n)  {

  if (n <= 1)
      return 1;
  else
      return n * fact (n-1);
}

static int C(int n, int p)  {

  if ((n == 0) || (p == n))
```

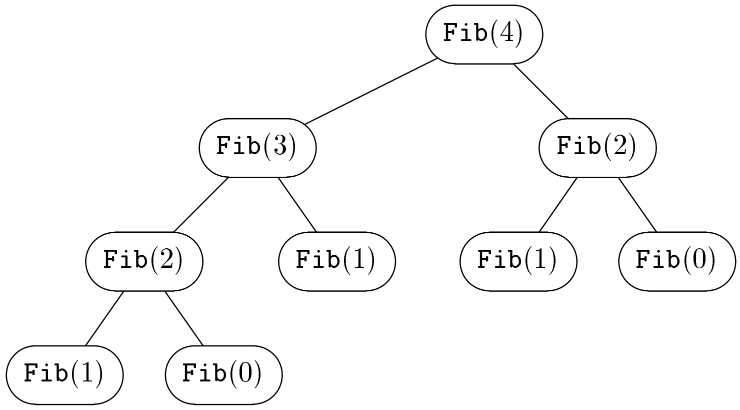


FIG. 2.1 – *Appels récursifs pour* fib(4)

```
            return 1;
        else
            return C(n-1, p-1) + C(n-1, p);
    }
```

On peut se demander comment Java s'y prend pour faire le calcul des fonctions récursives. Nous allons essayer de le suivre sur le calcul de fib(4). Rappelons nous que les arguments sont transmis par valeur dans le cas présent, et donc qu'un appel de fonction consiste à évaluer l'argument, puis à se lancer dans l'exécution de la fonction avec la valeur de l'argument. Donc

```
fib(4) -> fib (3) + fib (2)
       -> (fib (2) + fib (1)) + fib (2)
       -> ((fib (1) + fib (1)) + fib (1)) + fib(2)
       -> ((1 + fib(1)) + fib (1)) + fib(2)
       -> ((1 + 1) + fib (1)) + fib(2)
       -> (2 + fib(1)) + fib(2)
       -> (2 + 1) + fib(2)
       -> 3 + fib(2)
       -> 3 + (fib (1) + fib (1))
       -> 3 + (1 + fib(1))
       -> 3 + (1 + 1)
       -> 3 + 2
       -> 5
```

Il y a donc un bon nombre d'appels successifs à la fonction fib (9 pour fib(4)). Comptons le nombre d'appels récursifs R_n pour cette fonction. Clairement $R_0 = R_1 = 1$, et $R_n = 1 + R_{n-1} + R_{n-2}$ pour $n > 1$. En posant $R'_n = R_n + 1$, on en déduit que $R'_n = R'_{n-1} + R'_{n-2}$ pour $n > 1$, $R'_1 = R'_0 = 2$. D'où $R'_n = 2 \times \texttt{fib}(n)$, et donc le nombre d'appels récursifs R_n vaut $2 \times \texttt{fib}(n) - 1$, c'est à dire 1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, 1973, 3193, 5167, 8361, 13529, 21891,... Le nombre d'appels récursifs est donc très élevé, d'autant plus qu'il existe une méthode itérative simple en calculant simultanément $\texttt{fib}(n)$ et $\texttt{fib}(n-1)$. En effet, on a un calcul linéaire simple

$$\begin{pmatrix} \texttt{fib}(n) \\ \texttt{fib}(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \texttt{fib}(n-1) \\ \texttt{fib}(n-2) \end{pmatrix}$$

Ce qui donne le programme itératif suivant

```
static int fib(int n)  {

  int u, v;
  int u0, v0;
  int i;

  u = 1; v = 1;
  for (i = 2; i <= n; ++i) {
      u0 = u; v0 = v;
      u = u0 + v0;
      v = v0;
  }
  return u;
}
```

Pour résumer, une bonne règle est de ne pas trop essayer de suivre dans les moindres détails les appels récursifs pour comprendre le sens d'une fonction récursive. Il vaut mieux en général comprendre synthétiquement la fonction. La fonction de Fibonacci est un cas particulier car son calcul récursif est particulièrement long. Mais ce n'est pas le cas en général. Non seulement, l'écriture récursive peut se révéler efficace, mais elle est toujours plus naturelle et donc plus esthétique. Elle ne fait que suivre les définitions mathématiques par récurrence. C'est une méthode de programmation très puissante.

2.1.2 La fonction d'Ackermann

La suite de Fibonacci avait une croissance exponentielle. Il existe des fonctions récursives qui croissent encore plus rapidement. Le prototype est la fonction d'Ackermann. Au lieu de définir cette fonction mathématiquement, il est aussi simple d'en donner la définition récursive en Java

```
static int ack(int m, int 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));
}
```

On peut vérifier que

$$\begin{aligned}
\mathtt{ack}(0,n) &= n+1\\
\mathtt{ack}(1,n) &= n+2\\
\mathtt{ack}(2,n) &\simeq 2*n\\
\mathtt{ack}(3,n) &\simeq 2^n\\
\mathtt{ack}(4,n) &\simeq \left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}n
\end{aligned}$$

Donc $\mathtt{ack}(5,1) \simeq \mathtt{ack}(4,4) \simeq 2^{65536} > 10^{80}$, c'est à dire le nombre d'atomes de l'univers.

2.1.3 Récursion imbriquée

La fonction d'Ackermann contient deux appels récursifs imbriqués, c'est ce qui la fait croître si rapidement. Un autre exemple est la fonction 91 de MacCarthy

```
static int f(int n)  {

  if (n > 100)
      return n - 10;
  else
      return f(f(n+11));
}
```

Ainsi, le calcul de $\mathtt{f}(96)$ donne $\mathtt{f}(96) = \mathtt{f}(\mathtt{f}(107)) = \mathtt{f}(97) = \cdots \mathtt{f}(100) = \mathtt{f}(\mathtt{f}(111)) = \mathtt{f}(101) = 91$. On peut montrer que cette fonction vaut 91 si $n \leq 100$ et $n - 10$ si $n > 100$. Cette fonction anecdotique, qui utilise la récursivité imbriquée, est intéressante car il n'est pas du tout évident qu'une telle définition donne toujours un résultat.[1] Par exemple, la fonction de Morris suivante

```
static int g(int m, int n) {

  if (m == 0)
      return 1;
  else
      return g(m - 1, g(m, n));
}
```

Que vaut alors $\mathtt{g}(1,0)$? En effet, on a $\mathtt{g}(1,0) = \mathtt{g}(0, \mathtt{g}(1,0))$. Il faut se souvenir que Java passe les arguments des fonctions par valeur. On calcule donc toujours la valeur de l'argument avant de trouver le résultat d'une fonction. Dans le cas présent, le calcul de `g(1,0)` doit recalculer `g(1, 0)`. Et le calcul ne termine pas.

2.2 Indécidabilité de la terminaison

Les logiciens Gödel et Turing ont démontré dans les années 30 qu'il était impossible d'espérer trouver un programme sachant tester si une fonction récursive termine son calcul. L'arrêt d'un calcul est en effet *indécidable*. Dans cette section, nous montrons qu'il n'existe pas de fonction qui permette de tester si une fonction Java termine. Nous présentons cette preuve sous la forme d'une petite histoire:

Le responsable des travaux pratiques d'Informatique en a assez des programmes qui calculent indéfiniment écrits par des élèves peu expérimentés. Cela l'oblige à chaque fois à des manipulations compliquées pour stopper ces programmes. Il voit alors dans un journal spécialisé une publicité:

> *Ne laissez plus boucler vos programmes! Utilisez notre fonction* `termine(o)`. *Elle prend comme paramètre le nom d'un objet et répond* `true` *si la procédure* `f` *de cet objet ne boucle pas indéfiniment et* `false` *sinon. En n'utilisant que les procédures pour lesquelles* `termine` *répond* `true`*, vous évitez tous les problèmes de non terminaison. D'ailleurs, voici quelques exemples:*

```
class Turing {

  static boolean termine (Object o) {
```

1. Certains systèmes peuvent donner des résultats partiels, par exemple le système SYNTOX de François Bourdoncle qui arrive à traiter le cas de cette fonction

```
            // détermine la terminaison de o.f()
            boolean resultat;
            // contenu breveté par le vendeur
            return resultat;
        }
    }

    class Facile {
        void f () {
        }
    }

    class Boucle {
        void f () {
            for (int i = 1; i > 0; ++i)
                ;
        }
    }

    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));
        }
    }
```

pour lesquels `termine(o)` *répond* `true` *puis* `false`.

Impressionné par la publicité, le responsable des travaux pratiques achète à prix d'or cette petite merveille et pense que sa vie d'enseignant va être enfin tranquille. Un élève lui fait toutefois remarquer qu'il ne comprend pas l'acquisition faite par le Maître avec la classe suivante:

```
    class Absurde {
        void f () {
            while (Turing.termine(this))
                ;
         }

    class Test {
        public static void main (String args[]) {
            Absurde o3 = new Absurde();
            System.out.println (Turing.termine(o3));
        }
    }
```

Si la procédure `o3.f` boucle indéfiniment, alors `termine(o3)` = `false`. Donc la boucle `while` de `o3.f` s'arrête et `o3.f` termine. Sinon, si la procédure `o3.f` ne boucle pas indéfiniment, alors `termine(o3)` = `true`. La boucle `while` précédente boucle indéfiniment, et la procédure `o3.f` boucle indéfiniment. Il y a donc une contradiction sur les valeurs possibles de `termine(o3)`. Cette expression ne peut être définie. Ayant noté le mauvais esprit de l'Elève, le Maître conclut qu'on ne peut décidément pas faire confiance à la presse spécialisée!

L'histoire est presque vraie. Le Maître s'appelait David Hilbert et voulait montrer la validité de sa thèse par des moyens automatiques. L'Elève impertinent était Kurt Gödel. Le tout se passait vers 1930. Grâce à Gödel, on s'est rendu compte que toutes les fonctions mathématiques ne sont pas calculables par programme. Par exemple, il y a beaucoup plus de fonctions de $\mathbb{N}$ (entiers naturels) dans $\mathbb{N}$ que de programmes qui sont en quantité dénombrable. Gödel, Turing, Church et Kleene sont parmi les fondateurs de la théorie de la calculabilité.

Pour être plus précis, on peut remarquer que nous demandons beaucoup à notre fonction `termine`, puisqu'elle prend en argument un objet (en fait une "adresse mémoire"), désassemble peut-être la procédure correspondante, et décide de sa terminaison. Sinon, elle ne peut que lancer l'exécution de son argument et ne peut donc tester sa terminaison (quand il ne termine pas). Un résultat plus fort peut être montré: il n'existe pas de fonction prenant en argument le source de toute procédure (en tant que chaîne de caractères) et décidant de sa terminaison. C'est ce résultat qui est couramment appelé l'indécidabilité de l'arrêt. Mais montrer la contradiction en Java est alors beaucoup plus dur.

2.3 Procédures récursives

Les procédures, comme les fonctions, peuvent être récursives, et comporter un appel récursif. L'exemple le plus classique est celui des tours de Hanoi. On a 3 piquets en face de soi, numérotés 1, 2 et 3 de la gauche vers la droite, et n rondelles de tailles toutes différentes entourant le piquet 1, formant un cône avec la plus grosse en bas et la plus petite en haut. On veut amener toutes les rondelles du piquet 1 au piquet 3 en ne prenant qu'une seule rondelle à la fois, et en s'arrangeant pour qu'à tout moment il n'y ait jamais une rondelle sous une plus grosse. La légende dit que les bonzes passaient leur vie à Hanoi à résoudre ce problème pour $n = 64$, ce qui leur permettait d'attendre l'écroulement du temple de Brahma, et donc la fin du monde (cette légende fut inventée par le mathématicien français E. Lucas en 1883). Un raisonnement par récurrence permet de trouver la solution en quelques lignes. Si $n \leq 1$, le problème est trivial. Supposons maintenant le problème résolu pour $n-1$ rondelles pour aller du piquet i au piquet j. Alors, il y a une solution très facile pour transférer n rondelles de i en j:

1- on amène les $n-1$ rondelles du haut de i sur le troisième piquet $k = 6 - i - j$,

2- on prend la grande rondelle en bas de i et on la met toute seule en j,

3- on amène les $n-1$ rondelles de k en j.

Ceci s'écrit

```
static void hanoi(int n, int i, int j)  {

  if (n > 0) {
      hanoi (n-1, i, 6-(i+j));
      System.out.println (i + " -> " + j);
      hanoi (n-1, 6-(i+j), j);
  }
}
```

Ces quelques lignes de programme montrent bien comment en généralisant le problème, c'est-à-dire aller de tout piquet i à tout autre j, un programme récursif de quelques lignes peut résoudre un problème a priori compliqué. C'est la force de la récursion et du raisonnement par récurrence. Il y a bien d'autres exemples de programmation récursive, et la puissance de cette méthode de programmation a été étudiée dans la théorie dite de la *récursivité* qui s'est développée bien avant l'apparition de l'informatique (Kleene [25], Rogers [45]). Le mot récursivité n'a qu'un lointain rapport avec celui qui est employé ici, car il s'agissait d'établir une théorie abstraite de la calculabilité, c'est à dire de

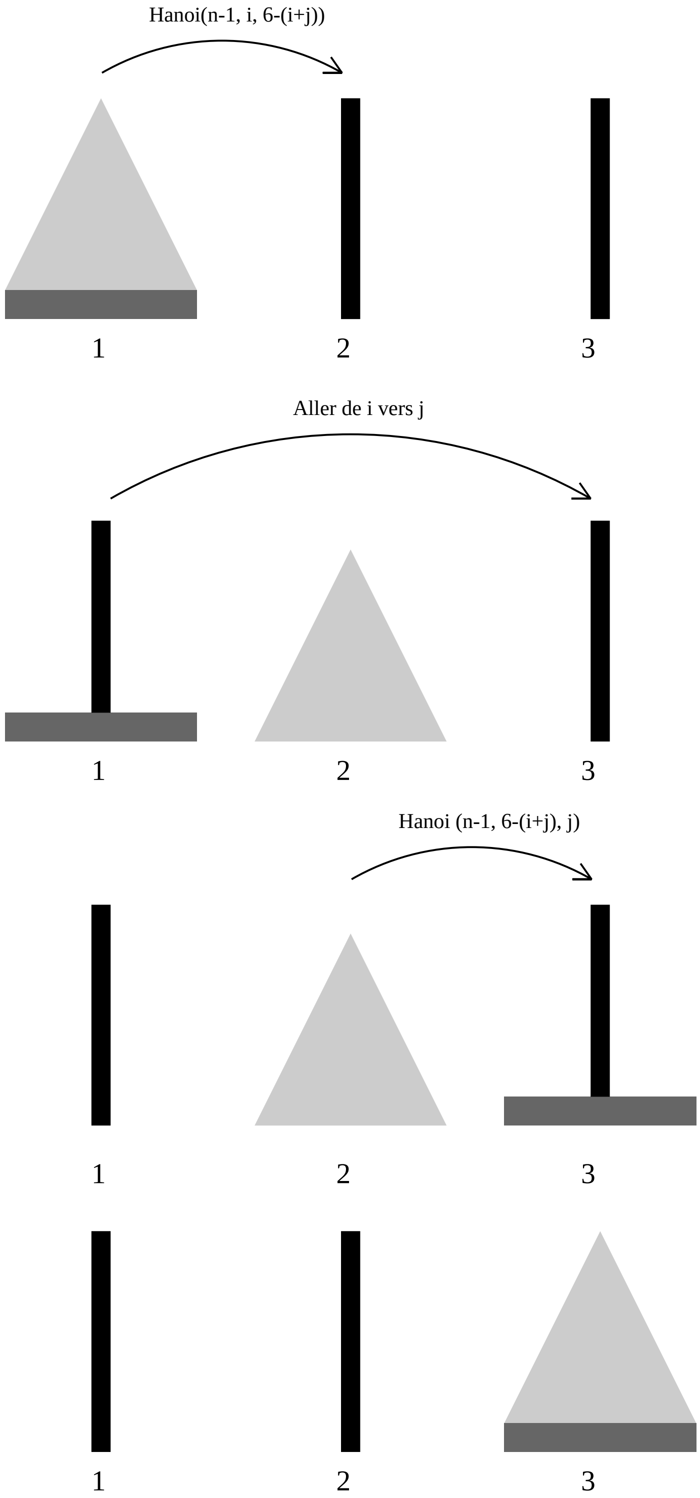


FIG. 2.2 – *Les tours de Hanoi*

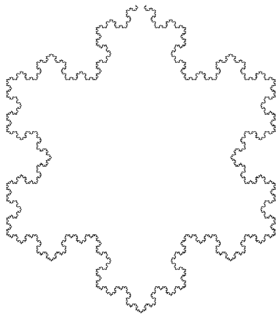


FIG. 2.3 – *Flocons de von Koch*

définir mathématiquement les objets qu'on sait calculer, et surtout ceux qu'on ne sait pas calculer. Mais l'idée initiale de la récursivité est certainement à attribuer à Kleene (1935).

2.4 Fractales

Considérons d'autres exemples de programmes récursifs. Des exemples spectaculaires sont le cas de fonctions graphiques fractales. Nous utilisons les fonctions graphiques du Macintosh (cf. page 218). Un premier exemple simple est le flocon de von Koch [11] qui est défini comme suit

> Le flocon d'ordre 0 est un triangle équilatéral.
> Le flocon d'ordre 1 est ce même triangle dont les côtés sont découpés en trois et sur lequel s'appuie un autre triangle équilatéral au milieu.
> Le flocon d'ordre $n+1$ consiste à prendre le flocon d'ordre n en appliquant la même opération sur chacun de ses côtés.

Le résultat ressemble effectivement à un flocon de neige idéalisé. L'écriture du programme est laissé en exercice. On y arrive très simplement en utilisant les fonctions trigonométriques `sin` et `cos`. Un autre exemple classique est la courbe du Dragon. La définition de cette courbe est la suivante: la courbe du Dragon d'ordre 1 est un vecteur entre deux points quelconques P et Q, la courbe du Dragon d'ordre n est la courbe du Dragon d'ordre $n-1$ entre P et R suivie de la même courbe d'ordre $n-1$ entre R et Q (à l'envers), où PRQ est le triangle isocèle rectangle en R, et R est à droite du vecteur PQ. Donc, si P et Q sont les points de coordonnées (x, y) et (z, t), les coordonnées (u, v) de R sont

$$u = (x+z)/2 + (t-y)/2$$
$$v = (y+t)/2 - (z-x)/2$$

La courbe se programme simplement par

```
static void dragon(int n, int x, int y, int z, int t) {

  int u, v;

  if (n == 1) {
      moveTo (x, y);
      lineTo (z, t);
```

FIG. 2.4 – *La courbe du Dragon*

```
  } else {
      u = (x + z + t - y) / 2;
      v = (y + t - z + x) / 2;
      dragon (n-1, x, y, u, v);
      dragon (n-1, z, t, u, v);
  }
}
```

Si on calcule `dragon` (20, 20, 20, 220, 220), on voit apparaître un petit dragon. Cette courbe est ce que l'on obtient en pliant 10 fois une feuille de papier, puis en la dépliant. Une autre remarque est que ce tracé lève le crayon, et que l'on préfère souvent ne pas lever le crayon pour la tracer. Pour ce faire, nous définissons une autre procédure `dragonBis` qui dessine la courbe à l'envers. La procédure `dragon` sera définie récursivement en fonction de `dragon` et `dragonBis`. De même, `dragonBis` est définie récursivement en termes de `dragonBis` et `dragon`. On dit alors qu'il y a une *récursivité croisée*.

```
static void dragon (int n, int x, int y, int z, int t)  {
  int u, v;

  if (n == 1) {
      moveTo (x, y);
      lineTo (z, t);
  } else {
      u = (x + z + t - y) / 2;
      v = (y + t - z + x) / 2;
      dragon (n-1, x, y, u, v);
      dragonBis (n-1, u, v, z, t);
  }
}

static void dragonBis(int n, int x, int y, int z, int t)  {
```

FIG. 2.5 – *Partition de Quicksort*

```
    int u, v;

    if (n == 1) {
        moveTo (x, y);
        lineTo (z, t);
    } else {
        u = (x + z - t + y) / 2;
        v = (y + t + z - x) / 2;
        dragon (n-1, x, y, u, v);
        dragonBis (n-1, u, v, z, t);
    }
}
```

Il y a bien d'autres courbes fractales comme la courbe de Hilbert, courbe de Peano qui recouvre un carré, les fonctions de Mandelbrot. Ces courbes servent en imagerie pour faire des parcours "aléatoires" de surfaces, et donnent des fonds esthétiques à certaines images.

2.5 Quicksort

Cette méthode de tri est due à C.A.R Hoare en 1960. Son principe est le suivant. On prend un élément au hasard dans le tableau à trier. Soit v sa valeur. On partitionne le reste du tableau en 2 zones: les éléments plus petits ou égaux à v, et les éléments plus grands ou égaux à v. Si on arrive à mettre en tête du tableau les plus petits que v et en fin du tableau les plus grands, on peut mettre v entre les deux zones à sa place définitive. On peut recommencer récursivement la procédure Quicksort sur chacune des partitions tant qu'elles ne sont pas réduites à un élément. Graphiquement, on choisit v comme l'un des a_i à trier. On partitionne le tableau pour obtenir la position de la figure 2.5 (c). Si g et d sont les bornes à gauche et à droite des indices du tableau à trier, le schéma du programme récursif est

```
static void qSort(int g, int d)  {

  if (g < d) {
      v = a[g];
      Partitionner le tableau autour de la valeur v
      et mettre v à sa bonne position m
      qSort (g, m-1);
      qSort (m+1, d);
  }
}
```

Nous avons pris `a[g]` au hasard, toute autre valeur du tableau `a` aurait convenu. En fait, la prendre vraiment au hasard ne fait pas de mal, car ça évite les problèmes pour les distributions particulières des valeurs du tableau (par exemple si le tableau est déjà trié). Plus important, il reste à écrire le fragment de programme pour faire la partition. Une méthode ingénieuse [6] consiste à parcourir le tableau de g à d en gérant deux indices i et m tels qu'à tout moment on a $a_j < v$ pour $g < j \leq m$, et $a_j \geq v$ pour $m < j < i$. Ainsi

```
m = g;
for (i = g+1; i <= d; ++i)
    if (a[i] < v) {
        ++m;
        x = a[m]; a[m] = a[i]; a[i] = x; // Echanger a_m et a_i
    }
```

ce qui donne la procédure suivante de Quicksort

```
static void qSort(int g, int d)  {

  int i, m, x, v;

  if (g < d) {
      v = a[g];
      m = g;
      for (i = g+1; i <= d; ++i)
          if (a[i] < v) {
              ++m;
              x = a[m]; a[m] = a[i]; a[i] = x; // Echanger a_m et a_i
          }
      x = a[m]; a[m] = a[g]; a[g] = x;         // Echanger a_m et a_g
      qSort (g, m-1);
      qSort (m+1, d);
  }
}
```

Cette solution n'est pas symétrique. La présentation usuelle de Quicksort consiste à encadrer la position finale de v par deux indices partant de 1 et N et qui convergent vers la position finale de v. En fait, il est très facile de se tromper en écrivant ce programme. C'est pourquoi nous avons suivi la méthode décrite dans le livre de Bentley [6]. Une méthode très efficace et symétrique est celle qui suit, de Sedgewick [47].

```
static void quickSort(int g, int d) {

  int v,t,i,j;

  if (g < d) {
      v = a[d]; i= g-1; j = d;
      do {
          do
              ++i;
          while (a[i] < v);
          do
              --j;
          while (a[j] > v);
          t = a[i]; a[i] = a[j]; a[j] = t;
      } while (j > i);
      a[j] = a[i]; a[i] = a[d]; a[d] = t;
      quickSort (g, i-1);
      quickSort (i+1, d);
  }
}
```

On peut vérifier que cette méthode ne marche que si des sentinelles à gauche et à droite du tableau existent, en mettant un plus petit élément que v à gauche et un plus grand à droite. En fait, une manière de garantir cela est de prendre toujours l'élément de gauche, de droite et du milieu, de mettre ces trois éléments dans l'ordre, en mettant le plus petit des trois en a_1, le plus grand en a_N et prendre le médian comme valeur v à placer dans le tableau `a`. On peut remarquer aussi comment le programme précédent rend bien symétrique le cas des valeurs égales à v dans le tableau. Le but recherché est d'avoir la partition la plus équilibrée possible. En effet, le calcul du nombre moyen C_N de comparaisons emprunté à [47] donne $C_0 = C_1 = 0$, et pour $N \geq 2$,

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k})$$

D'où par symétrie

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \leq k \leq N} C_{k-1}$$

En soustrayant, on obtient après simplification

$$NC_N = (N+1)C_{N-1} + 2N$$

En divisant par $N(N+1)$

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} = \frac{C_2}{3} + \sum_{3 \leq k \leq N} \frac{2}{k+1}$$

En approximant

$$\frac{C_N}{N+1} \simeq 2 \sum_{1 \leq k \leq N} \frac{1}{k} \simeq 2 \int_1^N \frac{1}{x} dx = 2 \ln N$$

D'où le résultat

$$C_N \simeq 1,38 N \log_2 N$$

Quicksort est donc très efficace en moyenne. Bien sûr, ce tri peut être en temps $O(N^2)$, si les partitions sont toujours dégénérées par exemple pour les suites monotones croissantes ou décroissantes. C'est un algorithme qui a une très petite constante (1,38) devant la fonction logarithmique. Une bonne technique consiste à prendre Quicksort pour de gros tableaux, puis revenir au tri par insertion pour les petits tableaux (≤ 8 ou 9 éléments).

Le tri par Quicksort est le prototype de la méthode de programmation *Divide and Conquer* (en français "diviser pour régner"). En effet, Quicksort divise le problème en deux (les 2 appels récursifs) et recombine les résultats grâce à la partition initialement faite autour d'un élément pivot. *Divide and Conquer* sera la méthode chaque fois qu'un problème peut être divisé en morceaux plus petits, et que l'on peut obtenir la solution à partir des résultats calculés sur chacun des morceaux. Cette méthode de programmation est très générale, et reviendra souvent dans le cours. Elle suppose donc souvent plusieurs appels récursifs et permet souvent de passer d'un nombre d'opérations linéaire $O(n)$ à un nombre d'opérations logarithmique $O(\log n)$.

2.6 Le tri par fusion

Une autre procédure récursive pour faire le tri est le *tri par fusion* (ou par interclassement). La méthode provient du tri sur bande magnétique (périphérique autrefois fort utile des ordinateurs). C'est aussi un exemple de la méthode *Divide and Conquer*. On remarque d'abord qu'il est aisé de faire l'interclassement entre deux suites de nombres triés dans l'ordre croissant. En effet, soient $\langle a_1, a_2, \ldots a_M \rangle$ et $\langle b_1, b_2, \ldots b_N \rangle$ ces deux suites. Pour obtenir, la suite interclassée $\langle c_1, c_2, \ldots c_{M+N} \rangle$ des a_i et b_j, il suffit de faire le programme suivant. (On suppose que l'on met deux sentinelles $a_{M+1} = \infty$ et $b_{N+1} = \infty$ pour ne pas compliquer la structure du programme.)

```
int[] a = new int[M+1],
      b = new int[N+1],
      c = new int[M+N];

int i = 0, j = 0;

a[M+1] = b[N+1] = Integer.MAX_VALUE;

for (int k = 0; k < M+N; ++k)
    if (a[i] <= b[j]) {
         c[k] = a[i];
         ++i;
     } else {
         c[k] = b[j];
         ++j;
     }
```

Successivement, c_k devient le minimum de a_i et b_j en décalant l'endroit où l'on se trouve dans la suite a ou b selon le cas choisi. L'interclassement de M et N éléments se fait donc en $O(M + N)$ opérations. Pour faire le tri fusion, en appliquant *Divide and Conquer*, on trie les deux moitiés de la suite $\langle a_1, a_2, \ldots a_N \rangle$ à trier, et on interclasse les deux moitiés triées. Il y a toutefois une difficulté car on doit copier dans un tableau annexe les 2 moitiés à trier, puisqu'on ne sait pas faire l'interclassement en place. Si g et d sont les bornes à gauche et à droite des indices du tableau à trier, le tri fusion est donc

```
static void TriFusion(int g, int d)   {

  int   i, j, k, m;

  if (g < d) {
      m = (g + d) / 2;
      TriFusion (g, m);
      TriFusion (m + 1, d);
      for (i = m; i >= g; --i)
           b[i] = a[i];
      for (j = m+1; j <= d; ++j)
           b[d+m+1-j] = a[j];
      i = g; j = d;
      for (k = g; k <= d; ++k)
          if (b[i] < b[j]) {
              a[k] = b[i]; ++i;
          } else {
              a[k] = b[j]; --j;
          }
  }
}
```

La recopie pour faire l'interclassement se fait dans un tableau `b` de même taille que `a`. Il y a une petite astuce en recopiant une des deux moitiés dans l'ordre inverse, ce qui permet de se passer de sentinelles pour l'interclassement, puisque chaque moitié sert de sentinelle pour l'autre moitié. Le tri par fusion a une très grande vertu. Son nombre d'opérations C_N est tel que $C_N = 2N + 2C_{N/2}$, et donc $C_N = O(N \log N)$. Donc le tri fusion est un tri qui garantit un temps $N \log N$, au prix d'un tableau annexe de N éléments. Ce temps est réalisé quelle que soit la distribution des données, à la différence de QuickSort. Plusieurs problèmes se posent immédiatement: peut on faire mieux? Faut-il utiliser ce tri plutôt que QuickSort?

Nous répondrons plus tard “non” à la première question, voir section 1. Quant à la deuxième question, on peut remarquer que si ce tri garantit un bon temps, la constante petite devant $N \log N$ de QuickSort fait que ce dernier est souvent meilleur. Aussi, QuickSort utilise moins de mémoire annexe, puisque le tri fusion demande un tableau qui est aussi important que celui à trier. Enfin, on peut remarquer qu'il existe une version itérative du tri par fusion en commençant par trier des sous-suites de longueur 2, puis de longueur 4, 8, 16, ...

2.7 Programmes en Caml

```
(* Fibonaci, voir page 45 *)
let rec fib n =
  if n <= 1 then 1
  else fib (n - 1) + fib ( n - 2);;
```

```
(* Factorielle, voir page 45 *)
let rec fact n =
  if n <= 1 then 1
  else n * fact (n - 1);;
```

```
(* Triangle de Pascal, voir page 45 *)
let rec C n p =
 if n = 0 || p = n then 1
 else C (n - 1) (p - 1) + C (n - 1) p;;
```

```
(* Fibonaci itératif, voir page 46 *)
let fib n =
  let u = ref 1 and v = ref 1 in
  for i = 2 to n do
    let u0 = !u and v0 = !v in
    u := u0 + v0;
    v := u0
  done;
  !u;;

(* Le même en style fonctionnel *)
let fib n =
  let rec fib_aux k u v =
    if k > n then u
    else fib_aux (k + 1) (u + v) u in
  fib_aux 2 1 1;;
```

```
(* La fonction d'Ackermann, voir page 47 *)
let rec ack m n =
  if m = 0 then n + 1 else
  if n = 0 then ack (m - 1) 1 else
```

```
  ack (m - 1) (ack m (n - 1));;
```

```
(* La fonction 91, voir page 48 *)
let rec f91 n =
  if n > 100 then n - 10
  else f91 (f91 (n + 11));;
```

```
(* La fonction de Morris, voir page 48 *)
let rec g m n =
 if m = 0 then 1
 else g (m - 1) (g m n);;
```

```
(* Les tours de Hanoi, voir page 50 *)
let rec hanoi n i j =
   if n > 0 then begin
     hanoi (n - 1) i (6 - (i + j));
     printf "%d -> %d\n" i j;
     hanoi (n - 1) (6 - (i + j)) j;
   end;;
```

```
#open "graphics";;
open_graph "";;

let rec dragon n x y z t =
  if n = 1 then begin
    moveto x y;
    lineto z t
  end else begin
    let u = (x + z + t - y) / 2
    and v = (y + t - z + x) / 2 in
    dragon (n - 1) x y u v;
    dragon (n - 1) z t u v
  end;;
```

```
(* La courbe du dragon, voir page 53 *)
let rec dragon n x y z t =
  if n = 1 then begin
    moveto x y;
    lineto z t
  end else begin
    let u = (x + z + t - y) / 2
    and v = (y + t - z + x) / 2 in
    dragon (n - 1) x y u v;
    dragon_bis (n - 1) u v z t
  end

and dragon_bis n x y z t =
  if n = 1 then begin
    moveto x y;
```

```
    lineto z t
  end else begin
    let u = (x + z - t + y) / 2
    and v = (y + t + z - x) / 2 in
    dragon (n - 1) x y u v;
    dragon_bis (n - 1) u v z t
  end;;

clear_graph();;
dragon 15 120 120 50 50;;
```

```
(* Quicksort, voir page 55 *)

let rec qsort g d =
  if g < d then begin
    let v = a.(g)
    and m = ref g in
    for i = g + 1 to d do
      if a.(i) < v then begin
        m := !m + 1;
        let x = a.(!m) in
        a.(!m) <- a.(i); a.(i) <- x
      end
    done;
    let x = a.(!m) in
    a.(!m) <- a.(g); a.(g) <- x;
    qsort g (!m - 1);
    qsort (!m + 1) d
  end;;
```

```
(* Quicksort, voir page 55 *)
let rec quicksort g d =
  if g < d then begin
    let v = a.(d)
    and t = ref 0
    and i = ref (g - 1)
    and j = ref d in
    while j > i do
      incr i;
      while a.(!i) < v do incr i done;
      decr j;
      while a.(!j) > v do decr j done;
      t := a.(!i);
      a.(!i) <- a.(!j);
      a.(!j) <- !t
    done;
    a.(!j) <- a.(!i);
    a.(!i) <- a.(d);
    a.(d) <- !t;
    quicksort g (!i - 1);
    quicksort (!i + 1) d
  end;;
```

```
(* Tri fusion, voir page 57 *)

let b = make_vect n 0;;

let rec tri_fusion g d =
  if g < d then begin
    let m = (g + d) / 2 in
    tri_fusion g m;
    tri_fusion (m + 1) d;
    for i = m downto g do
     b.(i) <- a.(i) done;
    for i = m + 1 to d do
     b.(d + m + 1 - i) <- a.(i) done;
    let i = ref g and j = ref d in
    for k = g to d do
      if b.(!i) < b.(!j) then begin
        a.(k) <- b.(!i); incr i
      end else begin
        a.(k) <- b.(!j); decr j
     end
    done
  end;;
```

Chapitre 3

Structures de données élémentaires

Dans ce chapitre, nous introduisons quelques structures utilisées de façon très intensive en programmation. Leur but est de gérer un ensemble fini d'éléments dont le nombre n'est pas fixé *a priori*. Les éléments de cet ensemble peuvent être de différentes sortes: nombres entiers ou réels, chaînes de caractères, ou des objets informatiques plus complexes comme les identificateurs de processus ou les expressions de formules en cours de calcul ... On ne s'intéressera pas aux éléments de l'ensemble en question mais aux opérations que l'on effectue sur cet ensemble, indépendamment de la nature de ses éléments. Ainsi les ensembles que l'on utilise en programmation, contrairement à ceux considérés en mathématiques qui sont fixés une fois pour toutes, sont des objets dynamiques. Le nombre de leurs éléments varie au cours de l'exécution du programme, puisqu'on peut y ajouter et supprimer des éléments en cours de traitement. Plus précisément les opérations que l'on s'autorise sur les ensembles sont les suivantes :

- *tester* si l'ensemble E est vide.
- *ajouter* l'élément x à l'ensemble E.
- *vérifier* si l'élément x appartient à l'ensemble E.
- *supprimer* l'élément x de l'ensemble E.

Cette gestion des ensembles doit, pour être efficace, répondre au mieux à deux critères parfois contradictoires: un minimum de place mémoire utilisée et un minimum d'instructions élémentaires pour réaliser une opération. La place mémoire utilisée devrait pour bien faire être très voisine du nombre d'éléments de l'ensemble E, multipliée par leur taille; c'est ce qui se passera pour les trois structures que l'on va étudier plus loin. En ce qui concerne la minimisation du nombre d'instructions élémentaires, on peut tester très simplement si un ensemble est vide et on peut réaliser l'opération d'ajout en quelques instructions, toutefois il est impossible de réaliser une suppression ou une recherche d'un élément quelconque dans un ensemble en utilisant un nombre d'opérations indépendant du cardinal de cet ensemble (à moins d'utiliser une structure demandant une très grande place en mémoire). Pour améliorer l'efficacité, on considère des structures de données dans lesquelles on restreint la portée des opérations de recherche et de suppression d'un élément en se limitant à la réalisation de ces opérations sur le dernier ou le premier élément de l'ensemble, ceci donne les structures de pile ou de file, nous verrons que malgré ces restrictions les structures en question ont de nombreuses applications.

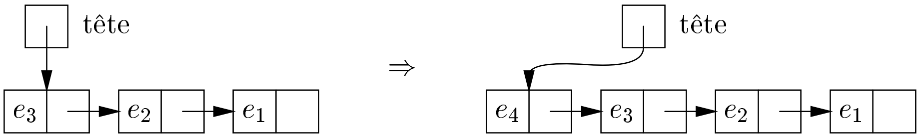


FIG. 3.1 – *Ajout d'un élément dans une liste*

3.1 Listes chaînées

La *liste* est une structure de base de la programmation, le langage LISP (*LISt Processing*), conçu par John MacCarthy en 1960, ou sa version plus récente Scheme [1], utilise principalement cette structure qui se révèle utile pour le calcul symbolique. Dans ce qui suit on utilise la liste pour représenter un ensemble d'éléments. Chaque élément est contenu dans une *cellule*, celle ci contient en plus de l'élément l'adresse de la cellule suivante, appelée aussi *pointeur*. La recherche d'un élément dans la liste s'apparente à un classique "jeu de piste" dont le but est de retrouver un objet caché: on commence par avoir des informations sur un lieu où pourrait se trouver cet objet, en ce lieu on découvre des informations sur un autre lieu où il risque de se trouver et ainsi de suite. Le langage Java permet cette réalisation à l'aide de classes et d'objets: les cellules sont des objets (c'est à dire des instances d'une classe) dont un des champs contient une réference vers la cellule suivante. La référence vers la première cellule est elle contenue dans une variable de tête de liste. Les déclarations correspondantes sont les suivantes, où l'on suppose ici que les éléments stockés dans chaque cellule dont de type entier..

```
class Liste {

  int contenu;
  Liste suivant;

  Liste (int x, Liste a) {
      contenu = x;
      suivant = a;
  }
}
```

L'instruction `new Liste (x, a)` construit une nouvelle cellule dont les champs sont `x` et `a`. La fonction `Liste(x,a)` est un constructeur de la classe `Liste` (un constructeur est une fonction non statique qui se distingue par le type de son résultat qui est celui de la classe courante et par son absence de nom pour l'identifier). L'objet `null` appartient à toute classe, et représentera dans le cas des listes le marqueur de fin de liste. Ainsi `new Liste(2, new Liste (7, new Liste (11, null)))` représentera la liste 2,7,11. Les opérations sur les ensembles que nous avons considérées ci-dessus s'expriment alors comme suit si on gère ceux-ci par des listes:

```
static boolean estVide (Liste a) {

  return a == null;
}
```

La procédure `ajouter` insère l'élément `x` en tête de liste. Ce choix de mettre l'élément en tête est indispensable pour que le nombre d'opérations lors de l'ajout soit indépendant de la taille de la liste; il suffit alors de modifier la valeur de la tête de liste ce qui est fait simplement par

```
static Liste ajouter (int x, Liste a) {

  return new Liste (x, a);   // L'ancienne tête se retrouve après x
```

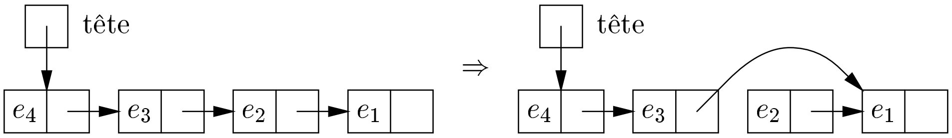


FIG. 3.2 – *Suppression d'un élément dans une liste*

```
}
```

La fonction `recherche`, qui vérifie si l'élément `x` figure bien dans la liste `a`, effectue un parcours de la liste pour rechercher l'élément. La variable `a` est modifiée itérativement par `a = a.suivant` de façon à parcourir tous les éléments jusqu'à ce que l'on trouve `x` ou que l'on arrive à la fin de la liste (`a = null`).

```
static boolean recherche (int x, Liste a) {

  while (a != null) {
      if (a.contenu == x)
          return true;
      a = a.suivant;
  }
  return false;
}
```

La fonction `recherche` peut aussi être écrite de manière récursive

```
static boolean recherche (int x, Liste a) {

  if (a == null)
      return false;
  else if (a.contenu == x)
      return true;
  else
      return recherche (x, a.suivant);
}
```

ou encore

```
static boolean recherche (int x, Liste a) {

  if (a == null)
      return false;
  else
      return (a.contenu == x) || recherche (x, a.suivant);
}
```

ou encore encore (quoique non recommandé):

```
static boolean recherche (int x, Liste a) {

  return a != null && (a.contenu == x || recherche (x, a.suivant));
}
```

Cette écriture récursive est systématique. pour les fonctions sur les listes. En effet, le type des listes vérifie l'équation suivante:

$$\boxed{\texttt{Liste} = \{\text{Liste\_vide}\} \ \uplus \ \texttt{Element} \times \texttt{Liste}}$$

où $\uplus$ est l'union disjointe et $\times$ le produit cartésien. Toute procédure ou fonction travaillant sur les listes peut donc s'écrire récursivement sur la structure de sa liste argument. Par exemple, la longueur d'une liste se calcule par

```
static int longueur(Liste a) {

  if (a == null)
      return 0;
  else
      return 1 + longueur (a.suivant);
}
```

ce qui est plus élégant que l'écriture itérative qui suit

```
static int longueurI(Liste a) {

  int   longueur = 0;
  while (a != null) {
      ++longueur;
      a = a.suivant;
  }
  return longueur;
}
```

Choisir entre la manière récursive ou itérative est affaire de goût. Autrefois, on disait que l'écriture itérative était plus efficace car utilisant moins de mémoire. Grâce aux techniques nouvelles de compilation (comme l'élimination des récursions terminales), c'est de moins en moins vrai; de plus l'occupation mémoire est, dans les ordinateurs d'aujourd'hui, une question nettement moins critique.

La suppression de la cellule qui contient `x` s'effectue en modifiant la valeur de `suivant` contenue dans le prédécesseur de `x`: le successeur du prédécesseur de `x` devient le successeur de `x`. Un traitement particulier doit être fait si l'élément à supprimer est le premier élément de la liste. La procédure récursive de suppression est très compacte dans sa définition.

```
static Liste supprimer (int x, Liste a) {

  if (a != null)
      if (a.contenu == x)
          a = a.suivant;
      else
          a.suivant = supprimer (x, a.suivant);
  return a;
}
```

Une procédure itérative demande beaucoup plus d'attention.

```
static Liste supprimer (int x, Liste a) {

  if (a != null)
      if (a.contenu == x)
          a = a.suivant;
      else {
          Liste b = a ;
          while (b.suivant != null && b.suivant.contenu != x)
              b = b.suivant;
          if (b.suivant != null)
              b.suivant = b.suivant.suivant;
```

```
        }
    return a;
  }
```

Noter que dans les deux fonctions ci-dessus, on a modifié la liste `a`, on ne peut donc disposer de deux listes distinctes l'une qui contient `x` et l'autre identique à la précédente, mais ne contenant pas `x`. Pour ce faire, il faut recopier une partie de la liste `a` dans une nouvelle liste, comme le fait le programme suivant, où il y a une utilisation un peu plus importante de la mémoire. Mais l'espace mémoire perdu est récupéré par le glaneur de cellules (GC) de Java, si personne n'utilise l'ancienne liste `a`. Avec les techniques actuelles de GC à générations, cette récupération s'effectuera très rapidement. Il faut donc noter la différence avec un langage comme Pascal ou C, où on doit se préoccuper de l'espace mémoire perdu, si on veut pouvoir le réutiliser.

```
static Liste supprimer (int x, Liste a) {

  if (a != null)
      return null;
  else if (a.contenu == x)
      return a.suivant;
  else
      return new Liste (a.contenu, supprimer (x, a.suivant));
}
```

Une technique souvent utilisée permet d'éviter quelques tests pour supprimer un élément dans une liste et plus généralement pour simplifier la programmation sur les listes. Elle consiste à utiliser une *garde* permettant de rendre homogène le traitement de la liste vide et des autres listes. En effet dans la représentation précédente, la liste vide n'a pas la même structure que les autres listes. On utilise une cellule placée au début et n'ayant pas d'information significative dans le champ `contenu`; l'adresse de la vraie première cellule se trouve dans le champ `suivant` de cette cellule. On obtient ainsi une liste *gardée*, l'avantage d'une telle garde est que la liste vide contient au moins la garde, et que par conséquent un certain nombre de programmes, qui devaient faire un cas spécial dans le cas de la liste vide ou du premier élément de liste, deviennent plus simples. Cette notion est un peu l'équivalent des sentinelles pour les tableaux. On utilisera cette technique dans les procédures sur les files un peu plus loin (voir page 71) . On peut aussi définir des listes circulaires gardées en mettant l'adresse de cette première cellule dans le champ `suivant` de la dernière cellule de la liste. Les listes peuvent aussi être gérées par différents autres mécanismes que nous ne donnons pas en détail ici. On peut utiliser, par exemple, des listes doublement chaînées dans lesquelles chaque cellule contient un élément et les adresses à la fois de la cellule qui la précède et de celle qui la suit. Des couples de tableaux peuvent aussi être utilisés, le premier `contenu` contient les éléments de l'ensemble, le second `adrsuivant` contient les adresses de l'élément suivant dans le tableau `contenu`.

Remarque La procédure *ajouter* effectue 3 opérations élémentaires. Elle est donc très efficace. En revanche, les procédures *recherche* et *supprimer* sont plus longues puisqu'on peut aller jusqu'à parcourir la totalité d'une liste pour retrouver un élément. On peut estimer, si on ne fait aucune hypothèse sur la fréquence respective des recherches, que le nombre d'opérations est en moyenne égal à la moitié du nombre d'éléments de la liste. Ceci est à comparer à la recherche dichotomique qui effectue un nombre logarithmique de comparaisons et à la recherche par hachage qui est souvent bien plus rapide encore.

Exemple A titre d'exemple d'utilisation des listes, nous considérons la construction d'une liste des nombres premiers inférieurs ou égaux à un entier n donné. Pour construire cette liste, on commence, dans une première phase, par y ajouter tous les entiers de 2 à n en commençant par le plus grand et en terminant par le plus petit. Du fait de l'algorithme d'ajout décrit plus haut, la liste contiendra donc les nombres en ordre croissant. On utilise ensuite la méthode classique du crible d'Eratosthène: on considère successivement les éléments de la liste dans l'ordre croissant et on supprime tous leurs multiples stricts. Ceci se traduit par la procédure suivante :

```
static Liste listePremier (int n) {

  Liste  a = null;
  int    k;

  for (int i = n; i >= 2; --i) {
       a = ajouter (i, a);
  }
  k = a.contenu;
  for (Liste b = a; k * k <= n ; b = b.suivant){
      k = b.contenu;
      for (int j = k; j <= n/k; ++j)
          a = supprimer (j * k, a);
      }
  return(a);
}
```

Remarque Nous ne prétendons pas que cette programmation soit efficace (loin de là!). Elle est toutefois simple à écrire, une fois que l'on a à sa disposition les fonctions sur les listes. Elle donne de bons résultats pour n inférieur à 10000. Un bon exercice consiste à en améliorer l'efficacité.

Exemple Un autre exemple, plus utile, d'application des listes est la gestion des collisions dans le hachage dont il a été question au chapitre 1 (voir page 36). Il s'agit pour chaque entier i de l'intervalle $[0 \dots N-1]$ de construire une liste chaînée L_i formée de toutes les clés x telles que $h(x) = i$. Les éléments de la liste sont des entiers permettant d'accéder aux tableaux des noms et des numéros de téléphone. Les procédures de recherche et d'insertion de x dans une table deviennent des procédures de recherche et d'ajout dans une liste. Ainsi, si la fonction h est mal choisie, le nombre de collisions devient important, la plus grande des listes devient de taille imposante et le hachage risque de devenir aussi coûteux que la recherche dans une liste chaînée ordinaire. Par contre, si h est bien choisie, la recherche est rapide.

```
Liste al[] = new Liste[N-1];

static void insertion (String x, int val) {

    int i = h(x);
    al[i] = ajouter (x, al[i]);
}

static int recherche (String x) {

  for (int a = al[h(x)]; a != null; a = a.suivant) {
      if (x.equals(nom[a.contenu]))
          return tel[a.contenu];
  }
  return -1;
}
```

3.2 Files

Les files sont utilisées en programmation pour gérer des objets qui sont en attente d'un traitement ultérieur, par exemple des processus en attente d'une ressource du système, des sommets d'un graphe, des nombres entiers en cours d'examen de certaines de leur propriétés, etc ... Dans une file les éléments sont systématiquement ajoutés en queue et supprimés en tête, la valeur d'une file est par convention celle de l'élément de tête. En anglais, on parle de stratégie FIFO *First In First Out*, par opposition à la stratégie LIFO *Last In First Out* des piles. De façon plus formelle, on se donne un ensemble E, l'ensemble des files dont les éléments sont dans E est noté $Fil(E)$, la file vide (qui ne contient aucun élément) est F_0, les opérations sur les files sont *vide, ajouter, valeur, supprimer*:

- *estVide* est une application de $Fil(E)$ dans (*vrai, faux*), $estVide(F)$ est égal à *vrai* si et seulement si la file F est vide.
- *ajouter* est une application de $E \times Fil(E)$ dans $Fil(E)$, $ajouter(x, F)$ est la file obtenue à partir de la file F en insérant l'élément x à la fin de celle-ci.
- *valeur* est une application de $Fil(E) \setminus F_0$ dans E qui à une file F non vide associe l'élément se trouvant en tête.
- *supprimer* est une application de $Fil(E) \setminus F_0$ dans $Fil(E)$ qui associe à une file F non vide la file obtenue à partir de F en supprimant son premier élément.

Les opérations sur les files satisfont les relations suivantes

Pour $F \neq F_0$

$$supprimer(ajouter(x, F)) = ajouter(x, supprimer(F))$$
$$valeur(ajouter(x, F)) = valeur(F)$$

Pour toute file F

$$estVide(ajouter(x, F)) = faux$$

Pour la file F_0

$$supprimer(ajouter(x, F_0)) = F_0$$
$$valeur(ajouter(x, F_0)) = x$$
$$estVide(F_0) = vrai$$

Une première idée de réalisation sous forme de programmes des opérations sur les files est empruntée à une technique mise en oeuvre dans des lieux où des *clients* font la queue pour être *servis*, il s'agit par exemple de certains guichets de réservation dans les gares, de bureaux de certaines administrations, ou des étals de certains supermarchés. Chaque client qui se présente obtient un numéro et les clients sont ensuite appelés par les serveurs du guichet en fonction croissante de leur numéro d'arrivée. Pour gérer ce système deux nombres doivent être connus par les gestionnaires: le numéro obtenu par le dernier client arrivé et le numéro du dernier client servi. On note ces deux nombres par *fin* et *début* respectivement et on gère le système de la façon suivante

- la file d'attente est vide si et seulement si *début* = *fin*,
- lorsqu'un nouveau client arrive on incrémente *fin* et on donne ce numéro au client,
- lorsque le serveur est libre et peut servir un autre client, si la file n'est pas vide, il incrémente *début* et appelle le possesseur de ce numéro.

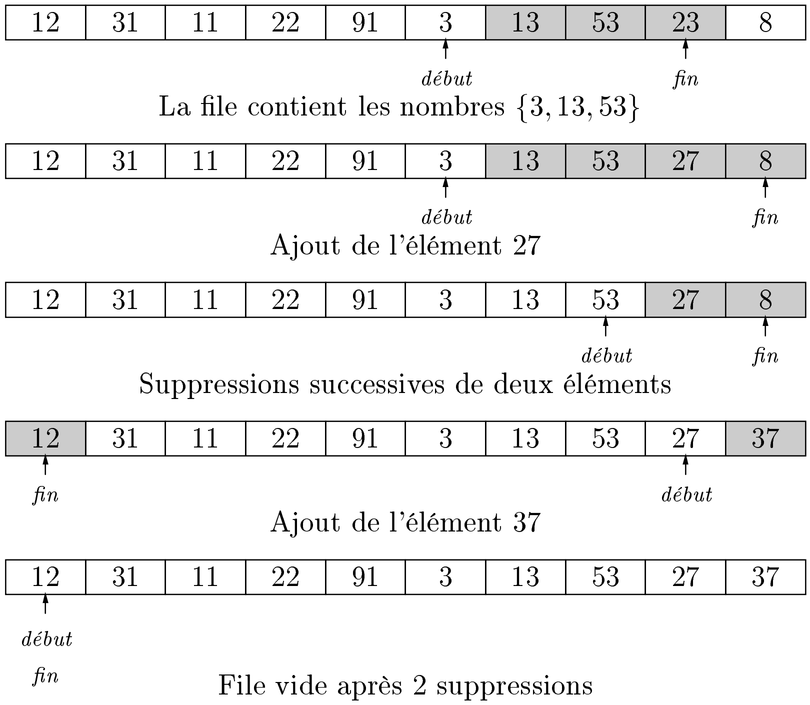


FIG. 3.3 – *File gérée par un tableau circulaire*

Dans la suite, on a représenté toutes ces opérations en Java en optimisant la place prise par la file en utilisant la technique suivante: on réattribue le numéro 0 à un nouveau client lorsque l'on atteint un certain seuil pour la valeur de *fin*. On dit qu'on a un tableau (ou tampon) circulaire.

```
class File {

  final static int MaxF = 100;

  int         debut;
  int         fin;
  boolean     pleine, vide;
  int         contenu[];

  File () {
    debut = 0; fin = 0;
    pleine = false; vide = true;
    contenu = new int[MaxF];
  }

  static File vide () {
    return new File();
  }

  static void faireVide (File f) {
    f.debut = 0; f.fin = 0;
    f.pleine = false; f.vide = true;
  }

  static boolean estVide(File f) {
```

```
    return  f.vide;
  }

  static boolean estPleine(File f) {
    return  f.pleine;
  }

  static int valeur (File f) {
    if (f.vide)
        erreur ("File Vide.");
    return  f.contenu[f.debut];
  }

  private static int Successeur(int i) {
    return  (i+1) % MaxF;
  }

  static void ajouter (int x, File f) {
    if (f.pleine)
        erreur ("File Pleine.");
    f.contenu[f.fin] = x;
    f.fin = Successeur(f.fin);
    f.vide = false;
    f.pleine = f.fin == f.debut;
  }

  static void supprimer (File f) {
    if (f.vide)
        erreur ("File Vide.");
    f.debut =  Successeur(f.debut);
    f.vide = f.fin == f.debut;
    f.pleine = false;
  }
}
```

Une autre façon de gérer des files consiste à utiliser des listes chaînées gardées (voir page 67) dans lesquelles on connaît à la fois l'adresse du premier et du dernier élément. Cela donne les opérations suivantes:

```
class File {

    Liste debut;
    Liste fin;

  File (Liste a, Liste b)  {
    debut = a;
    fin = b;
  }

  static File vide () {
    Liste garde = new Liste();
    return new File (garde, garde);
  }

  static void faireVide (File f) {
    Liste garde = new Liste();
    f.debut = f.fin = garde;
  }
```

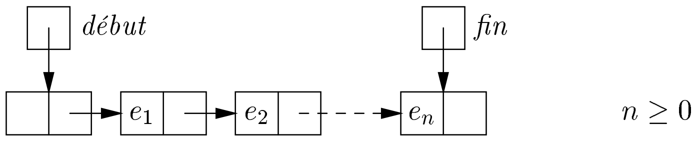


FIG. 3.4 – *File d'attente implémentée par une liste*

```
  static boolean estVide (File f) {
    return f.debut == f.fin;
  }

  static int valeur (File f) {
    Liste b = f.debut.suivant;
    return b.contenu;
  }

  static void ajouter (int x, File f) {
    Liste a = new Liste (x, null);
    f.fin.suivant = a;
    f.fin = a;
  }

  static void supprimer (File f) {
    if (estVide (f))
        erreur ("File Vide.");
    f.debut = f.debut.suivant;
  }
}
```

Nous avons deux réalisations possibles des files avec des tableaux ou des listes chaînées. L'écriture de programmes consiste à faire de tels choix pour représenter les structures de données. L'ensemble des fonctions sur les files peut être indistinctement un *module* manipulant des tableaux ou un module manipulant des listes. L'utilisation des files se fait uniquement à travers les fonctions *vide*, *ajouter*, *valeur*, *supprimer*. C'est donc l'*interface* des files qui importe dans de plus gros programmes, et non leurs réalisations. Les notions d'interface et de modules seront développées au chapitre 7.

3.3 Piles

La notion de pile intervient couramment en programmation, son rôle principal consiste à implémenter les appels de procédures. Nous n'entrerons pas dans ce sujet, plutôt technique, dans ce chapitre. Nous montrerons le fonctionnement d'une pile à l'aide d'exemples choisis dans l'évaluation d'expressions Lisp.

On peut imaginer une pile comme une boîte dans laquelle on place des objets et de laquelle on les retire dans un ordre inverse de celui dans lequel on les a mis: les objets sont les uns sur les autres dans la boîte et on ne peut accéder qu'à l'objet situé au "sommet de la pile". De façon plus formelle, on se donne un ensemble E, l'ensemble des piles dont les éléments sont dans E est noté $Pil(E)$, la pile vide (qui ne contient aucun élément) est P_0, les opérations sur les piles sont *vide, ajouter, valeur, supprimer* comme sur les files. Cette fois, les relations satisfaites sont les suivantes (où P_0 dénote la pile vide)

$supprimer(ajouter(x, P)) = P$
$estVide(ajouter(x, P)) = faux$
$valeur(ajouter(x, P)) = x$
$estVide(P_0) = vrai$

A l'aide de ces relations, on peut exprimer toute expression sur les piles faisant intervenir les 4 opérations précédentes à l'aide de la seule opération *ajouter* en partant de la pile P_0. Ainsi l'expression suivante concerne les piles sur l'ensemble des nombres entiers:

$$
\begin{aligned}
&supprimer\ (ajouter\ (7,\\
&\qquad supprimer\ (ajouter\ (valeur\ (ajouter\ (5,\ ajouter\ (3, P_0)))),\\
&\qquad\qquad ajouter\ (9, P_0))))
\end{aligned}
$$

Elle peut se simplifier en:

$ajouter(9, P_0)$

La réalisation des opérations sur les piles peut s'effectuer en utilisant un tableau qui contient les éléments et un indice qui indiquera la position du sommet de la pile. par référence.

```
class Pile {

  final static int maxP = 100;

  int          hauteur ;
  Element      contenu[];

  Pile ()  {
     hauteur = 0;
     contenu = new Element[maxP];
  }

  static File vide () {
    return new Pile();
  }

  static void faireVide (Pile p) {
    p.hauteur = 0;
  }

  static boolean estVide (Pile p) {
    return p.hauteur == 0;
  }

  static boolean estPleine (Pile p) {
    return p.hauteur == maxP;
  }

  static void ajouter (Element x, Pile p)
      throws ExceptionPile
  {
    if (estPleine (p))
         throw new ExceptionPile("pleine");
    p.contenu[p.hauteur] = x;
    ++ p.hauteur;
  }
```

```
    static Element valeur (Pile p)
        throws ExceptionPile
    {
      if (estVide (p))
          throw new ExceptionPile("vide");
      return p.contenu [p.hauteur - 1];
     }

    static void supprimer (Pile p)
        throws ExceptionPile
    {
      if (estVide (p))
          throw new ExceptionPile ("vide");
      -- p.hauteur;
    }
  }
```

où on suppose qu'une nouvelle classe définit les exceptions `ExceptionPile`:

```
  class ExceptionPile extends Exception {
    String nom;

    public ExceptionPile (String x) {
        nom = x;
    }
  }
```

Remarques Chacune des opérations sur les piles demande un très petit nombre d'opérations élémentaires et ce nombre est indépendant du nombre d'éléments contenus dans la pile. On peut gérer aussi une pile avec une liste chaînée, les fonctions correspondantes sont laissées à titre d'exercice. Les piles ont été considérées comme des arguments par référence pour éviter qu'un appel de fonction ne fasse une copie inutile pour passer l'argument par valeur.

3.4 Evaluation des expressions arithmétiques préfixées

Dans cette section, on illustre l'utilisation des piles par un programme d'évaluation d'expressions arithmétiques écrites de façon particulière. Rappelons qu'expression arithmétique signifie dans le cadre de la programmation: expression faisant intervenir des nombres, des variables et des opérations arithmétiques (par exemple: $+ * / - \sqrt{\ }$). Dans ce qui suit, pour simplifier, nous nous limiterons aux opérations binaires $+$ et $*$ et aux nombres naturels. La généralisation à des opérations binaires supplémentaires comme la division et la soustraction est particulièrement simple, c'est un peu plus difficile de considérer aussi des opérations agissant sur un seul argument comme la racine carrée, cette généralisation est laissée à titre d'exercice au lecteur. Nous ne considérerons aussi que les entiers naturels en raison de la confusion qu'il pourrait y avoir entre le symbole de la soustraction et le signe moins.

Sur certaines machines à calculer de poche, les calculs s'effectuent en mettant le symbole d'opération après les nombres sur lesquels on effectue l'opération. On a alors une notation dite *postfixée*. Dans certains langages de programmation, c'est par exemple le cas de Lisp ou de Scheme, on écrit les expressions de façon *préfixée* c'est-à-dire que le symbole d'opération précède cette fois les deux opérandes, on définit ces expressions récursivement. Les expressions préfixées comprennent:

- des symboles parmi les 4 suivants: `+ * ( )`

– des entiers naturels

Une *expression préfixée* est ou bien un nombre entier naturel ou bien est de l'une des deux formes:

(+ e_1 e_2) (* e_1 e_2)

où e_1 et e_2 sont des *expressions préfixées.*

Cette définition fait intervenir le nom de l'objet que l'on définit dans sa propre définition mais on peut montrer que cela ne pose pas de problème logique. En effet, on peut comparer cette définition à celle des nombres entiers: "tout entier naturel est ou bien l'entier 0 ou bien le successeur d'un entier naturel". On vérifie facilement que les suites de symboles suivantes sont des expressions préfixées.

```
47
(* 2 3)
(+ 12 8)
(+ (* 2 3) (+ 12 8))
(+ (*  (+ 35 36) (+ 5 7)) (* (* 2 3) (+ 95 96)))
```

Leur évaluation donne respectivement 47, 6, 20, 26 et 1998.

Pour représenter une expression préfixée en Java, on utilise ici un tableau dont chaque élément représente une entité de l'expression. Ainsi les expressions ci-dessus seront représentées par des tableaux de tailles respectives 1, 5, 5, 13, 29. Les éléments du tableau sont des objets à trois champs, le premier indique la nature de l'entité: (symbole ou nombre), le second champ est rempli par la valeur de l'entité dans le cas où celle ci est un nombre, enfin le dernier champ est un caractère rempli dans les cas où l'entité est un symbole.

```
class Element {
  boolean       estOperateur;
  int           valeur;
  char          valsymb;
}
```

On utilise les fonctions données ci-dessus pour les piles et on y ajoute les procédures suivantes :

```
static int calculer (char a, int x, int y) {

  switch (a) {
      case '+': return x + y;
      case '*': return x * y;
  }
  return -1;
}
```

La procédure d'évaluation consiste à empiler les résultats intermédiaires, la pile contiendra des opérateurs et des nombres, mais jamais deux nombres consécutivement. On examine successivement les entités de l'expression si l'entité est un opérateur ou si c'est un nombre et que le sommet de la pile est un opérateur, alors on empile cette entité. En revanche, si c'est un nombre et qu'en sommet de pile il y a aussi un nombre, on fait agir l'opérateur qui précède le sommet de pile sur les deux nombres et on répète l'opération sur le résultat trouvé.

```
static void inserer (Element x, Pile p)
    throws ExceptionPile
{
  Element    y, op;
```

												95
						5				2	+	+
			35		+	+			*	*	6	6
		+	+	71	71	71		*	*	*	*	*
	*	*	*	*	*	*	852	852	852	852	852	852
+	+	+	+	+	+	+	+	+	+	+	+	+

FIG. 3.5 – *Pile d'évaluation de l'expression dont le résultat est 1998*

```
  while (!(Pile.estVide(p)  || x.estOperateur ||
                            Pile.valeur(p).estOperateur))  {
      y = Pile.valeur(p);
      Pile.supprimer(p);
      op = Pile.valeur(p);
      Pile.supprimer(p);
      x.valeur = calculer (op.valsymb, x.valeur, y.valeur);
  }
  Pile.ajouter(x,p);
}

static int evaluer (Element u[])
    throws ExceptionPile
{
  Pile p = new Pile();

  for (int i = 0; i < u.length ; ++i) {
      inserer (u[i], p);
  }
  return Pile.valeur(p).valeur;
}
```

Dans ce cas, il peut être utile de donner un exemple de programme principal pilotant ces diverses fonctions. Remarquer qu'on se sert des arguments sur la ligne de commande pour séparer les différents nombres ou opérateurs.

```
public static void main (String args[]) {

  Element exp[] = new Element [args.length];

  for (int i = 0; i < args.length; ++i) {
     String s = args[i];
     if (s.equals ("+") || s.equals ("*"))
         exp[i] = new Element (true, 0, s.charAt(0));
     else
         exp[i] = new Element (false, Integer.parseInt(s), ' ');
  }
  try {
      System.out.println (evaluer (exp));
  } catch (ExceptionPile x) {
      System.err.println ("Pile " + x.nom);
  }
}
```

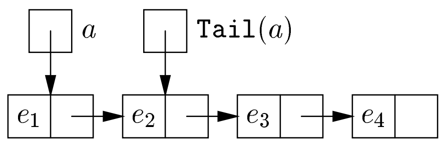


FIG. 3.6 – *Queue d'une liste*

3.5 Opérations courantes sur les listes

Nous donnons dans ce paragraphe quelques algorithmes de manipulation de listes. Ceux-ci sont utilisés dans les langages où la liste constitue une structure de base. La fonction `Tail` est une primitive classique, elle supprime le premier élément d'une liste

```
class Liste {

  Object contenu;
  Liste suivant;

  Liste (Object x, Liste a) {
      contenu = x;
      suivant = a;
  }

  static Liste cons (Object x, Liste a) {
    return new Liste (x, a);
  }

  static Object head (Liste a) {
    if (a == null)
        erreur ("Head d'une liste vide.");
    return a.contenu;
  }

  static Liste tail (Liste a) {
    if (a == null)
        erreur ("Tail d'une liste vide.");
    return a.suivant;
  }
```

Des procédures sur les listes construisent une liste à partir de deux autres, il s'agit de mettre deux listes bout à bout pour en construire une dont la longueur est égale à la somme des longueurs des deux autres. Dans la première procédure `append`, les deux listes ne sont pas modifiées; dans la seconde `nConc`, la première liste est transformée pour donner le résultat. Toutefois, on remarquera que, si `append` copie son premier argument, il partage la fin de liste de son résultat avec son deuxième argument.

```
static Liste append (Liste a, Liste b) {
  if (a == null)
      return b;
  else
      return ajouter (a.contenu, append (a.suivant, b)) ;
}

static Liste nConc (Liste a, Liste b) {
  if (a == null)
      return b;
  else {
```

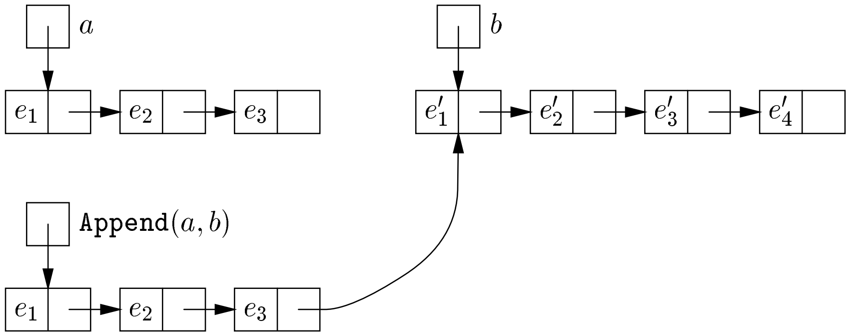


FIG. 3.7 – *Concaténation de deux listes par* Append

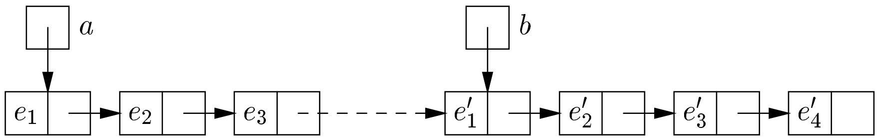


FIG. 3.8 – *Concaténation de deux listes par* Nconc

```
        Liste c = a;
        while (c.suivant != null)
     c = c.suivant;
        c.suivant = b;
        return a;
    }
}
```

Cette dernière procédure peut aussi s'écrire récursivement:

```
static Liste nConc (Liste a, Liste b) {
  if (a == null)
      return b;
  else {
      a.suivant = nConc (a.suivant, c);
      return a;
  }
}
```

La procédure de calcul de l'image miroir d'une liste a consiste à construire une liste dans laquelle les éléments de a sont rencontrés dans l'ordre inverse de ceux de a. La réalisation de cette procédure est un exercice classique de la programmation sur les listes. On en donne ici deux solutions l'une itérative, l'autre récursive, la complexité est en $O(n^2)$ donc quadratique, mais classique. A nouveau, `nReverse` modifie son argument, alors que `Reverse` ne le modifie pas et construit une nouvelle liste pour son résultat.

```
static Liste nReverse (Liste a) {
  Liste b = null;
  while (a != null) {
      Liste c = a.suivant;
      a.suivant = b;
      b = a;
      a = c;
```

```
    }
   return b;
  }

  static Liste reverse (Liste a) {
    if (a == null)
        return  a;
    else
        return  append (reverse (a.suivant),
        ajouter (a.contenu, null));
  }
```

On peut aussi avoir une version linéaire de la version récursive, grâce à une fonction auxiliaire accumulant le résultat dans un de ses arguments:

```
static Liste nReverse (Liste a) {
  return nReverse1 (null, a);
}

static Liste nReverse1 (Liste b, Liste a) {
  if (a == null)
      return b;
  else
      return nReverse1 (ajouter (a.contenu, b), a.suivant);
}
```

Un autre exercice formateur consiste à gérer des listes dans lesquelles les éléments sont rangés en ordre croissant. La procédure d'ajout devient plus complexe puisqu'on doit retrouver la position de la cellule où il faut ajouter après avoir parcouru une partie de la liste.

Nous ne traiterons cet exercice que dans le cas des *listes circulaires gardées*, voir page 67. Dans une telle liste, la valeur du champ `contenu` de la première cellule n'a aucune importance. On peut y mettre le nombre d'éléments de la liste si l'on veut. Le champ `suivant` de la dernière cellule contient lui l'adresse de la première.

```
static Liste inserer (int v, Liste a) {

  Liste b = a;
  while (b.suivant != a && v > head(b.suivant))
      b = b.suivant;
  b.suivant = ajouter (v, b.suivant);
  a.contenu = head(a) + 1;
  return a;
}
```

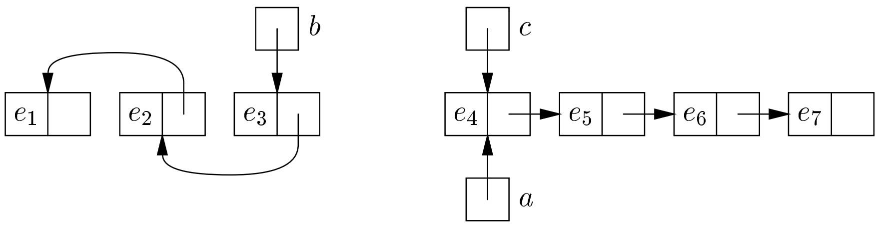


FIG. 3.9 – *Transformation d'une liste au cours de* nReverse

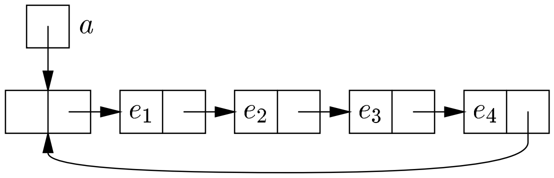


FIG. 3.10 – *Liste circulaire gardée*

3.6 Programmes en Caml

```
/* Déclaration des listes, voir page 64 */

type element == int;;

type liste = Nil | Cons of cellule

and cellule =
  { mutable contenu : element;
    mutable suivant : liste };;
```

Remarque: en Caml, les listes sont prédéfinies, et la majorité des fonctions suivantes préexistent. Nous nous amusons donc à les redéfinir. Toutefois, nos nouvelles listes sont modifiables.

```
(* Liste vide, voir page 64 *)
let estVide a =
  a = Nil;;

(* Ajouter, voir page 64 *)
let ajouter x a =
  Cons {contenu = x; suivant = a};;
```

```
(* Recherche, voir page 65 *)
let recherche x a =
 let l = ref a in
 let result = ref false in
 while !l <> Nil do
  match !l with
  | Cons {contenu = y; suivant = s} ->
     if x = y then result := true
     else l := s
  | _ -> ()
 done;
 !result;;
```

```
(* Recherche récursive, voir page 65 *)
let recherche x a =
  match a with
    Nil -> false
  | Cons {contenu = y; suivant = s} ->
     if x = y then true
     else recherche x s;;
```

```
let recherche x a =
  match a with
    Nil -> false
  | Cons {contenu = y; suivant = s} ->
     x = y || recherche x s;;
```

```
(* Longueur d'une liste, voir page 66 *)
let rec longueur = function
    Nil -> 0
  | Cons {suivant = reste; _} ->
      1 + longueur reste;;
```

```
(* Longueur d'une liste, voir page 66 *)
let longueur a =
  let r = ref 0
  and accu = ref a in
  while !accu <> Nil do
    incr r;
    match !l with
    | Cons {suivant = reste; _} ->
        accu := reste
    | _ -> ()
  done;
  !r;;
```

```
(* Supprimer, voir page 66 *)
let rec supprimer x a =
   match a with
   Nil -> a
 | Cons ({contenu = c; suivant = s} as cellule) ->
    if c = x then s
    else begin
      cellule.suivant <- supprimer x s;
      a
    end;;
```

```
let rec supprimer x a =
   match a with
   Nil -> Nil
 | Cons {contenu = c; suivant = s} ->
    if c = x then s
    else Cons {contenu = c; suivant = s};;
```

```
(* Liste des nombres premiers, voir page 68 *)
let liste_premier n =
  let a = ref Nil in
  for i = n downto 2 do ajouter i a done;
```

```
  let k = ref 2 in
  let b = ref !a in
  while !k * !k <= n do
    match !b with
      Nil -> failwith "liste_premier"
    | Cons {contenu = c; suivant = s} ->
       k := c;
       for j = c to n / c do
         supprimer (j * !k) a done;
       b := s
  done;
  a;;
```

```
(* Les files avec des vecteurs, voir page 70 *)
let maxF = 100;;

type 'a file =
  { mutable debut: int;
    mutable fin: int;
    mutable pleine: bool;
    mutable vide: bool;
    contenu: 'a vect };;

let vide () = {debut = 0; fin = 0;
  pleine = false; vide = true;
  contenu = make_vect maxF 0};;

let faireVide f = begin
  f.debut = 0; f.fin = 0;
  f.pleine = false; f.vide = true;
  end;;

let estVide f = f.vide;;

let estPleine f = f.pleine;;

let valeur f = begin
  if f.vide then failwith "Pile vide.";
  f.contenu.(f.debut)
  end;;

let successeur x = (x + 1) mod maxF ;;

let ajouter x f = begin
  if f.pleine then failwith "Pile pleine.";
  f.contenu.(f.fin) <- x;
  f.fin <- successeur (f.fin);
  f.vide <- false;
  f.pleine <- f.fin = f.debut;
  end;;

let supprimer f = begin
  if f.vide then failwith "File vide.";
  f.debut <- successeur (f.debut);
  f.vide <-  f.fin = f.debut;
  f.pleine <- false;
  end;;
```

```
(* Les files avec des listes, voir page 71 *)

type 'a liste = Nil | Cons of 'a cellule
and 'a cellule =
  {contenu: 'a; mutable suivant: 'a liste};;

type 'a file =
  {mutable debut: 'a cellule;
   mutable fin: 'a cellule };;

let vide () =
  let b = {contenu = 0; suivant = Nil} in
  {debut = b; fin = b};;

let faireVide f =
  let b = {contenu = 0; suivant = Nil} in
  f.debut <- b; f.fin <- b;;

let estVide f = f.fin == f.debut;;

let valeur f =
  match f.debut.suivant with
    Nil -> failwith "Pile vide"
  | Cons cell -> cell.contenu;;

let ajouter x f =
  let b = {contenu = x; suivant = Nil} in
  f.fin.suivant <- Cons b;
  f.fin <- b;;

let supprimer f =
  match f.debut.suivant with
    Nil -> ()
  | Cons cell -> f.debut <- cell;;
```

```
(* Déclarations et opérations sur les piles, voir page 73 *)

exception ExceptionPile of string;;

let maxP = 100;;

type 'a pile =
  {mutable hauteur: int;
   mutable contenu: 'a vect};;

let vide () =
  {hauteur = 0; contenu = make_vect maxP 0};;

let faireVide p = p.hauteur <- 0;;

let estVide p = p.hauteur = 0;;

let estPleine p = p.hauteur = maxP;;

let ajouter x p =
  if estPleine p then
    raise (ExceptionPile "pleine");
  p.contenu.(p.hauteur) <- x;
  p.hauteur <- p.hauteur + 1;;
```

```
let pvaleur p =
  if estVide p then
    raise (ExceptionPile "vide");
  p.contenu.(p.hauteur - 1);;

let psupprimer p =
  if estVide p then
    raise (ExceptionPile "vide");
  p.hauteur <- p.hauteur - 1;;
```

```
(* Opérations sur les piles, version plus traditionelle *)

type 'a pile == 'a list ref;;

let vide () = ref [];;

let faireVide (p) = p := [];;

let estVide p = (!p = []);;

let ajouter x p = p := x :: !p;;

let valeur p = match !p with
    [] -> failwith "Pile Vide."
  | x :: p' -> x;;

let supprimer p =  p := match !p with
    [] -> failwith "Pile Vide."
  | x :: p' -> p';;
```

```
(* Evaluation des expressions préfixées,
   voir page 75 *)
type expression == element vect
and element =
    Symbole of char
  | Nombre of int;;

let calculer a x y =
  match a with
    `+` -> x + y
  | `*` -> x * y
  | _ -> failwith "unknown operator";;

let rec inserer x p =
  match x with
    Symbole c -> ajouter x p
  | Nombre n ->
     if pvide p then ajouter x p else
     match valeur p with
       Symbole c as y -> ajouter x p
     | Nombre m ->
        supprimer p;
        match valeur p with
          Symbole c ->
          supprimer p;
```

```
        let res = Nombre (calculer c n m) in
        inserer res p
      | _ -> failwith "pile mal construite";;

let evaluer u =
  let p =
    {hauteur = 0;
     contenu = make_vect 100 (Nombre 0)} in
  for i = 0 to vect_length u - 1 do
    inserer u.(i) p
  done;
  match valeur p with
    Symbole c -> failwith "pile mal construite"
  | Nombre n -> n;;
```

```
let pile_of_string s =
  let u = make_vect (string_length s) (Symbole ` `) in
  let l = ref 0 in
  for i = 0 to string_length s - 1 do
    let element =
      match s.[i] with
        `0` .. `9` as c ->
         let n =
          int_of_char c - int_of_char `0` in
         begin match u.(!l) with
           Symbole c -> Nombre n
         | Nombre m ->
            decr l; Nombre (10 * m + n)
         end
      | c -> Symbole c in
    incr l;
    u.(!l) <- element
  done;
  sub_vect u 0 !l;;

evaluer
 (pile_of_string
   "(* (+ 10 (* 2 3)) (+ (* 10 10) (* 9 9)))");;
- : int = 1996
```

```
(* Tail et cons, voir page 77 *)
let tail a =
  match a with
    Nil -> failwith "tail"
  | Cons cell -> cell.contenu;;

let cons x l =
  Cons {contenu = x; suivant = l};;

(* Append et nconc, voir page 77 *)
let rec append a b =
  match a with
    Nil -> b
```

```
  | Cons cell ->
     cons (cell.contenu)
          (append cell.suivant b);;

let nconc ap b =
  match !ap with
    Nil -> ap := b
  | _ ->
     let c = ref !ap in
     while
      match !c with
        Cons {suivant = (Cons cell as l); _} ->
          c := l; true
      | _ -> false
     do () done;
     match  !c with
        Cons cell -> cell.suivant <- b
      | _ -> ();;
```

(* Version plus conforme au style Caml: avec une fonction récursive locale, on fait l'opération physique sur la liste *)

```
let rec nconc a b =
  let rec nconc1 c =
   match c with
     Nil -> b
   | Cons ({suivant = Nil; _} as cell) ->
       cell.suivant <- b; a
   | Cons {suivant = l; _} -> nconc1 l in
 nconc_aux a;;
```

(* Nreverse et reverse, voir page 78 *)

```
let nreverse ap =
 let a = ref !ap in
 let b = ref Nil in
 let c = ref Nil in
 while
   match !a with
   | Nil -> false
   | Cons ({suivant = s; _} as cell) ->
      c := s;
      cell.suivant <- !b;
      b := !a;
      a := !c;
      true
 do () done;
 ap := !b;;
```

(* Version plus conforme à Caml: on renverse la liste en place et l'on rend la nouvelle tête de liste *)

```
let nreverse l =
 let rec nreverse1 a b =
  match a with
```

```
  | Nil -> b
  | Cons ({suivant = s; _} as cell) ->
     cell.suivant <- b;
     nreverse1 s a in
  nreverse_aux l Nil;;

let rec reverse a =
   match a with
   | Nil -> a
   | Cons cell ->
      append (reverse cell.suivant)
             (cons (cell.contenu) Nil);;
```

```
(* Insert, voir page 79 *)
let insert v l =
  match l with
    Nil -> failwith "insert"
  | Cons c ->
     let rec insert_aux a =
      match a with
       | Nil -> failwith "insert"
       | Cons cell ->
          if v > cell.contenu && cell != c
          then insert_aux cell.suivant
          else cell.suivant <-
                 cons v cell.suivant in
     insert_aux c.suivant;
     c.contenu <- l.contenu + 1;;
```

Chapitre 4

Arbres

Nous avons déjà vu la notion de fonction récursive dans le chapitre 2. Considérons à présent son équivalent dans les structures de données: la notion d'arbre. Un arbre est soit un arbre atomique (une *feuille*), soit un *noeud* et une suite de sous-arbres. Graphiquement, un arbre est représenté comme suit

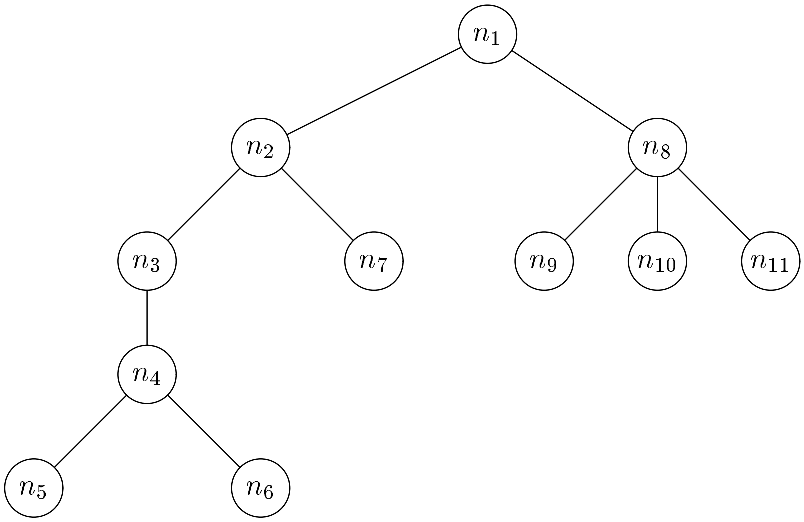


FIG. 4.1 – *Un exemple d'arbre*

Le noeud n_1 est la *racine* de l'arbre, n_5, n_6, n_7, n_9, n_{10}, n_{11} sont les *feuilles*, n_1, n_2, n_3, n_4, n_8 les *noeuds internes*. Plus généralement, l'ensemble des *noeuds* est constitué des noeuds internes et des feuilles. Contrairement à la botanique, on dessine les arbres avec la racine en haut et les feuilles vers le bas en informatique. Il y a bien des définitions plus mathématiques des arbres, que nous éviterons ici. Si une branche relie un noeud n_i à un noeud n_j plus bas, on dira que n_i est un *ancêtre* de n_j. Une propriété fondamentale d'un arbre est qu'un noeud n'a qu'un seul père. Enfin, un noeud peut contenir une ou plusieurs valeurs, et on parlera alors d'*arbres étiquetés* et de la valeur (ou des valeurs) d'un noeud. Les *arbres binaires* sont des arbres tels que les noeuds ont au plus 2 fils. La *hauteur*, on dit aussi la *profondeur* d'un noeud est la longueur du chemin qui le joint à la racine, ainsi la racine est elle même de hauteur 0, ses fils de hauteur 1 et les autres noeuds de hauteur supérieure à 1.

Un exemple d'arbre très utilisé en informatique est la représentation des expressions arithmétiques et plus généralement des termes dans la programmation symbolique. Nous

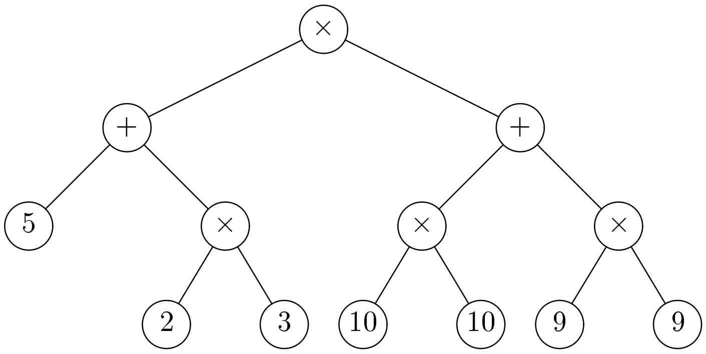


FIG. 4.2 – *Représentation d'une expression arithmétique par un arbre*

traiterons ce cas dans le chapitre sur l'analyse syntaxique, et nous nous restreindrons pour l'instant au cas des arbres de recherche ou des arbres de tri. Toutefois, pour montrer l'aspect fondamental de la structure d'arbre, on peut tout de suite voir que les expressions arithmétiques calculées dans la section 3.4 se représentent simplement par des arbres comme dans la figure 4.2 pour `(* (+ 5 (* 2 3)) (+ (* 10 10) (* 9 9)))`. Cette représentation contient l'essence de la structure d'expression arithmétique et fait donc abstraction de toute notation préfixée ou postfixée.

4.1 Files de priorité

Un premier exemple de structure arborescente est la structure de tas (*heap*[1])utilisée pour représenter des files de priorité. Donnons d'abord une vision intuitive d'une file de priorité.

On suppose, comme au paragraphe 3.2, que des gens se présentent au guichet d'une banque avec un numéro écrit sur un bout de papier représentant leur degré de priorité. Plus ce nombre est élevé, plus ils sont importants et doivent passer rapidement. Bien sûr, il n'y a qu'un seul guichet ouvert, et l'employé(e) de la banque doit traiter rapidement tous ses clients pour que tout le monde garde le sourire. La file des personnes en attente s'appelle une *file de priorité*. L'employé de banque doit donc savoir faire rapidement les 3 opérations suivantes: trouver un maximum dans la file de priorité, retirer cet élément de la file, savoir ajouter un nouvel élément à la file. Plusieurs solutions sont envisageables.

La première consiste à mettre la file dans un tableau et à trier la file de priorité dans l'ordre croissant des priorités. Trouver un maximum et le retirer de la file est alors simple: il suffit de prendre l'élément de droite, et de déplacer vers la gauche la borne droite de la file. Mais l'insertion consiste à faire une passe du tri par insertion pour mettre le nouvel élément à sa place, ce qui peut prendre un temps $O(n)$ où n est la longueur de la file.

Une autre méthode consiste à gérer la file comme une simple file du chapitre précédent, et à rechercher le maximum à chaque fois. L'insertion est rapide, mais la recherche du maximum peut prendre un temps $O(n)$, de même que la suppression.

Une méthode élégante consiste à gérer une structure d'ordre partiel grâce à un arbre. La file de n éléments est représentée par un arbre binaire contenant en chaque noeud un élément de la file (comme illustré dans la figure 4.3). L'arbre vérifie deux propriétés

1. Le mot *heap* a malheureusement un autre sens en Pascal: c'est l'espace dans lequel sont allouées les variables dynamiques référencées par un pointeur après l'instruction `new`. Il sera bien clair d'après le contexte si nous parlons de tas au sens des files de priorité ou du tas de Pascal pour allouer les variables dynamiques.

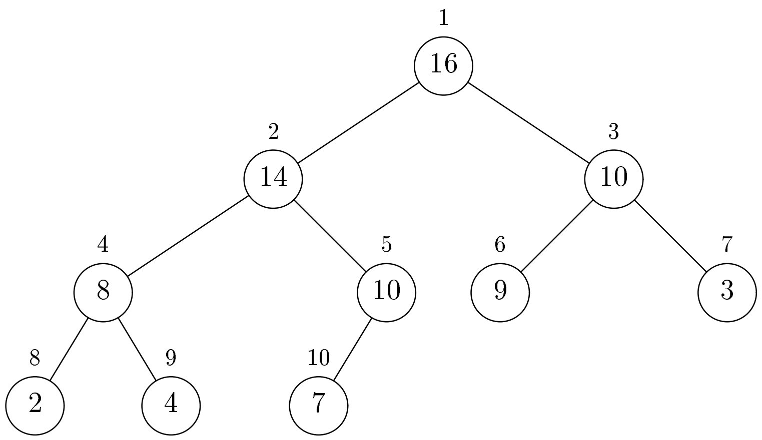


FIG. 4.3 – *Représentation en arbre d'un tas*

i	1	2	3	4	5	6	7	8	9	10
$a[i]$	16	14	10	8	10	9	3	2	4	7

FIG. 4.4 – *Représentation en tableau d'un tas*

importantes: d'une part la valeur de chaque noeud est supérieure ou égale à celle de ses fils, d'autre part l'arbre est quasi complet, propriété que nous allons décrire brièvement. Si l'on divise l'ensemble des noeuds en lignes suivant leur hauteur, on obtient en général dans un arbre binaire une ligne 0 composée simplement de la racine, puis une ligne 1 contenant au plus deux noeuds, et ainsi de suite (la ligne i contenant au plus 2^i noeuds). Dans un arbre quasi complet les lignes, exceptée peut être la dernière, contiennent toutes un nombre maximal de noeuds (soit 2^i). De plus les feuilles de la dernière ligne se trouvent toutes à gauche, ainsi tous les noeuds internes sont binaires, excepté le plus à droite de l'avant dernière ligne qui peut ne pas avoir de fils droit. Les feuilles sont toutes sur la dernière et éventuellement l'avant dernière ligne.

On peut numéroter cet arbre en largeur d'abord, c'est à dire dans l'ordre donné par les petits numéros figurant au dessus de la figure 4.3. Dans cette numérotation on vérifie que tout noeud i a son père en position $\lfloor i/2 \rfloor$, le fils gauche du noeud i est $2i$, le fils droit $2i+1$. Formellement, on peut dire qu'un tas est un tableau a contenant n entiers (ou des éléments d'un ensemble totalement ordonné) satisfaisant les conditions:

$$\begin{array}{rcl} 2 \leq 2i \leq n & \Rightarrow & a[2i] \geq a[i] \\ 3 \leq 2i+1 \leq n & \Rightarrow & a[2i+1] \geq a[i] \end{array}$$

Ceci permet d'implémenter cet arbre dans un tableau `a` (voir figure 4.4) où le numéro de chaque noeud donne l'indice de l'élément du tableau contenant sa valeur.

L'ajout d'un nouvel élément v à la file consiste à incrémenter n puis à poser $a[n] = v$. Ceci ne représente plus un tas car la relation $a[n/2] \geq v$ n'est pas nécessairement satisfaite. Pour obtenir un tas, il faut échanger la valeur contenue au noeud n et celle contenue par son père, remonter au père et réitérer jusqu'à ce que la condition des tas soit vérifiée. Ceci se programme par une simple itération (cf. la figure 4.5).

```
static void ajouter (int v)  {

  ++nTas;
  int i = nTas - 1;
```

```
    while (i > 0 && a [(i - 1)/2] <= v) {
        a[i] = a[(i - 1)/2];
        i = (i - 1)/2;
    }
    a[i] = v;
}
```

On vérifie que, si le tas a n éléments, le nombre d'opérations n'excédera pas la hauteur de l'arbre correspondant. Or la hauteur d'un arbre binaire complet de n noeuds est $\log n$. Donc `Ajouter` ne prend pas plus de $O(\log n)$ opérations.

On peut remarquer que l'opération de recherche du maximum est maintenant immédiate dans les tas. Elle prend un temps constant $O(1)$.

```
static int maximum ()  {

  return a[0];
}
```

Considérons l'opération de suppression du premier élément de la file. Il faut alors retirer la racine de l'arbre représentant la file, ce qui donne deux arbres! Le plus simple pour reformer un seul arbre est d'appliquer l'algorithme suivant: on met l'élément le plus à droite de la dernière ligne à la place de la racine, on compare sa valeur avec celle de ses fils, on échange cette valeur avec celle du vainqueur de ce tournoi, et on réitère cette opération jusqu'à ce que la condition des tas soit vérifiée. Bien sûr, il faut faire attention, quand un noeud n'a qu'un fils, et ne faire alors qu'un petit tournoi à deux. Le placement de la racine en bonne position est illustré dans la figure 4.6.

```
static void supprimer () {

  int i, j;
  int v;

  a[0] = a[nTas - 1];
  --nTas;
  i = 0; v = a[0];
  while (2*i + 1 < nTas) {
      j = 2*i + 1;
      if (j + 1 < nTas)
          if (a[j + 1] > a[j])
              ++j;
      if (v >= a[j])
          break;
      a[i] = a[j]; i = j;
  }
  a[i] = v;
}
```

A nouveau, la suppression du premier élément de la file ne prend pas un temps supérieur à la hauteur de l'arbre représentant la file. Donc, pour une file de n éléments, la suppression prend $O(\log n)$ opérations. La représentation des files de priorités par des tas permet donc de faire les trois opérations demandées: ajout, retrait, chercher le plus grand en $\log n$ opérations. Ces opérations sur les tas permettent de faire le tri *HeapSort*. Ce tri peut être considéré comme alambiqué, mais il a la bonne propriété d'être toujours en temps $n \log n$ (comme le Tri fusion, cf page 57).

HeapSort se divise en deux phases, la première consiste à construire un tas dans le tableau à trier, la seconde à répéter l'opération de prendre l'élément maximal, le retirer du tas en le mettant à droite du tableau. Il reste à comprendre comment on peut

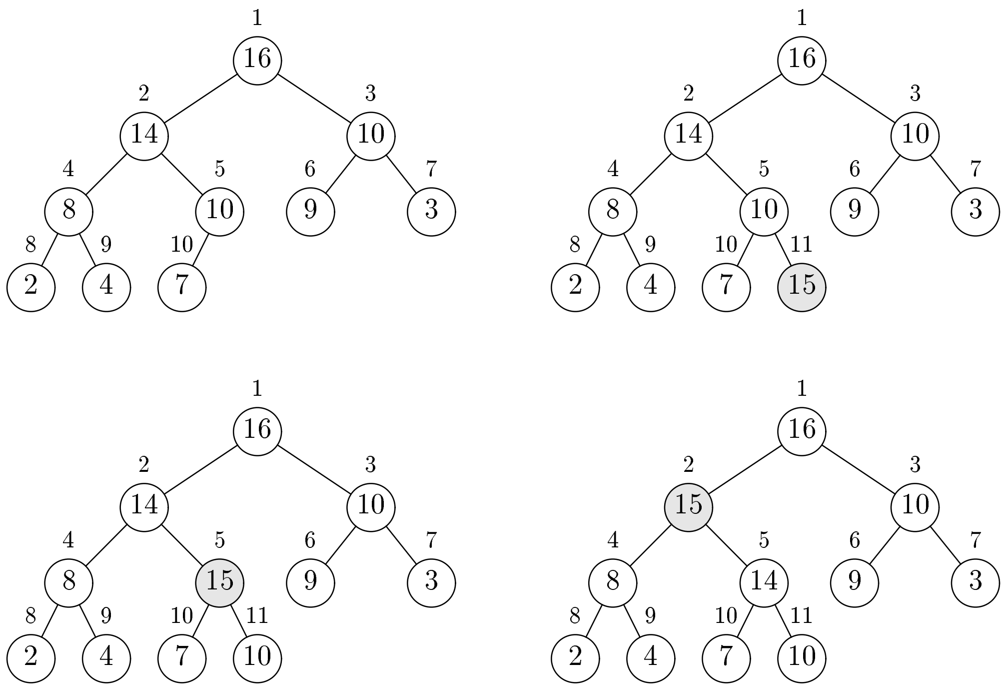


FIG. 4.5 – *Ajout dans un tas*

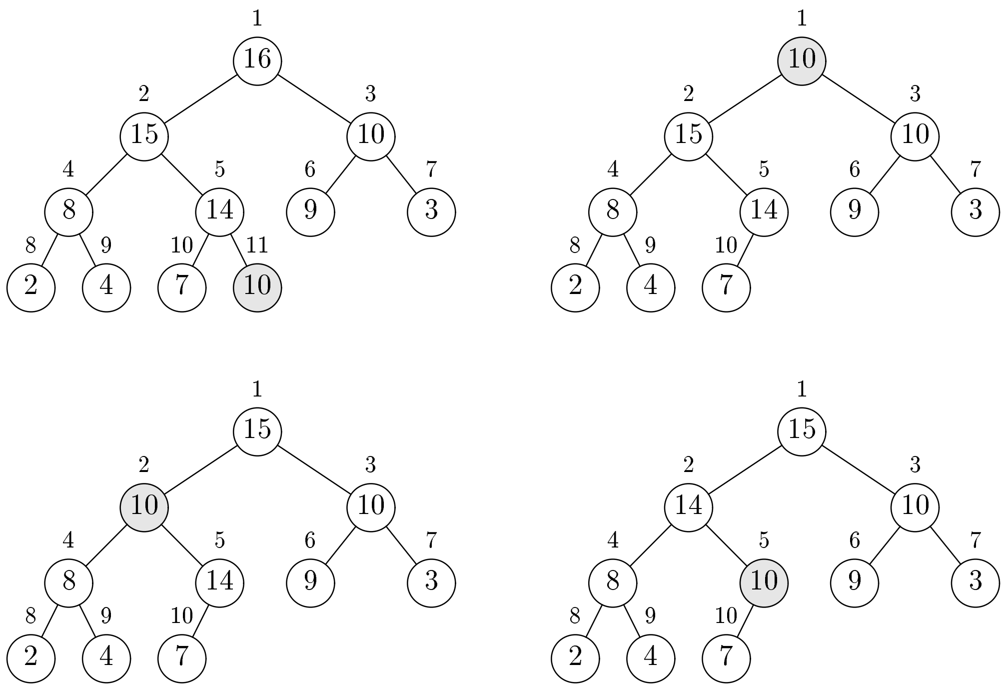


FIG. 4.6 – *Suppression dans un tas*

construire un tas à partir d' un tableau quelconque. Il y a une méthode peu efficace, mais systématique. On remarque d'abord que l'élément de gauche du tableau est à lui seul un tas. Puis on ajoute à ce tas le deuxième élément avec la procédure `Ajouter` que nous venons de voir, puis le troisième, ... A la fin, on obtient bien un tas de N éléments dans le tableau `a` à trier. Le programme est

```
static void heapSort ()  {

  int      i;
  int      v;

  nTas = 0;
  for (i = 0; i < N; ++i)
      ajouter (a[i]);
  for (i = N - 1; i >= 0; --i) {
      v = maximum();
      supprimer();
      a[i] = v;
   }
}
```

Si on fait un décompte grossier des opérations, on remarque qu'on ne fait pas plus de $N \log N$ opérations pour construire le tas, puisqu'il y a N appels à la procédure `Ajouter`. Une méthode plus efficace, que nous ne décrirons pas ici, qui peut être traitée à titre d'exercice, permet de construire le tas en $O(N)$ opérations. De même, dans la deuxième phase, on ne fait pas plus de $N \log N$ opérations pour défaire les tas, puisqu'on appelle N fois la procédure `Supprimer`. Au total, on fait $O(N \log N)$ opérations quelle que soit la distribution initiale du tableau `a`, comme dans le tri fusion. On peut néanmoins remarquer que la constante qui se trouve devant $N \log N$ est grande, car on appelle des procédures relativement complexes pour faire et défaire les tas. Ce tri a donc un intérêt théorique, mais il est en pratique bien moins bon que Quicksort ou le tri Shell.

4.2 Borne inférieure sur le tri

Il a été beaucoup question du tri. On peut se demander s'il est possible de trier un tableau de N éléments en moins de $N \log N$ opérations. Un résultat ancien de la théorie de l'information montre que c'est impossible si on n'utilise que des comparaisons.

En effet, il faut préciser le modèle de calcul que l'on considère. On peut représenter tous les tris que nous avons rencontrés par des arbres de décision. La figure 4.7 représente un tel arbre pour le tri par insertion sur un tableau de 3 éléments. Chaque noeud interne pose une question sur la comparaison entre 2 éléments. Le fils de gauche correspond à la réponse négative, le fils droit à l'affirmatif. Les feuilles représentent la permutation à effectuer pour obtenir le tableau trié.

Théorème 1 *Le tri de N éléments, fondé uniquement sur les comparaisons des éléments deux à deux, fait au moins $O(N \log N)$ comparaisons.*

Démonstration Tout arbre de décision pour trier N éléments a $N!$ feuilles représentant toutes les permutations possibles. Un arbre binaire de $N!$ feuilles a une hauteur de l'ordre de $\log N! \simeq N \log N$ par la formule de Stirling. □

Corollaire 1 HeapSort *et le tri fusion sont optimaux (asymptotiquement).*

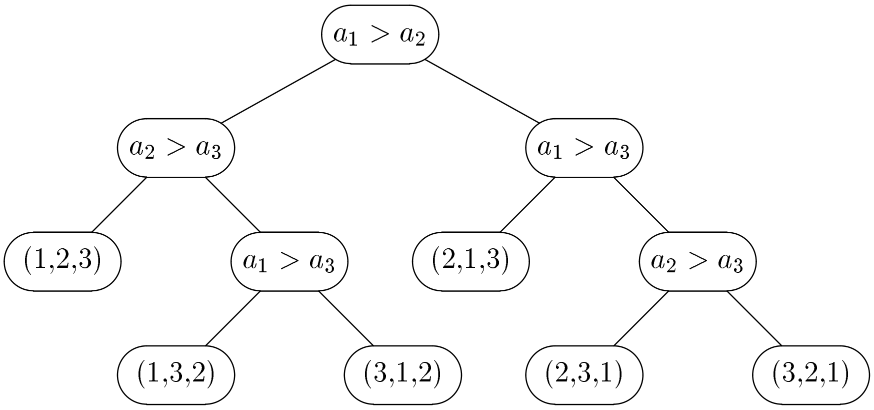


FIG. 4.7 – *Exemple d'arbre de décision pour le tri*

En effet, ils accomplissent le nombre de comparaisons donné comme borne inférieure dans le théorème précédent. Mais, nous répétons qu'un tri comme Quicksort est aussi très bon en moyenne. Le modèle de calcul par comparaisons donne une borne inférieure, mais peut-on faire mieux dans un autre modèle? La réponse est oui, si on dispose d'informations annexes comme les valeurs possibles des éléments a_i à trier. Par exemple, si les valeurs sont comprises dans l'intervalle $[1, k]$, on peut alors prendre un tableau b annexe de k éléments qui contiendra en b_j le nombre de a_i ayant la valeur j. En une passe sur a, on peut remplir le tableau k, puis générer le tableau a trié en une deuxième passe en ne tenant compte que de l'information rangée dans b. Ce tri prend $O(k + 2N)$ opérations, ce qui est très bon si k est petit.

4.3 Implémentation d'un arbre

Jusqu'à présent, les arbres sont apparus comme des entités abstraites ou n'ont été implémentés que par des tableaux en utilisant une propriété bien particulière des arbres complets. On peut bien sûr manipuler les arbres comme les listes avec des enregistrements et des pointeurs. Tout noeud sera représenté par un enregistrement contenant une valeur et des pointeurs vers ses fils. Une feuille ne contient qu'une valeur. On peut donc utiliser des enregistrements avec variante pour signaler si le noeud est interne ou une feuille. Pour les arbres binaires, les deux fils seront représentés par les champs **`filsG, filsD`** et il sera plus simple de supposer qu'une feuille est un noeud dont les fils gauche et droit ont une valeur vide.

```
class Arbre {

  int      contenu;
  Arbre    filsG;
  Arbre    filsD;

  Arbre (int v, Arbre a, Arbre b) {
      contenu = v;
      filsG = a;
      filsD = b;
  }
}
```

Pour les arbres quelconques, on peut gagner plus d'espace mémoire en considérant des enregistrements variables. Toutefois, en Pascal, il y a une difficulté de typage à considérer des noeuds n-aires (ayant n fils). On doit considérer des types différents pour les noeuds binaires, ternaires, ... ou un gigantesque enregistrement avec variante. Deux solutions systématiques sont aussi possibles: la première consiste à considérer le cas n maximum (comme pour les arbres binaires)

```
class Arbre {

  int      contenu;
  Arbre[]  fils;

  Arbre (int v, int n) {
      contenu = v;
      fils = new Arbre[n];
  }
}
```

la deuxième consiste à enchaîner les fils dans une liste

```
class Arbre {

  int           contenu;
  ListeArbres   fils;
}

class ListeArbres {

  Arbre         contenu;
  ListeArbres   suivant;
}
```

Avec les tailles mémoire des ordinateurs actuels, on se contente souvent de la première solution. Mais, si les contraintes de mémoire sont fortes, il faut se rabattre sur la deuxième. Dans une bonne partie de la suite, il ne sera question que d'arbres binaires, et nous choisirons donc la première représentation avec les champs `filsG` et `filsD`.

Considérons à présent la construction d'un nouvel arbre binaire `c` à partir de deux arbres `a` et `b`. Un noeud sera ajouté à la racine de l'arbre et les arbres `a` et `b` seront les fils gauche et droit respectivement de cette racine. La fonction correspondante prend la valeur du nouveau noeud, les fils gauche et droit du nouveau noeud. Le résultat sera un pointeur vers ce noeud nouveau. Voici donc comment créer l'arbre de gauche de la figure 4.8.

```
class Arbre {

  int      contenu;
  Arbre    filsG;
  Arbre    filsD;

  Arbre (int v, Arbre a, Arbre b) {
      contenu = v;
      filsG = a;
      filsD = b;
  }

  public static void main(String args[]) {

    Arbre a5, a7;
    a5 = new Arbre (12, new Arbre (8, new Arbre (6, null, null), null),
                        new Arbre (13, null, null));
    a7 = new Arbre (20, new Arbre (3, new Arbre (3, null, null), a5),
                        new Arbre (25, new Arbre (21, null, null),
```

```
                                          new Arbre (28, null, null)));
    imprimer (a7);
}
```

Une fois un arbre créé, il est souhaitable de pouvoir l'imprimer. Plusieurs méthodes sont possibles. La plus élégante utilise les fonctions graphiques du Macintosh **drawString**, **moveTo**, **lineTo**. Une autre consiste à utiliser une notation linéaire avec des parenthèses. C'est la notation *infixe* utilisée couramment si les noeuds internes sont des opérateurs d'expressions arithmétique. L'arbre précédent s'écrit alors

```
((3 3 ((6 8 nil) 12 13)) 20 (21 25 28))
```

Utilisons une méthode plus rustique en imprimant en alphanumérique sur plusieurs lignes. Ainsi, en penchant un peu la tête vers la gauche, on peut imprimer l'arbre précédent comme suit

```
20      25      28
        21
 3      12      13
         8
         6
 3
```

La procédure d'impression prend comme argument l'arbre à imprimer et la tabulation à faire avant l'impression, c'est à dire le nombre d'espaces. On remarquera que toute la difficulté de la procédure est de bien situer l'endroit où on effectue un retour à la ligne. Le reste est un simple parcours récursif de l'arbre en se plongeant d'abord dans l'arbre de droite.

```
static void imprimer (Arbre a) {
  imprimer (a, 0);
  System.out.println ();
}

static void imprimer1 (Arbre a, int tab) {

  if (a != null) {
    System.out.print (leftAligned (3, a.contenu + "") + "     ");
      imprimer1 (a.filsD, tab + 8);
      if (a.filsG != null) {
          System.out.println ();
          for (int i = 1; i <= tab; ++i)
              System.out.print (" ");
      }
      imprimer1 (a.filsG, tab);
  }
}

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);
}
```

Nous avons donc vu comment représenter un arbre dans un programme, comment le construire, et comment l'imprimer. Cette dernière opération est typique: pour explorer une structure de donnée récursive (les arbres), il est naturel d'utiliser des procédures récursives. C'est à nouveau une manière non seulement naturelle, mais aussi très efficace dans le cas présent.

Comme pour les listes (cf. page 65), la structure récursive des programmes manipulant des arbres découle de la définition des arbres, puisque le type des arbres binaires vérifie l'équation:

$$\boxed{\texttt{Arbre} = \{\text{Arbre\_vide}\} \uplus \texttt{Arbre} \times \texttt{Element} \times \texttt{Arbre}}$$

Comme pour l'impression, on peut calculer le nombre de noeuds d'un arbre en suivant la définition récursive du type des arbres:

```
static int taille (Arbre a) {

  if (a == null)     (* a = Arbre_vide *)
      return 0;
  else               (* a ∈ Arbre × Element × Arbre *)
      return 1 + taille (a.filsG) + taille (a.filsD);
}
```

L'écriture itérative dans le cas des arbres est en général impossible sans utiliser une pile. On vérifie que, pour les arbres binaires qui ne contiennent pas de noeuds unaires, la taille t, le nombre de feuilles N_f et le nombre de noeuds internes N_n vérifient $t = N_n + N_f$ et $N_n = 1 + N_f$.

4.4 Arbres de recherche

La recherche en table et le tri peuvent être aussi traités avec des arbres. Nous l'avons vu implicitement dans le cas de Quicksort. En effet, si on dessine les partitions successives obtenues par les appels récursifs de Quicksort, on obtient un arbre. On introduit pour les algorithmes de recherche d'un élément dans un ensemble ordonné la notion d'*arbre binaire de recherche* celui-ci aura la propriété fondamentale suivante: tous les noeuds du sous-arbre gauche d'un noeud ont une valeur inférieure (ou égale) à la sienne et tous les noeuds du sous-arbre droit ont une valeur supérieure (ou égale) à la valeur du noeud lui-même (comme dans la figure 4.8). Pour la recherche en table, les arbres de recherche ont un intérêt quand la table évolue très rapidement, quoique les méthodes avec hachage sont souvent aussi bonnes, mais peuvent exiger des contraintes de mémoire impossibles à satisfaire. (En effet, il faut connaître la taille maximale *a priori* d'une table de hachage). Nous allons voir que le temps d'insertion d'un nouvel élément dans un arbre de recherche prend un temps comparable au temps de recherche si cet arbre est bien agencé. Pour le moment, écrivons les procédures élémentaires de recherche et d'ajout d'un élément.

```
static Arbre recherche (int v, Arbre a) {

  if (a == null || v == a.contenu)
       return a;
  else if (v < a.contenu)
       return recherche (v, a.filsG);
  else
       return recherche (v, a.filsD);
}

static Arbre ajouter (int v, Arbre a) {

  if (a == null)
      a = new Arbre (v, null, null);
```

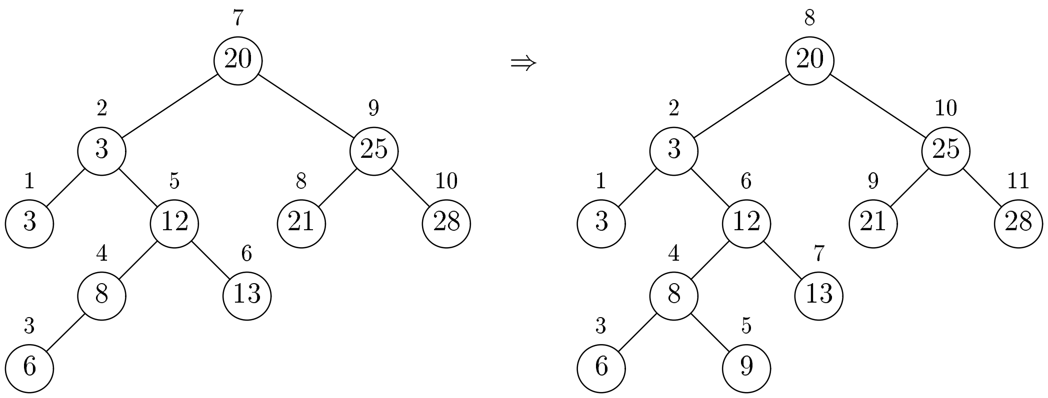


FIG. 4.8 – *Ajout dans un arbre de recherche*

```
    else if (v <= a.contenu)
        a.filsG = ajouter (v, a.filsG);
    else
        a.filsD = ajouter (v, a.filsD);
    return a;
}
```

A nouveau, des programmes récursifs correspondent à la structure récursive des arbres. La procédure de recherche renvoie un pointeur vers le noeud contenant la valeur recherchée, `null` si échec. Il n'y a pas ici d'information associée à la clé recherchée comme au chapitre 1. On peut bien sûr associer une information à la clé recherchée en ajoutant un champ dans l'enregistrement décrivant chaque noeud. Dans le cas du bottin de téléphone, le champ `contenu` contiendrait les noms et serait du type `String`; l'information serait le numéro de téléphone.

La recherche teste d'abord si le contenu de la racine est égal à la valeur recherchée, sinon on recommence récursivement la recherche dans l'arbre de gauche si la valeur est plus petite que le contenu de la racine, ou dans l'arbre de droite dans le cas contraire. La procédure d'insertion d'une nouvelle valeur suit le même schéma. Toutefois dans le cas de l'égalité des valeurs, nous la rangeons ici par convention dans le sous arbre de gauche. La procédure `ajouter` modifie l'arbre de recherche. Si nous ne voulons pas le modifier, nous pouvons adopter comme dans le cas des listes la procédure suivante, qui consomme légèrement plus de place, laquelle place peut être récupérée très rapidement par le glaneur de cellules:

```
static Arbre ajouter (int v, Arbre a) {

    if (a == null)
        return new Arbre (v, null, null);
    else if (v <= a.contenu)
        return new Arbre (v, ajouter (v, a.filsG), a.filsD);
    else
        return new Arbre (v, a.filsG, ajouter (v, a.filsG));
}
```

Le nombre d'opérations de la recherche ou de l'insertion dépend de la hauteur de l'arbre. Si l'arbre est bien équilibré, pour un arbre de recherche contenant N noeuds, on effectuera $O(\log N)$ opérations pour chacune des procédures. Si l'arbre est un peigne, c'est à dire complètement filiforme à gauche ou à droite, la hauteur vaudra N et le nombre d'opérations sera $O(N)$ pour la recherche et l'ajout. Il apparaît donc souhaitable d'équilibrer les arbres au fur et à mesure de l'ajout de nouveaux éléments, ce que nous allons voir dans la section suivante.

Enfin, l'ordre dans lequel sont rangés les noeuds dans un arbre de recherche est appelé *ordre infixe*. Il correspond au petit numéro qui se trouve au dessus de chaque noeud dans la figure 4.8. Nous avons déjà vu dans le cas de l'évaluation des expressions arithmétiques (cf page 74) *l'ordre préfixe*, dans lequel tout noeud reçoit un numéro d'ordre inférieur à celui de tous les noeuds de son sous-arbre de gauche, qui eux-mêmes ont des numéros inférieurs aux noeuds du sous-arbre de droite. Finalement, on peut considérer *l'ordre postfixe* qui ordonne d'abord le sous-arbre de gauche, puis le sous-arbre de droite, et enfin le noeud. C'est un bon exercice d'écrire un programme d'impression correspondant à chacun de ces ordres, et de comparer l'emplacement des différents appels récursifs.

4.5 Arbres équilibrés

La notion d'arbre équilibré a été introduite en 1962 par deux russes Adel'son-Vel'skii et Landis, et depuis ces arbres sont connus sous le nom d'arbres AVL. Il y a maintenant beaucoup de variantes plus ou moins faciles à manipuler. Au risque de paraître classiques et vieillots, nous parlerons principalement des arbres AVL. Un arbre AVL vérifie la propriété fondamentale suivante: la différence entre les hauteurs des fils gauche et des fils droit de tout noeud ne peut excéder 1. Ainsi l'arbre de gauche de la figure 4.8 n'est pas équilibré. Il viole la propriété aux noeuds numérotés 2 et 7, tous les autres noeuds validant la propriété. Les arbres représentant des tas, voir figure 4.5 sont trivialement équilibrés.

On peut montrer que la hauteur d'un arbre AVL de N noeuds est de l'ordre de $\log N$, ainsi les temps mis par la procédure `Recherche` vue page 98 seront en $O(\log N)$.

Il faut donc maintenir l'équilibre de tous les noeuds au fur et à mesure des opérations d'insertion ou de suppression d'un noeud dans un arbre AVL. Pour y arriver, on suppose que tout noeud contient un champ annexe `bal` contenant la différence de hauteur entre le fils droit et le fils gauche. Ce champ représente donc la balance ou l'équilibre entre les hauteurs des fils du noeud, et on s'arrange donc pour maintenir $-1 \leq$ `a.bal` ≤ 1 pour tout noeud pointé par `a`.

L'insertion se fait comme dans un arbre de recherche standard, sauf qu'il faut maintenir l'équilibre. Pour cela, il est commode que la fonction d'insertion retourne une valeur représentant la différence entre la nouvelle hauteur (après l'insertion) et l'ancienne hauteur (avant l'insertion). Quand il peut y avoir un déséquilibre trop important entre les deux fils du noeud où l'on insère un nouvel élément, il faut recréer un équilibre par une rotation simple (figure 4.9) ou une rotation double (figure 4.10). Dans ces figures, les rotations sont prises dans le cas d'un rééquilibrage de la gauche vers la droite. Il existe bien sûr les 2 rotations symétriques de la droite vers la gauche. On peut aussi remarquer que la double rotation peut se réaliser par une suite de deux rotations simples. Dans la figure 4.10, il suffit de faire une rotation simple de la droite vers la gauche du noeud A, suivie d'une rotation simple vers la droite du noeud B. Ainsi en supposant déjà écrites les procédures de rotation `rotD` vers la droite et `rotG` vers la gauche, la procédure d'insertion s'écrit

```
static Arbre ajouter (int v, Arbre a) {
  return ajouter1 (v, a).p2;
}

static Paire ajouter1 (int v, Arbre a) {

  int        incr, r;
  Paire      p;
```

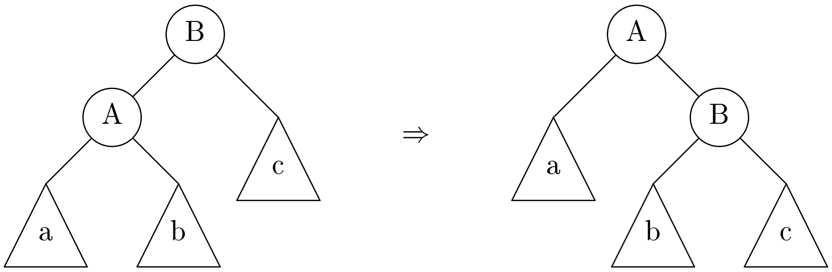


FIG. 4.9 – *Rotation dans un arbre AVL*

```
  r = 0;
  if (a == null) {
      a = new Arbre (v, null, null);
      a.bal = 0;
      r = 1;
  } else {
     if (v <= a.contenu) {
          p = ajouter1 (v, a.filsG);
          incr = -p.p1;
          a.filsG = p.p2;
     } else {
          p = ajouter1 (v, a.filsD);
          incr = p.p1;
          a.filsD = p.p2;
    }
      a.bal = a.bal + incr;
      if (incr != 0 && a.bal != 0)
          if (a.bal < -1)
              /* La gauche est trop grande */
              if (a.filsG.bal < 0)
                  a = rotD (a);
              else {
                  a.filsG = rotG (a.filsG);
                  a = rotD (a);
              }
          else
          if (a.bal > 1)
              /* La droite est trop grande */
              if (a.filsD.bal > 0)
                  a = rotG (a);
              else {
                  a.filsD = rotD (a.filsD);
                  a = rotG (a);
              }
          else
             r = 1;
  }
  return new Paire (r, a);
}
```

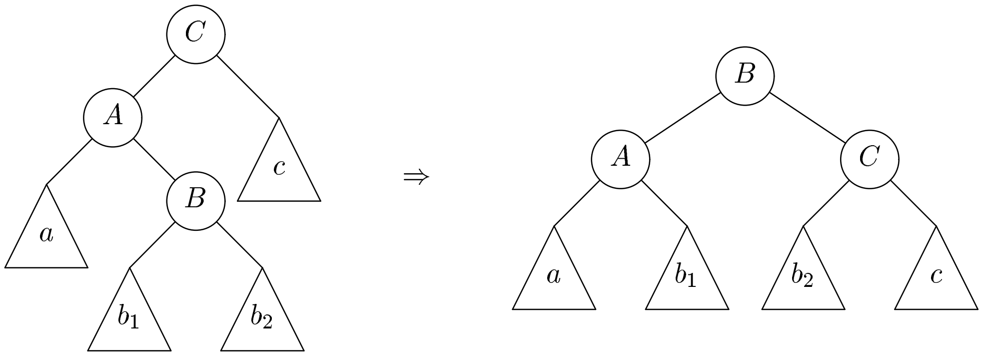


FIG. 4.10 – *Double rotation dans un arbre AVL*

```
static class Paire {
  int      p1;
  Arbre    p2;

  Paire (int r, Arbre a) {
     p1 = r;
     p2 = a;
  }
}
```

Clairement cette procédure prend un temps $O(\log N)$. On vérifie aisément qu'au plus une seule rotation (éventuellement double) est nécessaire lors de l'insertion d'un nouvel élément. Il reste à réaliser les procédures de rotation. Nous ne considérons que le cas de la rotation vers la droite, l'autre cas s'obtenant par symétrie.

```
static Arbre rotD (Arbre a) {

  Arbre  b;
  int    bA, bB, bAnew, bBnew;

  b = a;
  a = a.filsG;
  bA = a.bal; bB = b.bal;
  b.filsG = a.filsD;
  a.filsD = b;
  // Recalculer le champ a.bal
  bBnew = 1 + bB - Math.min(0, bA);
  bAnew = 1 + bA + Math.max(0, bBnew);
  a.bal = bAnew;
  b.bal = bBnew;
  return a;
}
```

Il y a un petit calcul savant pour retrouver le champ représentant l'équilibre après rotation. Il pourrait être simplifié si nous conservions toute la hauteur du noeud dans un champ. La présentation avec les champs `bal` permet de garder les valeurs possibles entre -2 et 2, de tenir donc sur 3 bits, et d'avoir le reste d'un mot machine pour le champ `contenu`. Avec la taille des mémoires actuelles, ce calcul peut se révéler surperflu. Toutefois, soient $h(a)$, $h(b)$ et $h(c)$ les hauteurs des arbres a, b et c de la figure 4.9. En appliquant la définition du champ `bal`, les nouvelles valeurs $b'(A)$ et $b'(B)$ de ces champs aux noeuds A et B se calculent en fonction des anciennes valeurs $b(A)$ et $b(B)$ par

$$\begin{aligned}
b'(B) &= h(c) - h(b) \\
&= h(c) - 1 - \lceil h(a), h(b) \rceil + 1 + \lceil h(a), h(b) \rceil - h(b) \\
&= b(B) + 1 + \lceil h(a) - h(b), 0 \rceil \\
&= 1 + b(B) - \lfloor 0, b(A) \rfloor
\end{aligned}$$

$$\begin{aligned}
b'(A) &= 1 + \lceil h(b), h(c) \rceil - h(a) \\
&= 1 + h(b) - h(a) + \lceil 0, h(c) - h(b) \rceil \\
&= 1 + b(A) + \lceil 0, b'(B) \rceil
\end{aligned}$$

Les formules pour la rotation vers la gauche s'obtiennent par symétrie. On peut même remarquer que le champ `bal` peut tenir sur 1 bit pour signaler si le sous-arbre a une hauteur égale ou non à celle de son sous-arbre "frère". La suppression d'un élément dans un arbre AVL est plus dure à programmer, et nous la laissons en exercice. Elle peut demander jusqu'à $O(\log N)$ rotations.

Les arbres AVL sont délicats à programmer à cause des opérations de rotation. On peut montrer que les rotations deviennent inutiles si on donne un peu de flexibilité dans le nombre de fils des noeuds. Il existe des arbres de recherche 2-3 avec 2 ou 3 fils. L'exemple le plus simple est celui des arbres 2-3-4 amplement décrits dans le livre de Sedgewick [47]. Un exemple d'arbre 2-3-4 est décrit dans la figure 4.11. La propriété fondamentale d'un tel arbre de recherche est la même que pour les noeuds binaires: tout noeud doit avoir une valeur supérieure ou égale à celles contenues dans ses sous-arbres gauches, et une valeur inférieure (ou égale) à celles de ses sous-arbres droits. Les noeuds ternaires contiennent 2 valeurs, la première doit être comprise entre les valeurs des sous-arbres gauches et du centre, la deuxième entre celles des sous-arbres du centre et de droite. On peut deviner aisément la condition pour les noeuds à 4 fils.

L'insertion d'un nouvel élément dans un arbre 2-3-4 se fait en éclatant tout noeud quaternaire que l'on rencontre comme décrit dans la figure 4.12. Ces opérations sont locales et ne font intervenir que le nombre de fils des noeuds. Ainsi, on garantit que l'endroit où on insère la nouvelle valeur n'est pas un noeud quaternaire, et il suffit de mettre la valeur dans ce noeud à l'endroit désiré. (Si la racine est quaternaire, on l'éclate en 3 noeuds binaires). Le nombre d'éclatements maximum peut être $\log N$ pour un arbre de N noeuds. Il a été mesuré qu'en moyenne très peu d'éclatements sont nécessaires.

Les arbres 2-3-4 peuvent être programmés en utilisant des arbres binaires bicolores. On s'arrange pour que chaque branche puisse avoir la couleur rouge ou noire (en trait gras sur notre figure 4.13). Il suffit d'un indicateur booléen dans chaque noeud pour dire si la branche le reliant à son père est rouge ou noire. Les noeuds quaternaires sont représentés par 3 noeuds reliés en noir. Les noeuds ternaires ont une double représentation possible comme décrit sur la figure 4.13. Les opérations d'éclatement se programment alors facilement, et c'est un bon exercice d'écrire la procédure d'insertion dans un arbre bicolore.

4.6 Programmes en Caml

```
(* Ajouter à un tas, avec une structure enregistrement *)
type 'a tas =
  { mutable cardinal: int;
    tas: 'a vect };;

let ajouter v t =
  t.cardinal <- t.cardinal + 1;
```

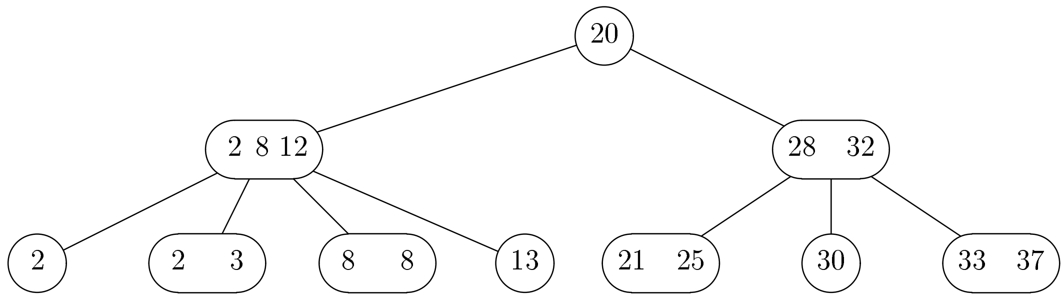


FIG. 4.11 – *Exemple d'arbre 2-3-4*

```
  let a = t.tas in
  let nTas = t.cardinal in
  let i = ref (nTas - 1) in
  if !i >= vect_length a
  then failwith "tas plein" else
  while !i > 0 && a.((!i - 1) / 2) <= v do
    a.(!i) <- a.((!i - 1) / 2);
    i := (!i - 1) / 2
  done;
  a.(!i) <- v;;
```

```
(* Ajouter à un tas, voir page 91 *)
let nTas = ref 0;;

let ajouter v a =
  incr nTas;
  let i = ref (!nTas - 1) in
  while !i > 0 && a.((!i - 1) / 2) <= v do
    a.(!i) <- a.((!i - 1) / 2);
    i := (!i - 1) / 2
  done;
  a.(!i) <- v;;

(* Maximum d'un tas, voir page 92 *)
let maximum t = t.tas.(0);;

(* Supprimer dans un tas, voir page 92 *)
let supprimer t =
  t.cardinal <- t.cardinal - 1;
  let a = t.tas in
  let nTas = t.cardinal in
  a.(0) <- a.(nTas);
  let i = ref 0
  and v = a.(0)
  and j = ref 0 in
  begin
    try
      while 2 * !i + 1 < nTas do
        j := 2 * !i + 1;
        if !j + 1 < nTas && a.(!j + 1) > a.(!j)
```

```
        then j := !j + 1;
        if v >= a.(!j) then raise Exit;
        a.(!i) <- a.(!j);
        i := !j
      done
    with Exit -> ()
  end;
  a.(!i) <- v;;
```

```
(* HeapSort, voir page 94 *)
let heapsort a =
  let n = vect_length a - 1 in
  let t = {cardinal = 0; tas = a} in
  for i = 0 to n do ajouter a.(i) t done;
  for i = n downto 0 do
    let v = maximum t in
    supprimer t;
    a.(i) <- v
  done;;
```

```
(* Déclaration d'un arbre binaire, voir page 95 *)
type 'a arbre = Vide | Noeud of 'a noeud
and 'a noeud =
  {contenu: 'a; filsG: 'a arbre; filsD: 'a arbre};;

(* Déclaration d'un arbre n-aire, voir page 96 *)

type 'a arbre = Vide | Noeud of 'a noeud
and 'a noeud = {contenu: 'a; fils: 'a arbre vect};;

(* Cas n-aire, les fils sont implémentés par une liste d'arbres. *)

type 'a arbre = Vide | Noeud of 'a noeud
and 'a noeud = {contenu: 'a; fils: 'a arbre list};;
```

```
(* Ajouter dans un arbre *)
let nouvel_arbre v a b =
  Noeud {contenu = v; filsG = a; filsD = b};;

let main () =
  let a5 =
    nouvel_arbre 12
      (nouvel_arbre 8 (nouvel_arbre 6 Vide Vide) Vide)
      (nouvel_arbre 13 Vide Vide) in
  nouvel_arbre 20
    (nouvel_arbre 3 (nouvel_arbre 3 Vide Vide) a5)
    (nouvel_arbre 25
      (nouvel_arbre 21 Vide Vide)
      (nouvel_arbre 28 Vide Vide));;
```

```
(* Impression d'un arbre, voir page 97 *)
#open "printf";;
```

```
let imprimer_arbre a =
  let rec imprimer1 a tab =
  match a with
    Vide -> ()
  | Noeud {contenu = c; filsG = fg; filsD = fd} ->
     printf "%3d     " c; imprimer1 fd (tab + 8);
     if fg <> Vide then
      printf "\n%s" (make_string tab ‘ ‘);
     imprimer1 fg tab;;
  in
    imprimer1 a 0; print_newline();;
```

```
(* Taille d'un arbre, voir page 98 *)
let rec taille = function
    Vide -> 0
  | Noeud {filsG = fg; filsD = fd; _} ->
     1 + taille fg + taille fd;;
```

```
(* Arbre de recherche, voir page 98 *)
let rec recherche v a =
  match a with
    Vide -> Vide
  | Noeud
     {contenu = c; filsG = fg; filsD = fd} ->
     if c = v then a else
     if c < v then recherche v fg
     else recherche v fd;;
```

```
(* Ajouter, purement fonctionnel, voir page 99 *)
let rec ajouter v a =
  match a with
    Vide -> nouvel_arbre v Vide Vide
  | Noeud
     {contenu = c; filsG = fg; filsD = fd} ->
     if v <= c
     then nouvel_arbre c (ajouter v fg) fd
     else nouvel_arbre c (ajouter v fd) fg;;
```

```
(* Ajouter avec effet de bord,
   voir page  \pageref{prog:recherche-arb-ajouter-fonctionnel}} *)
type 'a arbre = Vide | Noeud of 'a noeud
and 'a noeud =
  {mutable contenu: 'a;
   mutable filsG: 'a arbre;
   mutable filsD: 'a arbre};;

let nouvel_arbre v a b =
 Noeud {contenu = v; filsG = a; filsD = b};;
```

```
let rec ajouter v a =
  match a with
    Vide -> nouvel_arbre v Vide Vide
  | Noeud
     ({contenu = c; filsG = fg; filsD = fd}
       as n) ->
     if v <= c
     then n.filsG <- ajouter v fg
     else n.filsD <- ajouter v fd;
     a;;
```

```
(* Ajouter, purement fonctionnel avec un autre type d'arbres *)
type 'a arbre = Vide | Noeud of 'a arbre * 'a * 'a arbre;;

let nouvel_arbre v a b =
  Noeud (a, v, b);;

let rec ajouter v a =
  match a with
    Vide -> nouvel_arbre v Vide Vide
  | Noeud (fg, c, fd) ->
     if v <= c
     then nouvel_arbre c (ajouter v fg) fd
     else nouvel_arbre c (ajouter v fd) fg;;
```

```
(* Arbres AVL, voir page 100 *)

type 'a avl = Vide | Noeud of 'a noeud

and 'a noeud =
  { mutable balance: int;
    mutable contenu: 'a;
    mutable filsG: 'a avl;
    mutable filsD: 'a avl };;

let nouvel_arbre bal v a b =
  Noeud
   {balance = bal; contenu = v;
    filsG = a; filsD = b};;
```

```
#open "format";;

let rec print_avl = function
   Vide -> ()
 | Noeud
    {balance = bal; contenu = v;
     filsG = a; filsD = b} ->
    open_box 1; print_string "(";
    print_int bal; print_string ": ";
    print_int v; print_space();
    print_avl a; print_space(); print_avl b;
    print_cut(); print_string ")";
    close_box();;
```

```
install_printer "print_avl";;
```

```
(* Rotation dans un AVL, voir page 102 *)
let rotD a =
  match a with
    Vide -> failwith "rotD"
  | Noeud
      ({balance = bB; contenu = v;
        filsG = A; filsD = c} as nB) as B ->
     match A with
       Vide -> failwith "rotD"
     | Noeud
         ({balance = bA; contenu = v;
           filsG = a; filsD = b} as nA) ->
          nA.filsD <- B;
          nB.filsG <- b;
          let bBnew = bB + 1 - min 0 bA in
          let bAnew = bA + 1 + max 0 bBnew in
          nA.balance <- bAnew;
          nB.balance <- bBnew;
          A;;
```

```
(* Rotation dans un AVL, voir page 102 *)
let rotG a =
  match a with
    Vide -> failwith "rotG"
  | Noeud
      ({balance = bA; contenu = v;
        filsG = c; filsD = B} as nA) as A ->
     match B with
     | Vide -> failwith "rotG"
     | Noeud
         ({balance = bB; contenu = v;
           filsG = a; filsD = b} as nB) ->
          nA.filsD <- a;
          nB.filsG <- A;
          let bAnew = bA - 1 - max 0 bB in
          let bBnew = bB - 1 + min 0 bAnew in
          nA.balance <- bAnew;
          nB.balance <- bBnew;
          B;;
```

```
(* Ajout dans un AVL, voir page 100 *)
let rec ajouter v a =
 match a with
   Vide -> (nouvel_arbre 0 v Vide Vide, 1)
 | Noeud
    ({balance = bal; contenu = c;
      filsG = fg; filsD = fd} as noeud) ->
    let diff =
```

```
  if v <= c then begin
    let (a, incr) = ajouter v fg
    in
    noeud.balance <- noeud.balance - incr;
    noeud.filsG <- a;
    incr
  end else begin
     let (a, incr) = ajouter v fd
     in
     noeud.balance <- noeud.balance + incr;
     noeud.filsD <- a;
     incr
   end in
if diff <> 0 && noeud.balance <> 0 then
  if noeud.balance < -1 then begin
    match fg with
      Vide -> failwith "Vide"
    | Noeud {balance = b; _} ->
       if b < 0 then (rotD a, 0)
       else begin
         noeud.filsG <- rotG fg; (rotD a, 0)
       end
  end else
  if noeud.balance > 1 then begin
    match fd with
      Vide -> failwith "Vide"
    | Noeud {balance = b; _} ->
      if b > 0 then (rotG a, 0)
      else begin
        noeud.filsD <- rotD fd; (rotG a, 0)
      end
  end
  else (a, 1)
else (a, 0);;
```

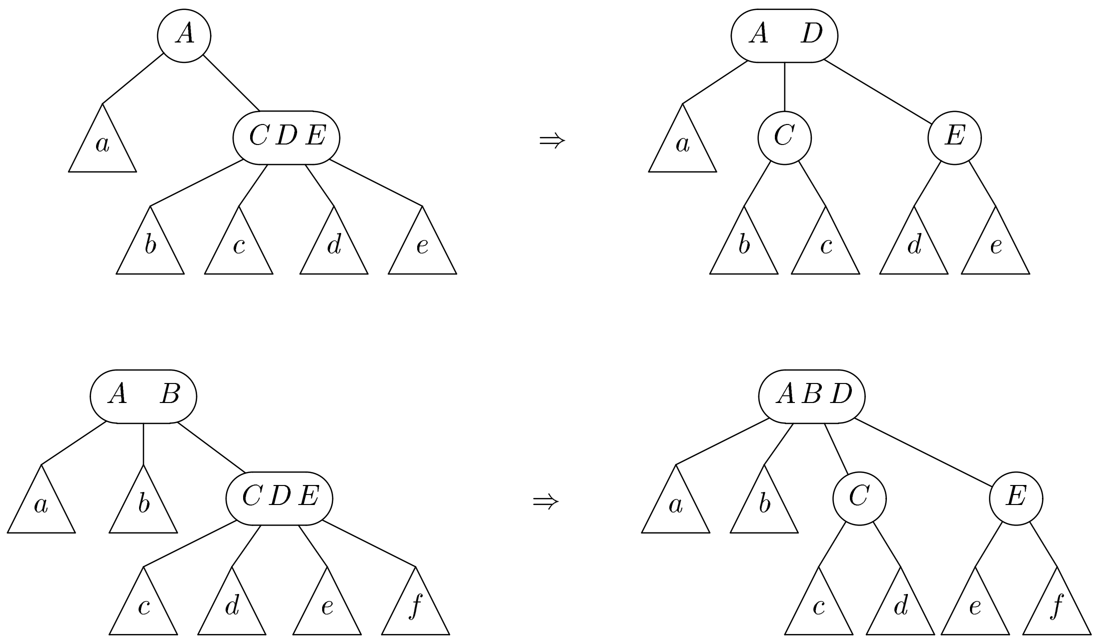


FIG. 4.12 – *Eclatement d'arbres 2-3-4*

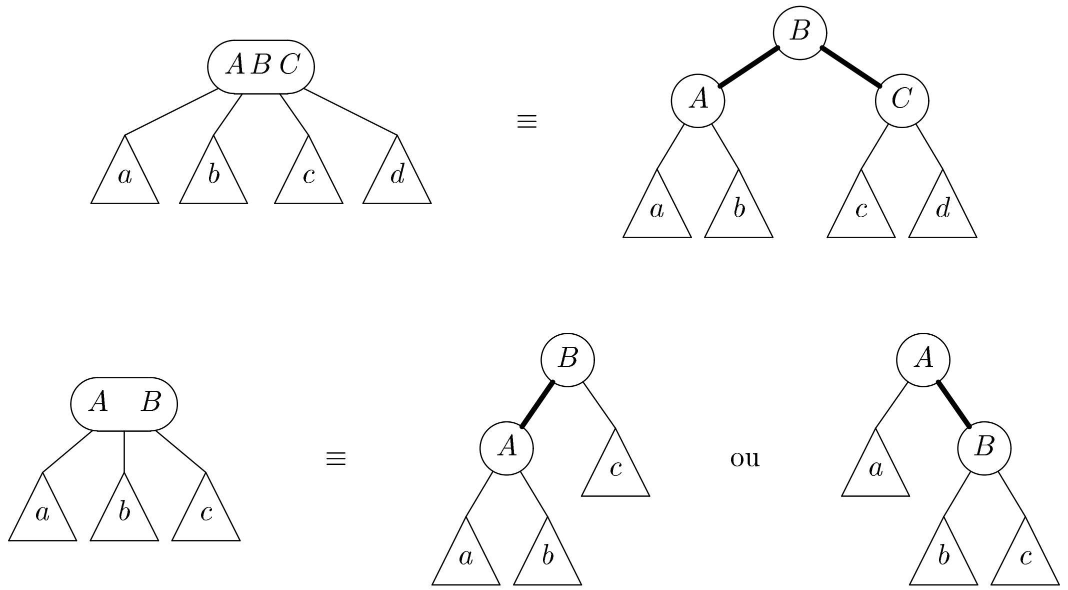


FIG. 4.13 – *Arbres bicolores*

Chapitre 5

Graphes

La notion de graphe est une structure combinatoire permettant de représenter de nombreuses situations 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.

5.1 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 *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 ensemble des* arcs *de G.*

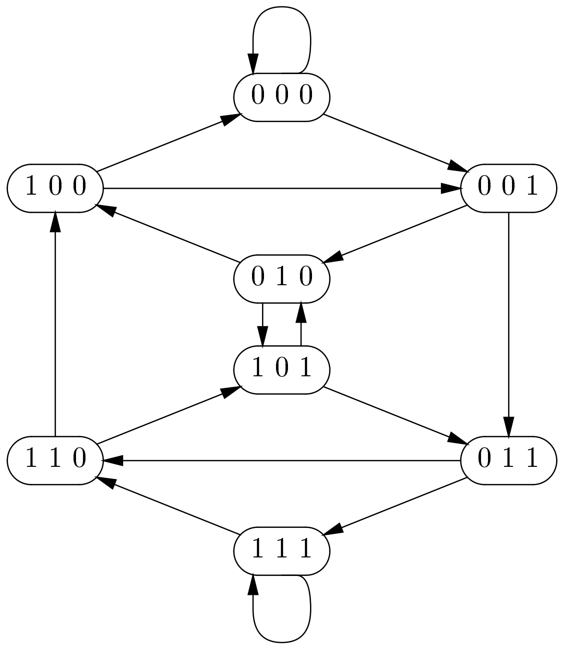


FIG. 5.1 – *Le graphe de De Bruijn pour* $k = 3$

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

$$or(a) = x; \quad ext(a) = y$$

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 $(x, y) \in A$, x est alors un *prédécesseur* de y.

Définition 2 *Un chemin f du graphe $G = (X, A)$ est une suite finie d'arcs $a_1, a_2, \ldots, a_p$ telle que:*

$$\forall i, 1 \leq i < p \qquad or(a_{i+1}) = ext(a_i).$$

L'*origine* d'un chemin f, notée $or(f)$ 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 $or(f) = ext(f)$ est appelé un *circuit*.

Exemple 1 *Graphes de De Bruijn*

Les sommets d'un tel graphe sont les suites de longueur k formées de symboles 0 ou 1, un arc joint la suite f à la suite g si

$$f = xh, \qquad g = hy$$

où x et y sont des symboles (0 ou 1) et où h est une suite quelconque de $k-1$ symboles.

Exemple 2 *Graphes des diviseurs*

Les sommets sont les nombres $\{2, 3, \ldots, n\}$, un arc joint p à q si p divise q.

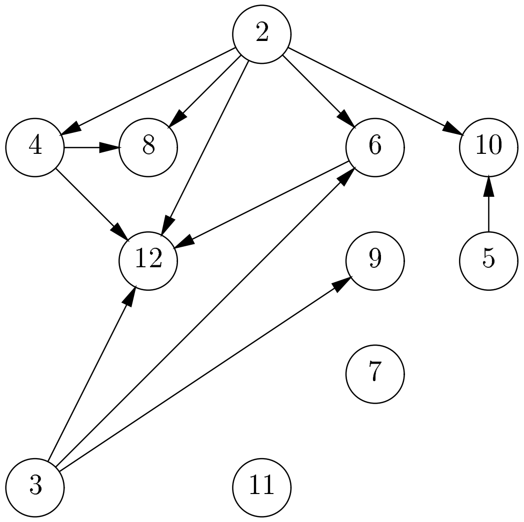


FIG. 5.2 – *Le graphe des diviseurs, $n = 12$*

5.2 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:

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

M est une matrice carrée $n \times n$ dont les coefficients sont 0 et 1 telle que:

$$M_{i,j} = 1 \quad \text{si} \quad (x_i, x_j) \in A, \quad M_{i,j} = 0 \quad \text{si} \quad (x_i, x_j) \notin A$$

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

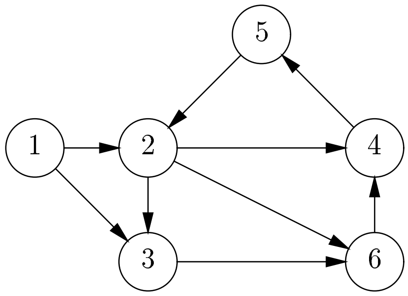
```
class GrapheMat {

  int m[][];  (* la matrice M d'adjacence, *)
  int nb;     (* n est le nombre de sommets *)

  GrapheMat (int n) {
      nb = n;
      m = new int[n][n];
  }
}
```

Un intérêt de cette 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 2 *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.*

Preuve On effectue une récurrence sur p. Pour $p = 1$ le résultat est immédiat car un chemin de longueur 1 est un arc du graphe. Le calcul de M^p, pour $p > 1$ donne:

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



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

FIG. 5.3 – *Un exemple de graphe et sa matrice d'adjacence*

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. □

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

```
static boolean existeChemin (int i, int j, GrapheMat g) {

    int n = g.nb;
    int m[][] = g.m;
    int u[][] = copy(m);
    int v[][] = copy(m);

    for (int k = 1; u[i][j] == 0 && k < n; ++k) {
        multiplier (v, v, m);
        additionner (u, u, v);
    }
    return u[i][j] != 0;
}
```

Dans cet algorithme, les procédures `multiplier(c, a, b)` et `additonner(c, a, b)` sont respectivement des procédures qui multiplient et ajoutent les deux matrices $n \times n$ `a` et `b` pour obtenir la matrice `c`.

Remarques

1. L'algorithme utilise le fait, facile à démontrer, que l'existence d'un chemin d'origine x et d'extrémité y implique celle d'un tel chemin ayant une longueur inférieure au nombre total de sommets du graphe.

2. 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 être amené à effectuer n produits de matrices. La recherche du meilleur algorithme possible

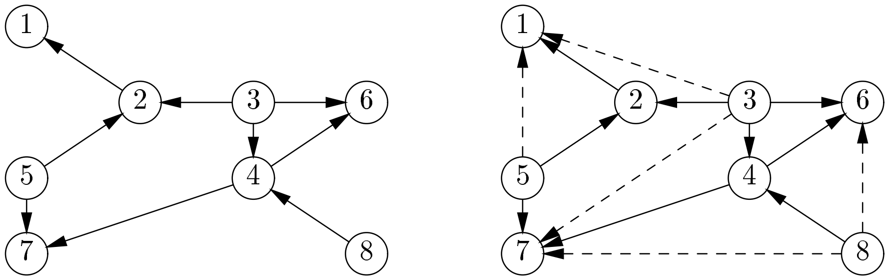


FIG. 5.4 – *Un graphe et sa fermeture transitive*

pour le calcul du produit de deux matrices a été très intense ces dernières années et plusieurs améliorations de l'algorithme élémentaire demandant n^3 multiplications ont été successivement trouvées, et il est rare qu'une année se passe sans que quelqu'un n'améliore la borne. Coppersmith et Winograd ont ainsi proposé un algorithme en $O(n^{2.5})$; mais ceci est un résultat de nature théorique car la programmation de l'algorithme de Coppersmith et Winograd est loin d'être aisée et l'efficacité espérée n'est atteinte que pour des valeurs très grandes de n. Il est donc nécessaire de construire d'autres algorithmes, faisant intervenir des notions différentes.

3. Cet algorithme construit une matrice (notée ici `u`) qui peut être utilisée chaque fois que l'on veut tester s'il existe un chemin entre deux sommets (x_i et y_i). Dans le cas où on se limite à la recherche de l'existence d'un chemin entre deux sommets donnés (et si ceci ne sera fait qu'une seule fois) on peut ne calculer qu'une ligne de la matrice, ce qui diminue notablement la complexité.

5.3 Fermeture transitive

La fermeture transitive d'un graphe $G = (X, A)$ est la relation transitive minimale contenant la relation (X, A), 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 de répondre aux questions concernant l'existence de chemins entre x et y dans G et ceci pour tout couple de sommets x, y. Ce calcul complet n'est pas vraiment utile s'il s'agit de répondre un petit nombre de fois à des questions sur l'existence de chemins entre des couples de sommets, on utilise alors des algorithmes qui seront décrits dans les paragraphes suivants. Par contre lorsque l'on s'attend à avoir à répondre de nombreuses fois à ce type de question il est préférable de calculer au préalable (X, A^*), la réponse à chaque question est alors immédiate par simple consultation d'un des coefficients de la matrice d'adjacence de G^*. On dit que l'on effectue un *prétraitement*, opération courante en programmation dans maintes applications, ainsi le tri des éléments d'un fichier peut aussi être considéré comme un prétraitement en vue d'une série de consultations du fichier, le temps mis pour trier le fichier est récupéré ensuite dans la rapidité de la consultation. Le calcul de la fermeture transitive d'un graphe se révèle très utile par exemple, dans certains compilateurs-optimiseurs: un graphe est associé à chaque procédure d'un programme, les sommets de ce graphe représentent les variables de la procédure et un arc entre la variable `a` et la variable `b` indique qu'une instruction de calcul de `a` fait apparaître `b` dans son membre droit. On l'appelle souvent *graphe de dépendance*. La fermeture transitive de ce graphe donne ainsi toutes les variables nécessaires directement ou indirectement au calcul de `a`; cette information est utile lorsque l'on veut minimiser la quantité de calculs à effectuer

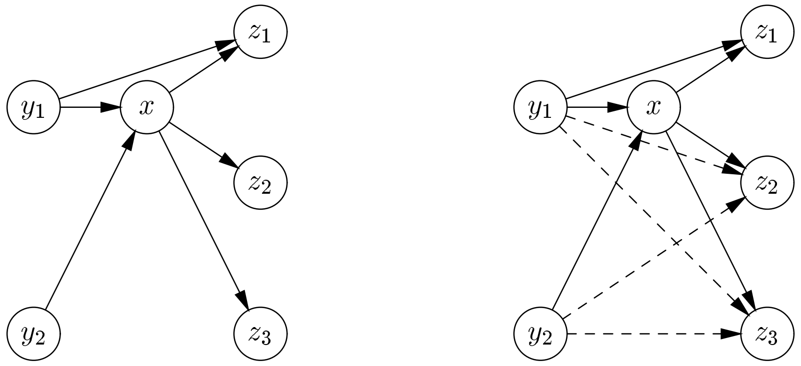


FIG. 5.5 – *L'effet de l'opération Φ_x: les arcs ajoutés sont en pointillé*

en machine pour l'exécution du programme.

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. Plus formellement on pose :

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

Cette opération satisfait les deux propriétés suivantes:

Proposition 1 *Pour tout sommet x on a*

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

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

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

Preuve 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 raison de 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 $(x, y), (y, v) \in A$, ceci implique $(u, y) \in \Phi_x(A)$ et $(u, v) \in \Phi_y(\Phi_x(A))$
$\square$

Proposition 2 *La fermeture transitive A^* est donnée par :*

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

Preuve 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 suivant pour le calcul de la fermeture transitive d'un graphe, il est en général attribué à Roy et Warshall:

```
static void phi (GrapheMat g, int x) {

  for (int i = 0; i < g.nb; ++i)
  if (g.m[i][x] == 1)
      for (int j = 0; j < g.nb; ++j)
          if (g.m[x][j] == 1)
              g.m[i][j] = 1;
}

static public void fermetureTransitive (GrapheMat g) {

  for (int k = 0; k < g.nb; ++k)
      phi(g, k);
 }
```

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

5.4 Listes de successeurs

Une façon plus compacte de représenter un graphe consiste à associer à chaque sommet x la liste de ses successeurs. Ceci peut se faire, par exemple, à l'aide d'un tableau à double indice que l'on notera *Succ*. On suppose que les sommets sont numérotés de 1 à n, alors pour un sommet x et un entier i, $Succ[x, i]$ est le $i$ème successeur de x. Cette représentation est utile pour obtenir tous les successeurs d'un sommet x. Elle permet d'y accéder en un nombre d'opérations égal au nombre d'éléments de cet ensemble et non pas, comme c'est le cas dans la matrice d'adjacence, au nombre total de sommets. Ainsi si dans un graphe de 20000 sommets chaque sommet n'a que 5 successeurs l'obtention de tous les successeurs de x se fait en consultant 4 ou 5 valeurs au lieu des 20000 tests à effectuer dans le cas des matrices. L'utilisation d'un symbole supplémentaire noté ω, signifiant "indéfini" et n'appartenant pas à X permet une gestion plus facile de la fin de liste. On le place à la suite de tous les successeurs de x pour indiquer que l'on a terminé la liste. Ainsi
$Succ[x, i] = y \quad \in X$ signifie que y est le $i$ème successeur de x
$Succ[x, i] = \omega$ signifie que x a $i - 1$ successeurs.

Le graphe donné figure 5.3 plus haut admet alors la représentation par liste de successeurs suivante:

$$\begin{array}{llllll}
1: & 2 & 3 & \omega & & \\
2: & 4 & 3 & 6 & \omega \\
3: & 6 & \omega & & \\
4: & 5 & \omega & & \\
5: & 2 & \omega & & \\
6: & 4 & \omega & &
\end{array}$$

Les déclarations Java correspondantes peuvent être alors les suivantes:

```
class GrapheSucc{

  int succ[][];
  int nb;
  final static int Omega = -1;

  GrapheSucc (int n) {
      nb = n;
      succ = new int[n][n];
  }
```

Le parcours de la liste des successeurs y d'un sommet x s'effectue alors à l'aide de la suite d'instructions suivantes, et on retrouvera cette suite d'instructions comme brique de base de beaucoup de constructions d'algorithmes sur les graphes :

```
for (k = 0; succ[x,k] != Omega; ++k) {
    int y = succ[x,k];
    // Traiter y
}
```

On peut transformer la matrice d'adjacence d'un graphe en une structure de liste de successeurs par l'algorithme suivant :

```
GrapheSucc (GrapheMat g) {

  int nbMaxSucc;
  nb = g.nb;
  for (int i = 0; i < nb ; ++i) {
      nbMaxSucc = 0;
      for (int j = 0; j < nb ; ++j)
          if (g.m[i][j] != 0)
              nbMaxSucc = Math.max (nbMaxSucc, j);
  }
  succ = new int[nb][nbMaxSucc + 1];
  for (int i = 0; i < nb ; ++i) {
      int k = 0;
      for (int j = 0; j < nb  ; ++j)
          if (g.m[i][j] != 0)
              succ[i][k++] = j;
      succ[i][k] = Omega;
  }
}
```

Remarque La structure de liste de successeurs peut être remplacée par une structure de liste chaînée. Cette programmation permet de gagner en place mémoire en évitant de déclarer un nombre de successeurs maximum pour chacun des sommets. Elle permet aussi de diminuer le nombre d'opérations chaque fois que l'on effectue des opérations d'ajout et de suppression de successeurs. Cette notion peut être omise en première lecture, en particulier par ceux qui ne se sentent pas très à l'aise dans le maniement des pointeurs. Dans toute la suite, les algorithmes sont écrits avec la structure matricielle `succ[x,i]`. Un simple jeu de traduction permettrait de les transformer en programmation par pointeurs; on utilise les structures de données suivantes :

```
class GrapheListe{

  Liste listeSucc[];
  int nb;
```

```
    GrapheListe (GrapheSucc g) {

      nb = g.nb;
      listeSucc = new Liste[g.nb];
      for (int x = 0; x < g.nb ; ++x) {
          listeSucc[x] = null;
          for (int k = 0; g.succ[x][k] != GrapheSucc.Omega; ++k) {
              int y = g.succ[x][k];
              listeSucc[x] = Liste.ajouter(y, listeSucc[x]);
          }
      }
    }
}
```

La transformation de la forme matricielle *Succ* en une structure de liste chaînée par le constructeur de la classe donné ci dessus, on peut noter que celui-ci inverse l'ordre dans lequel sont rangés les successeurs d'un sommet, ceci n'a pas d'importance dans la plupart des cas.

5.5 Arborescences

Définition 3 *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,*

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

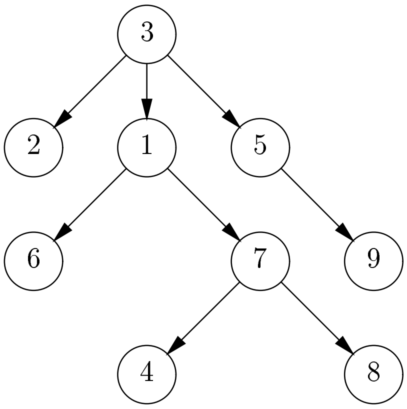
tels que:

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

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 prédécesseur et un seul, ceci implique:

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

La différence entre une arborescence et un arbre (voir chapitre 4) 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. On se sert depuis fort longtemps des arborescences pour représenter des arbres généalogiques aussi 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 $y_0, y_1, \ldots y_{p-1}, y_p$, où $p \geq 0$, formant le chemin de r à $x = y_p$ est appelé ensemble des *ancêtres* de x, les successeurs de x sont aussi appelés ses *fils*. L'ensemble des sommets 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. Une arborescence est avantageusement représentée par le vecteur `pere` qui à chaque sommet différent de la racine associe son père. Il est souvent commode dans la programmation des algorithmes sur les arborescences de considérer que la racine de l'arborescence est elle-même son père, c'est la convention que nous adopterons dans la suite.



i :	1	2	3	4	5	6	7	8	9
$pere(i)$:	3	3	3	7	3	1	1	7	5

FIG. 5.6 – *Une arborescence et son vecteur pere*

La transformation des listes de successeurs décrivant une arborescence en le vecteur `pere` s'exprime très simplement:

```
class Arbo {

  int pere[];
  final static int Omega = -1;

  Arbo (GrapheSucc g, int r) {

    pere = new int[g.nb];
    pere[r] = r;
    for (int x = 0; x < g.nb ; ++x)
        for (int k = 0; g.succ[x][k] != GrapheSucc.Omega; ++k)
            pere[g.succ[x][k]] = x;
  }
}
```

Dans la suite, on suppose que l'ensemble des sommets X est l'ensemble des entiers compris entre 1 et n, 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 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 3 *Pour toute arborescence (X, A, r) il existe une re-numérotation des éléments de X qui la rend préfixe.*

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

- La racine est numérotée 1.

- Un des fils x_1 de la racine est numéroté 2.

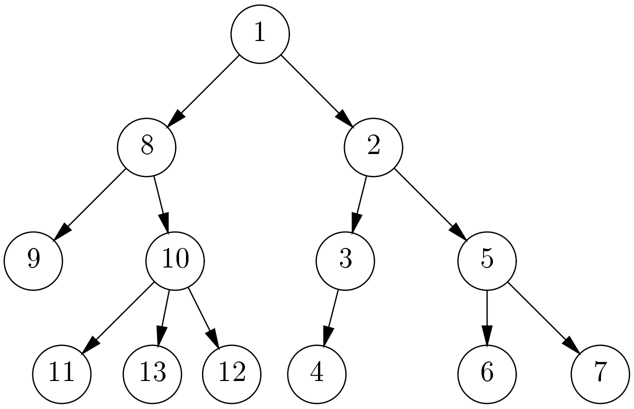


FIG. 5.7 – *Une arborescence préfixe*

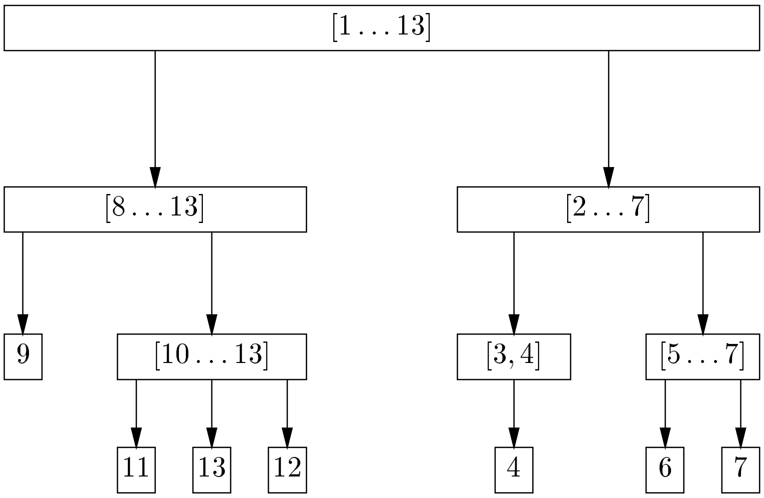


FIG. 5.8 – *Emboitement des descendants dans une arborescence préfixe*

- L'arborescence des descendants de x_1 est numérotée par appels récursifs de l'algorithme on obtient ainsi des sommets numérotés de 2 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 preuve de ce que la numérotation obtenue est préfixe se fait par récurrence sur le nombre de sommets de l'arborescence et utilise le caractère récursif de l'algorithme. $\square$

L'algorithme qui est décrit dans la preuve ci-dessus peut s'écrire, on suppose que l'arborescence est représentée par une matrice `Succ` de successeurs la re-numérotation se fait par un vecteur `numero` et `r` est la racine de l'arborescence.

```
static int[] numPrefixe (int r, GrapheSucc g) {

  int numero[] = new int[g.nb];
  numPrefixe1 (r, g, numero, 0);
  return numero;
}

static void numPrefixe1 (int x, GrapheSucc g, int numero[], int num) {
  numero[x] = num++;
  for (int k = 0; g.succ[x][k] != GrapheSucc.Omega; ++k)
      numPrefixe1 (g.succ[x][k], g, numero, num);
}
```

5.6 Arborescence des plus courts chemins.

Le parcours d'un graphe $G = (X, A)$, c'est à dire la recherche de chemins entre deux sommets revient au calcul de certaines arborescences dont l'ensemble des sommets et des arcs sont inclus dans X et A respectivement. Nous commençons par décrire celle des plus courts chemins.

Définition 4 *Dans un graphe $G = (X, A)$, pour chaque sommet x, une arborescence des plus courts chemins (Y, B) de racine x est une arborescence telle que:*

- *Un sommet y appartient à Y si et seulement si 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).*

L'existence de l'arborescence des plus courts chemins est une conséquence de la remarque suivante:

Remarque Si $a_1, a_2, \ldots, a_p$ est un plus court chemin entre $x = or(a_1)$ et $y = ext(a_p)$ alors, pour tout i tel que $1 \leq i \leq p$, $a_1, a_2, \ldots, a_i$ est un plus court chemin entre x et $ext(a_i)$.

Théorème 3 *Pour tout graphe $G = (X, A)$ et tout sommet x de G il existe une arborescence des plus courts chemins de racine x.*

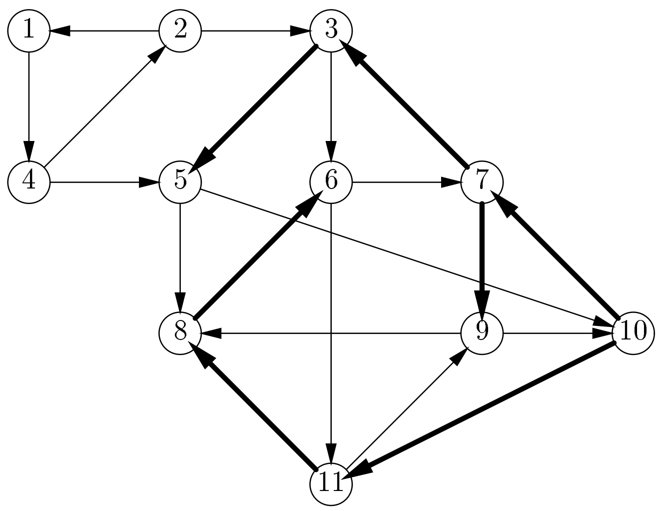


FIG. 5.9 – *Une arborescence des plus courts chemins de racine* 10

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

- $Y_0 = \{x\}$.
- 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 alors une arborescence de par sa construction même, le fait qu'il s'agisse de l'arborescence des plus courts chemins résulte de la remarque ci-dessus. □

La figure 5.9 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:

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

La preuve de ce théorème, comme c'est souvent le cas en mathématiques discrètes se transforme très simplement en un algorithme de construction de l'arborescence (Y, B). Cet algorithme est souvent appelé algorithme de *parcours en largeur* ou *breadth-first search*, en anglais. Nous le décrivons ci dessous, il utilise une file avec les primitives associées: ajout, suppression, valeur du premier, test pour savoir si la file est vide. La file gère les ensembles Y_i. On ajoute les éléments des Y_i successivement dans la file qui contiendra 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 à l'aide du prédicat `(pere[y] = omega)`.

```
static Arbo arbPlusCourt (GrapheSucc g, int x0) {
```

```
  Arbo a = new Arbo(g.nb, x0);

  File f = new File.vide();
  File.ajouter(x0, f);
  while (!File.estVide(f)) {
      int x = File.valeur(f);
      File.supprimer(f);
      for (int k = 0; g.succ[x][k] != GrapheSucc.Omega; ++k) {
          int y = g.succ[x][k];
          if (a.pere[y] == Omega) {
              a.pere[y] = x;
              File.ajouter(y, f);
          }
      }
  }
 return a;
}
```

5.7 Arborescence de Trémaux

Un autre algorithme très ancien de parcours dans un graphe a été mis au point par un ingénieur du siècle dernier, Trémaux, dont les travaux sont cités dans un des premiers livres sur les graphes dû à Sainte Lagüe. Son but étant 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* 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 dans le paragraphe suivant. 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 procédure 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. On peut en programmation représenter ce fil d'Ariane par une pile.

Ceci donne deux versions de l'algorithme que nous donnons ci-dessous.

```
static void tremauxRec (int x, GrapheSucc g, Arbo a) {

  for (int k = 0; g.succ[x][k] != Omega; ++k) {
      int y = g.succ[x][k];
      if (a.pere[y] == Omega) {
          a.pere[y] = x;
          tremauxRec(y, g, a);
      }
  }
}
```

Le calcul effectif de l'arborescence de Trémaux de racine x s'effectue en initialisant le vecteur `pere` par le constructeur de la classe des arborescences.

```
static Arbo tremaux (int x, GrapheSucc g) {

  Arbo a = new Arbo (g.nb, x0);
```

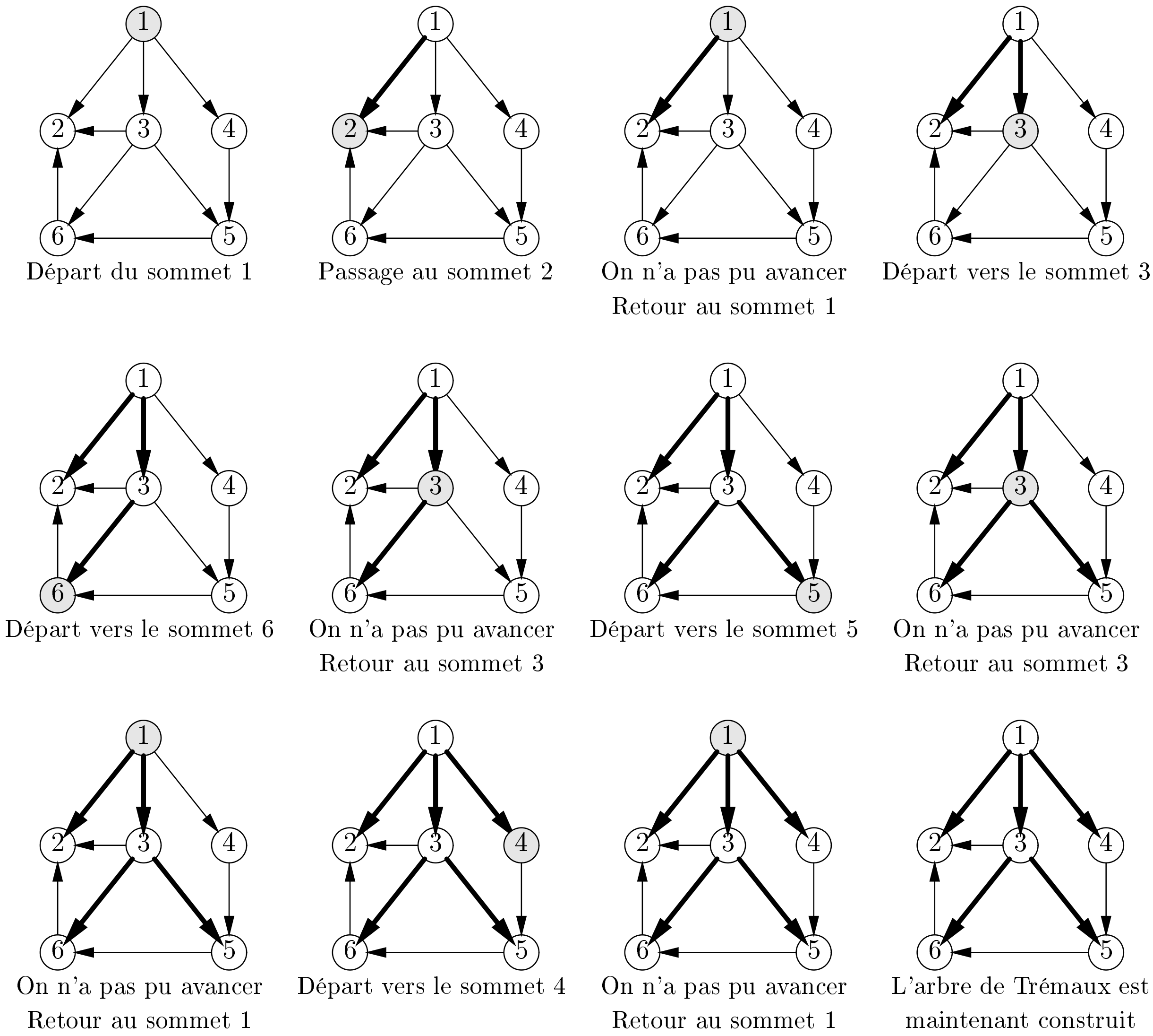


FIG. 5.10 – *Exécution de l'algorithme de Trémaux*

```
    tremauxRec(x0, g, a);
    return a;
  }
```

La figure 5.10 explique l'exécution de l'algorithme sur un exemple, les appels de la procédure sont dans l'ordre:

```
tremauxRec(1)
     tremauxRec(2)
     tremauxRec(3)
         tremauxRec(6)
         tremauxRec(5)
     tremauxRec(4)
```

La procédure non récursive ressemble fortement à celle du calcul de l'arborescence des plus courts chemins à cela près que l'on utilise une pile et non une file et que l'on enlève le sommet courant de la pile une fois que l'on a visité tous ses successeurs.

```
static void tremauxPile (int x, GrapheSucc g, Arbo a) {

    int y, z;
    Pile p = new Pile();
    Pile.ajouter (x, p);
    a.pere[x] = x;
  boucle:
    while ( !Pile.estVide(p) ) {
        y = Pile.valeur (p);
        for (int k = 0; g.succ[y][k] != Omega; ++k) {
            z = g.succ[y][k];
            if (a.pere[z] == Omega)
                a.pere[z] = y;
                Pile.ajouter (z, p);
                continue boucle;
        }
        Pile.supprimer (p);
    }
}
```

Remarques

1 L'ensemble des sommets atteignables à partir du sommet x est formé des sommets tels que `Pere[y]` $\neq$ `Omega` à la fin de l'algorithme, on a donc un algorithme qui répond à la question `existeChemin(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 que l'utilisation des matrices.

2 L'algorithme non récursif tel qu'il est écrit n'est pas efficace car il lui arrive de parcourir plusieurs fois les successeurs d'un même sommet; pour éviter cette recherche superflue, il faudrait empiler en même temps qu'un sommet le rang du successeur que l'on est en train de visiter et incrémenter ce rang au moment du dépilement. Dans ce cas, on a une bien meilleure efficacité, mais la programmation devient inélégante et le programme difficile à lire; nous préférons de loin la version récursive.

L'ensemble des arcs du graphe $G = (X, A)$ qui ne sont pas dans l'arborescence de Trémaux (Y, T) de racine 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.

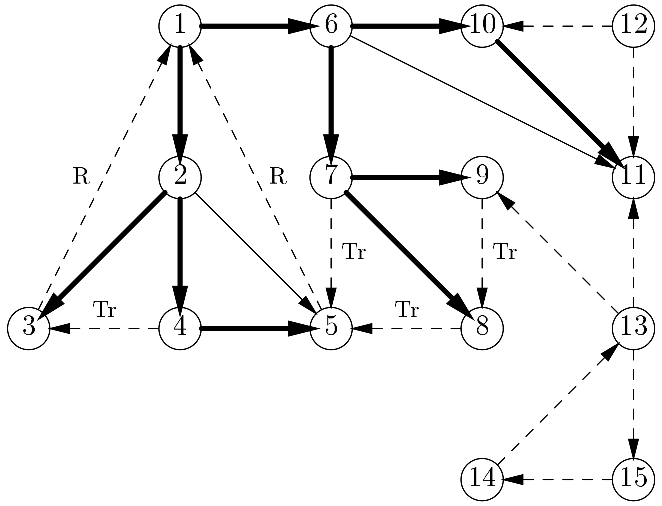


FIG. 5.11 – *Les arcs obtenus par Trémaux*

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), 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).

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).

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

Sur la figure 5.11, on a dessiné un graphe et les différentes sortes d'arcs y sont représentés par des lignes particulières. Les arcs de l'arborescence sont en traits gras, les arcs de descente en traits normaux (sur cet exemple, il y en a deux), les arcs dont l'origine n'est pas dans Y sont dessinés en pointillés, de même que les arcs de retour ou transverses qui sont munis d'une étiquette permettant de les reconnaître, celle ci est R pour les arcs de retour et Tr pour les arcs transverses. 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.

5.8 Composantes fortement connexes

Dans ce paragraphe, nous donnons une application du calcul de l'arbre de Trémaux, l'exemple a été choisi pour montrer l'utilité de certaines constructions ingénieuses d'algorithmes sur les graphes. La première sous-section expose le problème et donne une solution simple mais peu efficace, les autres sous-sections décrivent l'algorithme ingénieux de Tarjan. Il s'agit là de constructions combinatoires qui doivent être considérées comme un complément de lecture pour amateurs.

5.8.1 Définitions et algorithme simple

Définition 5 *Soit $G = (X, A)$ un graphe, on note $\equiv_G$ la relation 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*

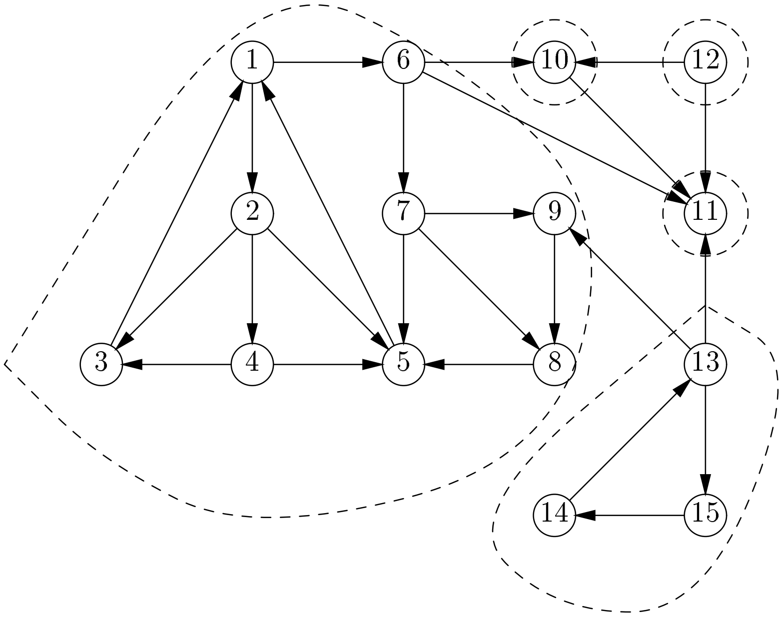


FIG. 5.12 – *Composantes fortement connexes du graphe de la figure 5.11*

à x.

Celle-ci est une relation d'équivalence. Sa définition même entraîne la symétrie et la réflexivité. 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. Les classes de cette relation d'équivalence sont appelées les *composantes fortement connexes* de G. La composante fortement connexe contenant le sommet u sera notée $C(u)$ dans la suite.

Le graphe de la figure 5.12 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*. Savoir si un graphe est fortement connexe est particulièrement important par exemple dans le choix de sens uniques pour les voies de circulation d'un quartier.

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'arborescence 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 dans les sections suivantes, 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.

5.8.2 Utilisation de l'arborescence de Trémaux

On étudie tout d'abord la numérotation des sommets d'un graphe que l'on obtient par l'algorithme de Trémaux. On la rappelle ici en y ajoutant une instruction de numérotation.

```
static int num = 0;

static void tremauxRec (int x, GrapheSucc g, Arbo a) {

  numero [x] = num++;
  for (int k = 0; g.succ[x][k] != Omega; ++k) {
      int y = g.succ[x][k];
      if (a.pere[y] == Omega) {
          a.pere[y] = x;
          tremauxRec(y, g, a);
      }
  }
}
```

Proposition 4 *Si on numérote les sommets au fur et à mesure de leur rencontre au cours de l'algorithme de Trémaux, on obtient une arborescence préfixe (Y, T), un arc (u, v) qui n'est pas dans T mais dont l'origine u et l'extrémité v sont dans Y est un arc de descente si $num(u) < num(v)$ et un arc de retour ou un arc transverse si $num(u) > num(v)$.*

On supposera dans la suite que les sommets sont numérotés de cette façon, ainsi lorsqu'on parlera du sommet i, cela voudra dire le $i$ème sommet rencontré lors du parcours de Trémaux et cela évitera certaines lourdeurs d'écriture. La proposition ci-dessus se traduit alors par le fait suivant:

Si v est un descendant de u dans (Y, T) et si un sommet w satisfait :

$$u \leq w \leq v$$

w est aussi un descendant de u dans cette arborescence.

Les liens entre arborescence de Trémaux (Y, T) de racine x et les composantes fortement connexes sont dus à la proposition suivante, que l'on énoncera après avoir donné une définition.

Définition 6 *Une sous-arborescence (Y', T') de racine r' d'une arborescence (Y, T) de racine r est constituée par des sous-ensembles Y' de Y et T' de T formant une arborescence de racine r'.*

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

Proposition 5 *Soit $G = (X, A)$ un graphe, $x \in X$, et (Y, T) une arborescence de Trémaux de racine x. Pour tout sommet u de Y, la composante fortement connexe $C(u)$ de G contenant u est une sous-arborescence de (Y, T).*

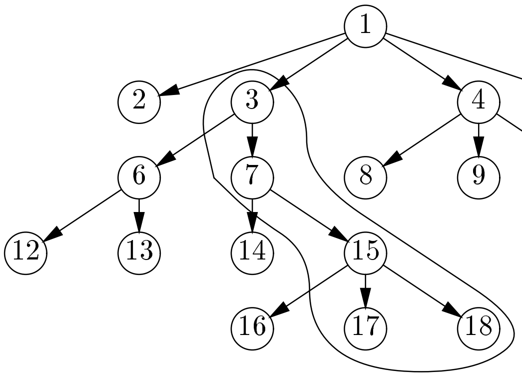


FIG. 5.13 – *Un exemple de sous-arborescence*

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

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 choisissons pour u_0 le sommet de plus petit numéro de $C(u)$ et montrons que tout v de $C(u)$ est un descendant de u_0 dans (Y,T). Supposons le contraire, v étant 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) 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 :

$$u_0 \leq w \leq w'$$

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

5.8.3 Points d'attache

Une notion utile pour le calcul des composantes fortement connexe est la notion de point d'attache dont la définition est donnée ci-dessous. Rappelons que l'on suppose les sommets numérotés dans l'ordre où on les rencontre par la procédure de Trémaux.

Définition 7 *Etant donné un graphe $G = (X,A)$, un sommet x de G et l'arborescence de Trémaux (Y,T) de racine x, le point d'attache $at(y)$ d'un sommet y de Y est le sommet de plus petit numéro extrémité d'un chemin de $G = (X,A)$, d'origine y et contenant au plus un arc (u,v) tel que $u > v$ (c'est à dire un arc de retour ou un arc transverse). On suppose que le chemin vide d'origine et extrémité égale à y est un tel chemin ainsi:*

$$at(y) \leq y$$

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

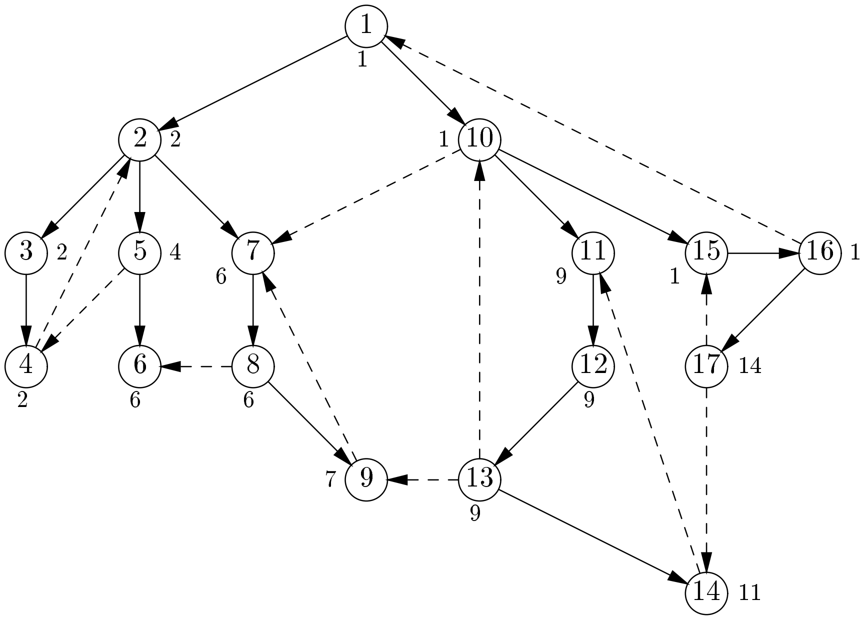


FIG. 5.14 – *Les points d'attaches des sommets d'un graphe*

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 5.14, 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; le point d'attache est indiqué en petit caractère à coté du sommet en question.

Le calcul des points d'attache se fait à l'aide d'un algorithme récursif qui est basé sur la proposition suivante, dont la preuve est immédiate:

Proposition 6 *Le point d'attache $at(y)$ du sommet y est le plus petit parmi les sommets suivants:*

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

L'algorithme est ainsi une adaptation de l'algorithme de Trémaux, il calcule `at[u]` en utilisant la valeur des `at[v]` pour tous les successeurs `v` de `u`.

```
static int pointAttache1 (int x, GrapheSucc g, int[] at) {

  int y, z;
  int m = x;
  at[x] = x;
  for (int k = 0; g.succ[x][k] != Omega; ++k) {
      y  = g.succ[x][k];
      if (at[y] == Omega)
          z = pointAttache1(y, g, at);
      else
          z = y;
      m = Math.min(m, z);
```

```
    }
    at[x] = mi;
    return mi;
  }

  static int pointAttache (int x, GrapheSucc g, int[] at) {

    for (x = 0; x < g.nb; ++x)
        at[x] = Omega;
    at[x] = PointAttache1 (x, g, at);
  }
```

Le calcul des composantes fortement connexes à l'aide des $at(u)$ est une conséquence du théorème suivant:

Théorème 4 *Si u est un sommet de Y satisfaisant:*

- *(i) $u = at(u)$*
- *(ii) Pour tout descendant v de u dans (Y, T) on a $at(v) < v$*

Alors, l'ensemble $desc(u)$ des descendants de u dans (Y, T) forme une composante fortement connexe de G.

Preuve 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) 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) 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) implique $v_2 \in desc(u)$. $\square$

Remarques

1. Il existe toujours un sommet du graphe satisfaisant les conditions de la proposition ci-dessus. En effet, si x est la racine de (Y, T) on a $at(x) = x$. Si x satisfait *(ii)*, alors 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.

2. 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.

3. Sur la figure 5.14, 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 $2, 6, 1$. La première composante trouvée se compose du sommet 6 uniquement, il est supprimé et le sommet 7 devient alors tel que $u = at(u)$. Tous ses descendants forment une composante fortement connexe $\{7, 8, 9\}$. Après leur suppression, le sommet 2 satisfait $u = at(u)$ et il n'a plus de descendant satisfaisant la même relation. On trouve ainsi une nouvelle composante $\{2, 3, 4, 5\}$. Une fois celle-ci supprimée 1 est le seul sommet qui satisfait la relation $u = at(u)$ d'où la composante $\{1, 10, 11, 12, 13, 14, 15, 16, 17\}$. Dans ce cas particulier du sommet 1, on peut atteindre tous les sommets du graphe et le calcul s'arrête donc là; en général il faut reconstruire une arborescence de Trémaux à partir d'un sommet non encore atteint.

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, techniquement long à prouver mais guère difficile que la suppression des descendants de u lorsque $u = at(u)$ ne modifie pas les calculs des $at(v)$ en cours. La programmation donnée ici suppose que les sommets ont déjà été numérotés par l'algorithme de Trémaux à partir de x:

```
static int[] compCon (GrapheSucc g) {

  int num = 0;
  int numComp[] = new int[g.nb];
  int at[] = new int[g.nb];
  for (int x = 0; x < g.nb; ++x) {
      numComp[x] = 0;
      at[x] = Omega;
  }
  for (int x = 0; x < g.nb ; ++x)
      if (numComp[x] == 0 && at[x] == Omega)
          num = pointAttache1 (x, g, at, numComp, num);
  return numComp;
}

static int pointAttache1 (int x, GrapheSucc g,
                          int at[], int numComp[], int num) {
  int res = num;
  at[x] = x;
  for (int k = 0; g.succ[x][k] != Omega; k++) {
      int y = g.succ[x][k];
      if (at[y] == Omega) {
          res = pointAttache1 (y, g, at, numComp, res);
          if (at[x] > at[y])
              at[x]= at[y];
      } else
        if (numComp[y] == 0)
             at[x] = Math.min (at[x], y) ;
  }
  if (x == at[x]) {
```

```
        ++res;
        supprimerComp(x, res, numComp, g);
    }
    return res;
}

static void supprimerComp (int x, int num, int[] numComp,
                           GrapheSucc g) {
    numComp[x] = num;
    for (int k = 0; g.succ[x][k] != Omega; ++k) {
        int y = g.succ[x][k];
        if (y > x && numComp[y] == 0)
            supprimerComp (y, num, numComp, g);
    }
}
```

5.9 Programmes en Caml

```
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 = vect_length m
and nb_colonnes m =
  if vect_length m = 0 then failwith "nb_colonnes"
  else vect_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 =
     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 =
    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 =
    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) vect;;
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 =
    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 = make_vect (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 = make_vect (vect_length succ) (-1);;
let num = ref (-1);;

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

```
let numPrefixe() = begin
  do_vect
    (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;
    do_list
      (function  x -> if numéro.(x) = -1 then
         begin
         fajouter x f;
         numéro.(x) <-  0
         end)
      succ.(k)
    end
  done
  end;;

let numLargeur() = begin
  do_vect
    (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;
    pajouter k p;
    let min = ref !num in begin
      do_list
        (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 " (pvaleur(p));
            numéro.(pvaleur(p)) <- max_int;
            psupprimer(p);
```

```
            if pvaleur(p) = k then raise Exit
            done
        with Exit -> printf "\n");
      !min
      end
    end;;

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

Chapitre 6

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 fondamentale en programmation, elle a suscité le développement de très nombreuses techniques qui ont aussi donné lieu à des théories maintenant classiques. La compilation d'un programme est réalisée en 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, 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.

Nous nous limitons dans ce chapitre à l'étude de l'analyse syntaxique. L'étude de la génération de code, qui est la partie la plus importante de la compilation, nous conduirait à des développements trop longs. En revanche, le choix aurait pu se porter sur l'analyse lexicale, et nous aurait fait introduire la notion d'automate. Nous préférons illustrer la notion d'arbre, étudiée au chapitre 4, et montrer des exemples d'arbres représentant une formule symbolique. La structure d'arbre est fondamentale en informatique. Elle permet de représenter de façon structurée et très efficace des notions qui se présentent sous forme d'une chaîne de caractères. Ainsi, 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 de cette importante problématique. 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 simples comment il faudrait faire. L'ouvrage de base pour l'étude de la compilation est celui de A. Aho, R. Sethi, J. Ullman [3]. Les premiers chapitres de l'ouvrage [33] constituent une intéressante introduction à divers aspect de l'informatique théorique qui doivent leur développement à des problèmes rencontrés en compilation.

6.1 Définitions et notations

6.1.1 Mots

Un programme peut être considéré comme une très longue chaîne de caractères, dont chaque élément est un des symboles le composant. Un minimum de terminologie sur les chaînes de caractères ou *mots* est nécessaire pour décrire les algorithmes d'analyse syntaxique. Pour plus de précisions sur les propriétés algébriques et combinatoires des mots, on pourra se reporter à [34].

On utilise un ensemble fini appelé *alphabet* A dont les éléments sont appelés des *lettres*. Un *mot* est une suite finie $f = a_1a_2\ldots a_n$ de lettres, l'entier n s'appelle sa longueur. On note par ϵ le *mot vide*, c'est le seul mot de longueur 0. Le *produit* de deux mots f et g est obtenu en écrivant f puis g à la suite, celui ci est noté fg. On peut noter que la longueur de fg est égale à la somme des longueurs de f et de g. En général fg est différent de gf. 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^*.

Exemples

1. Mots sans carré

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

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

   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 $g = \epsilon$. On peut noter 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.

2. Expressions préfixées

   Les expressions préfixées, considérées au chapitre 3 peuvent être transformées en des mots sur l'alphabet $A = \{+, *, (,), a\}$, on remplace tous les nombres par la lettre a pour en simplifier l'écriture. En voici deux exemples,

   $$f = (*aa) \qquad g = (*(+a(*aa))(+(*aa)(*aa)))$$

3. Un exemple proche de la compilation

   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 $f =$ `while p do begin if q then x else y ; z end`
   est un mot de longueur 13, qui peut se décomposer en

   $$f = \texttt{while p do begin}\ g\ \texttt{; z end}$$

   où $g =$ `if q then x else y`.

6.1.2 Grammaires

Pour construire des ensembles de mots, on utilise la notion de *grammaire*. Une grammaire $\mathcal{G}$ comporte deux alphabets A et Ξ, un *axiome* S_0 qui est une lettre appartenant à Ξ 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 Ξ est dit alphabet *auxiliaire*, ses lettres servent de variables intermédiaires servant à engendrer des mots. Une lettre S_0 de Ξ, appelée axiome, joue un rôle particulier.

– Les règles sont toutes de la forme:

$$S \to u$$

où S est une lettre de Ξ et u un mot comportant des lettres dans $A \cup \Xi$.

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

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

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

> On part de l'axiome S_0 et on choisit une règle de la forme $S_0 \to u$. Si u ne contient aucune lettre auxiliaire, on a terminé. Sinon, on écrit $u = u_1Tu_2$. On choisit une règle de la forme $T \to v$. On remplace u par $u' = u_1vu_2$. 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 Ξ. 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 $A \cup \Xi$, 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 $v = w_1Sw_2$ et $u = w_1ww_2$. 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 $w_0, w_1, \ldots w_n$, $n \geq 0$ tels que $w_0 = v$, $w_n = u$ et pour tout $i = 1, \ldots n$, on a $w_{i-1} \to w_i$.

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 $\mathcal{L}(\mathcal{G})$.

Exemple Reprenons l'exemple de grammaire $\mathcal{G}$ donné plus haut 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 $abbaba$ que l'on obtient 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.

Notations Dans la suite, on adoptera des conventions strictes de notations, ceci facilitera la lecture du chapitre. Les éléments de A sont notés par des lettres minuscules du début de l'alphabet $a, b, c, \ldots$ éventuellement indexées si nécessaire $a_1, a_2, b_1, b_2 \ldots$, ou bien des symboles appartenant à des langages de programmation. Les éléments de Ξ sont choisis parmi les lettres majuscules S, T, U par exemple. Enfin les mots de A^* sont notés par f, g, h et ceux de $(A \cup \Xi)^*$ par u, v, w, indexés si besoin est.

6.2 Exemples de Grammaires

6.2.1 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 immédiatement sous la forme de la grammaire suivante:

$A = \{a, b\}$, $\Xi = \{S\}$, l'axiome est S et 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 $S \to \epsilon$.

6.2.2 Les expressions arithmétiques préfixées

Ces expressions ont été définies dans le chapitre 3 et la structure de pile a été utilisée pour leur évaluation. Là encore, la définition récursive se traduit immédiatement par une grammaire:

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

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

Les mots donnés en exemple plus haut sont engendrés de la façon suivante:

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

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 Ξ et prendre comme nouvelles règles:

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

On peut aussi augmenter la grammaire de façon à engendrer les nombres en notation décimale, la lettre a devrait alors être remplacée par un élément U de Ξ et des règles sont ajoutées pour que U engendre une suite de chiffres ne débutant pas par un 0.

$$\begin{array}{llll} U \to V_1 U_1 & U \to V & U_1 \to V U_1 & U_1 \to V \\ V_1 \to i \quad \text{pour} & 1 \le i \le 9 & & \\ V \to 0 & V \to V_1 & & \end{array}$$

6.2.3 Les expressions arithmétiques

C'est un des langages que l'on choisit souvent comme exemple en analyse syntaxique, car il contient la plupart des difficultés d'analyse que l'on rencontre dans les langages de programmation. Les mots engendrés par la grammaire suivante sont toutes les expressions arithmétiques que l'on peut écrire avec les opérateurs $+$ et $*$ on les appelle parfois expressions arithmétiques infixes. On les interprète en disant que $*$ est prioritaire vis à vis de $+$.

$A = \{+, *, (,), a\}$, $\Xi = \{E, T, F\}$, l'axiome est E, les règles de grammaire sont données par:

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

Un mot engendré par cette grammaire est par exemple:

$$(a + a * a) * (a * a + a * a)$$

Il représente l'expression

$$(5 + 2 * 3) * (10 * 10 + 9 * 9)$$

dans laquelle tous les nombres ont été remplacés par le symbole a.

Les lettres de l'alphabet auxiliaire ont été choisies pour rappeler la *signification sémantique* des mots qu'elles engendrent. Ainsi E, T et F représentent respectivement les expressions, termes et facteurs. Dans cette terminologie, on constate que toute expression est somme de termes et que tout terme 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.

$$E \rightarrow E + T \rightarrow E + T + T \ldots\ldots \xrightarrow{*} T + T + T \ldots + T$$

$$T \rightarrow T * F \rightarrow T * F * F \ldots\ldots \xrightarrow{*} F * F * F \ldots * F$$

La convention usuelle de priorité de l'opération $*$ 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

$$E \rightarrow E - T \qquad\qquad T \rightarrow T \ / \ F$$

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 Ξ. Il faudrait faire intervenir de nouvelles variables auxiliaires si l'on introduit de nouvelles priorités.

La grammaire donnée ci-dessous engendre aussi le langage des expressions infixes. On verra que cette dernière permet de faire plus facilement l'analyse syntaxique. Elle n'est pas utilisée en général en raison de questions liées à la non-associativité de certains opérateurs comme par exemple la soustraction et la division. Ceci pose des problèmes lorsqu'on désire généraliser la grammaire et utiliser le résultat de l'analyse syntaxique pour effectuer la génération d'instructions machine.

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

6.2.4 Grammaires sous forme BNF

La grammaire d'un langage de programmation est très souvent présentée sous la forme dite grammaire BNF qui n'est autre qu'une version très légèrement différente de notre précédente notation.

Dans la convention d'écriture adoptée pour la forme BNF, les éléments de Ξ 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, ainsi *one of* permet d'écrire plusieurs règles sur la même ligne. Enfin, 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 de cet exemple trop long. 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
    FieldAccess
    MethodInvocation
    ArrayAccess

Ceci correspond approximativement dans notre notation à

$E \to M$	$E \to E$ + M	$E \to E$ - M	
$M \to U$	$M \to P$ * U	$M \to M$ / U	$M \to M$ % U
$U \to I$	$U \to D$		
$U \to$ +U	$U \to$ -U	$U \to U'$	
$U' \to P'$	$U' \to$ ~U	$U' \to$!U	$U' \to C$
$P \to P'$	$P \to N$	$P \to I'$	$P \to D'$
$P' \to P''$	$P' \to A$		
$P'' \to L$	$P'' \to$ `this`	$P'' \to (E)$	...

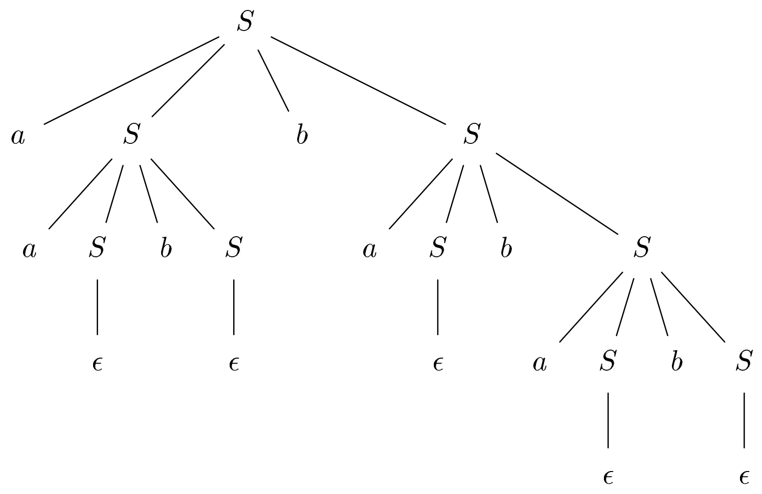


FIG. 6.1 – *Arbre de dérivation de aabbabab*

6.3 Arbres de dérivation et arbres de syntaxe abstraite

Le but de l'analyse syntaxique est d'abord 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 bien plus clairement sous forme d'un arbre, dit *arbre de dérivation* dont des exemples sont donnés figures 6.1 et 6.2. Il s'agit de ceux obtenus pour les mots *aabbabab* et $(a + a * a) * (a * a + a * a)$ engendrés respectivement par la grammaire des systèmes de parenthèses et par celle des expressions infixes. On verra que ces arbres, ou plutôt une version plus compact de ceux-ci, jouent un rôle important pour la phase suivante de la compilation. Une définition rigoureuse et complète des arbres de dérivation serait longue, contentons nous de quelques indications informelles. Dans un tel arbre, les noeuds internes sont étiquetés par des lettres auxiliaires (appartenant à Ξ) les feuilles par des lettres de l'alphabet terminal. L'étiquette de la racine est égale à l'axiome. Pour un noeud interne d' étiquette S, le mot u obtenu en lisant de gauche à droite les étiquettes de ses fils est tel que $S \to u$ est une règle. Enfin, le mot f dont on fait l'analyse est constitué des étiquettes des feuilles lues de gauche à droite.

Pour un mot donné du langage engendré par une grammaire, l'arbre de dérivation n'est pas nécessairement unique. L'existence de plusieurs arbres de dérivations pour un même programme signifie en général qu'il existe plusieurs interprétations possibles pour celui ci. On dit que la grammaire est ambiguë, c'est le cas pour l'imbrication des `if then` et `if then else` en Pascal. Des indications supplémentaires dans le manuel de référence du langage permettent alors de lever l'ambiguïté et d'associer un arbre unique à tout programme Pascal. Ceci permet de donner alors une interprétation unique. Toutes les grammaires données plus haut sont non-ambiguës, ceci peut se démontrer rigoureusement, toutefois les preuves sont souvent techniques et ne présentent pas beaucoup d'intérêt.

L'arbre de dérivation est parfois appelé arbre de syntaxe concrète pour le distinguer de *l'arbre de syntaxe abstraite* construit généralement par le compilateur d'un langage de programmation. Cet arbre de syntaxe abstraite est plus compact que le précédent et contient des informations sur la suite des actions effectuées par un programme. Chaque

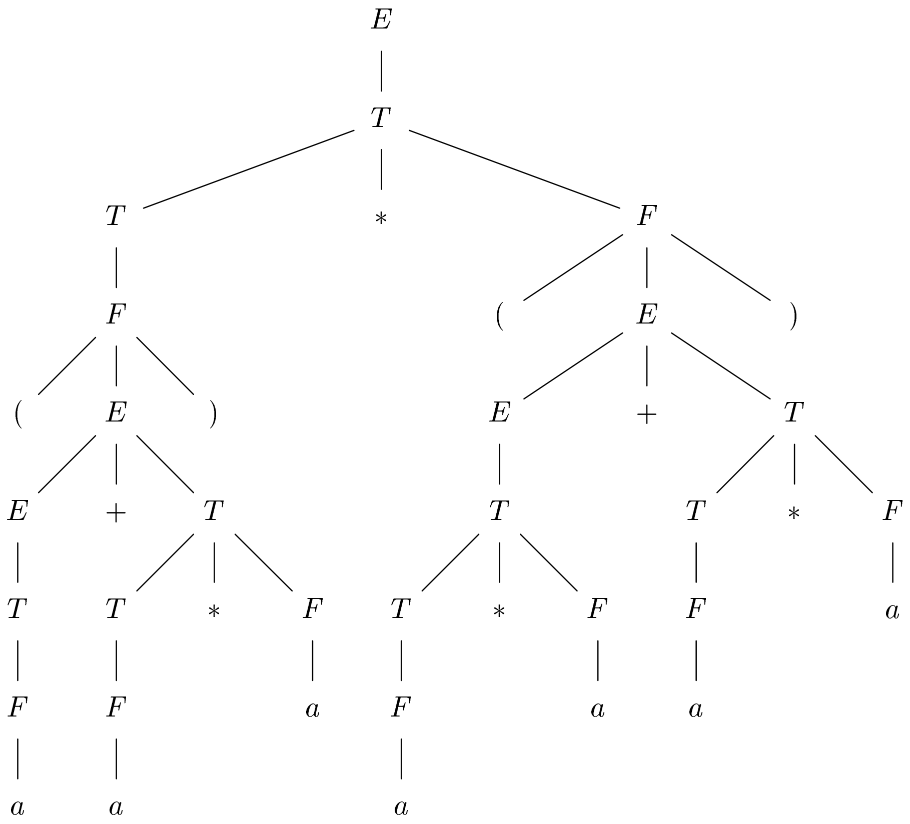


FIG. 6.2 – *Arbre de dérivation d'une expression arithmétique*

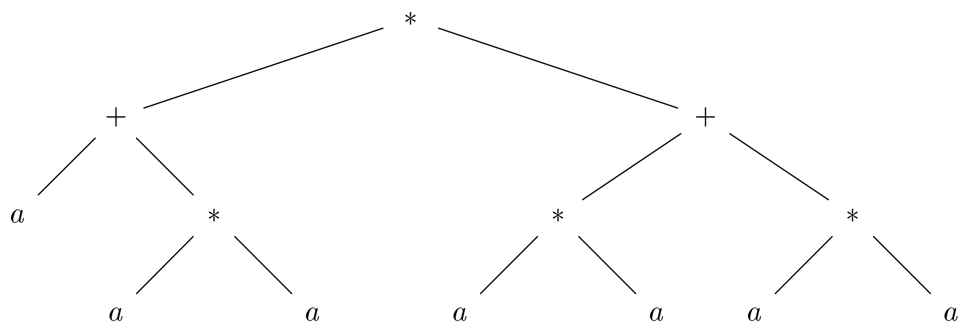


FIG. 6.3 – *Arbre de syntaxe abstraite de l'expression*

noeud interne de cet arbre possède une étiquette qui désigne une opération à exécuter. Il s'obtient par des transformations simples à partir de l'arbre de dérivation. On donne en exemple figure 6.3 l'arbre de syntaxe abstraite correspondant à l'arbre de dérivation de la figure 6.2.

6.4 Analyse descendante récursive

Deux principales techniques sont utilisées pour effectuer l'analyse syntaxique. Il faut en effet, étant donnés une grammaire G et un mot f, de construire la suite des dérivations de G ayant conduit de l'axiome au mot f,

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

La première technique consiste à démarrer de l'axiome 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_0. 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 cas très particuliers. Dans ces cas, l'algorithme d'analyse syntaxique devient une traduction fidèle de l'écriture de la grammaire. On utilise pour cela autant de procédures qu'il y a d'éléments dans Ξ chacune d'entre elles étant destinée à reconnaître un mot dérivant de l'élément correspondant de Ξ. Examinons comment cela se passe sur l'exemple de la grammaire des expressions infixes, nous choisissons ici la deuxième forme de cette grammaire:

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

Que l'on traduit par les trois procédures récursives croisées suivantes en Java. Celles ci construisent l'arbre de syntaxe abstraite comme dans le chapitre 4.

```
class ASA {

  char op;
  ASA filsG, filsD;

  ASA (char x, ASA a, ASA b) {
    op = x;
```

```
      filsG = a;
      filsD = b;
    }

    static String f;
    static int i = 0;

    static ASA expression() throws ErreurDeSyntaxe {

      ASA a = terme();
      if (f.charAt(i) == '+') {
          ++i;
          return new ASA('+', a, expression());
      }
      else return a;
    }

    static ASA terme() throws ErreurDeSyntaxe {

      ASA a = facteur();
      if (f.charAt(i) == '*') {
          ++i;
          return new ASA ('*', a, terme());
      }
      else return a;
    }

    static ASA facteur() throws ErreurDeSyntaxe {

      if (f.charAt(i) == '(') {
          ++i;
          ASA a = expression();
          if (f.charAt(i) == ')') {
              ++i;
              return a;
              }
          else throw new ErreurDeSyntaxe(i);
      }
      else if (f.charAt(i) == 'a') {
              ++i;
              return new ASA ('a', null, null);
           }
           else throw new ErreurDeSyntaxe(i);
    }
  }
```

Dans ce programme, le mot f à analyser est une variable globale. Il en est de même pour la variable entière `i` qui désigne la position à partir de laquelle on effectue l'analyse courante. Lorsqu'on active la procédure `expression`, on recherche une expression commençant en $f[i]$. A la fin de l'exécution de cette procédure, si aucune erreur n'est détectée, la nouvelle valeur (appelons la i_1) de i est telle que $f[i]f[i+1]...f[i_1-1]$ est une expression. Il en est de même pour les procédures `terme` et `facteur`. Chacune de ces procédures tente de retrouver à l'intérieur du mot à analyser une partie engendrée par E, T ou F. Ainsi, la procédure `expression` commence par rechercher un `terme`. Un nouvel appel à `expression` est effectué si ce terme est suivi par le symbole `+`. Son action se termine sinon. La procédure `terme` est construite sur le même modèle et la procédure `facteur` recherche un symbole `a` ou une expression entourée de parenthèses. Si une erreur de syntaxe est détectée, on envoit une exception avec la position courante dans le mot à reconnaitre, ce qui permettra de préciser l'endroit où l'erreur a été rencontrée. La classe de l'exception est definie par:

```
class ErreurDeSyntaxe extends Exception {
  int position;

  public ErreurDeSyntaxe (int i) {
    position = i;
  }
}
```

Cette technique fonctionne bien ici car les membres droits des règles de grammaire ont une forme particulière. Pour chaque élément S de Ξ, l'ensemble des membres droits $\{u_1, u_2 \ldots u_p\}$ de règles, dont le membre gauche est S, 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 Ξ sont tous facteurs gauches les uns des autres. Beaucoup de grammaires de langages de programmation satisfont ces conditions: Pascal, Java.

Une technique plus générale d'analyse 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 $u = S_0$. 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 $y \in A$, alors f ne peut dériver de u, et il faut faire repartir l'analyse du mot qui a donné u. Sinon $y \in \Xi$ et on recherche toutes les règles dont y est le membre gauche.

$$y \to u_1, y \to u_2, \ldots y \to u_k$$

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 8. On peut la représenter par la procédure suivante donnée sous forme informelle.

```
static boolean Analyse (String u, 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;
  }
}
```

On écrit donc un programme comme suit en utilisant la fonction `estAuxiliaire(y)` qui vaut vrai ssi $y \in \Xi$. On suppose que les mots f et u se terminent respectivement par `$` et `#`, il s'agit là de sentinelles permettant de détecter la fin de ces mots. On suppose aussi que l'ensemble des règles est contenu dans un tableau `regle[S][i]` qui donne la $i+1$-ème règle dont le membre gauche est S. Le nombre de règles dont le membre gauche est S est fourni par `nbRegles[S]`.

```
static int pos = 1;

static boolean analyseDescendante (String u, String f) {

  while (f.charAt(pos) == u.charAt(pos))
      ++pos;
  if (f.charAt(pos) == '$' && u.charAt(pos) == '#')
      return true;
  else {
      char y = u[pos];
      if (estAuxiliaire(y))
          for (int i = 0; i < nbRegles[y]; ++i) {
              String v = u.substring(0, pos) + regle[y][i]
                       + u.substring(pos+1);
              if (Analyse (v, f))
                  return true;
          }
      return false;
  }
}
```

Remarques

1. Cette procédure ne donne pas de résultat lorsque la grammaire est ce qu'on appelle récursive à gauche (le mot récursif n'a pas ici tout à fait le même sens que dans les procédures récursives), c'est à dire lorsqu'il existe une suite de dérivations partant d'un S de Ξ et conduisant à un mot u qui commence par S. Tel est le cas pour la première forme de la grammaire des expressions arithmétiques infixes qui ne peut donc être analysée par l'algorithme ci dessus.

2. Les transformations que l'on applique au mot u s'expriment bien à l'aide d'une pile dans laquelle on place le mot à analyser, sa première lettre en sommet de pile.

3. Cette procédure est très coûteuse en temps lors de l'analyse d'un mot assez long 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'est pas la bonne. Il faut alors tout recommencer avec une autre règle et éventuellement répéter plusieurs fois. La complexité de l'algorithme est ainsi une fonction exponentielle de la longueur du mot à analyser.

4. 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`. Noter que dans ces conditions la procédure d'analyse donne un résultat même dans le cas de grammaires récursives à gauche.

6.5 Analyse LL

Une technique pour éviter les calculs longs de l'analyse descendante récursive consiste à tenter de deviner la première règle qui a été appliquée 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

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

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 certaines comme par exemple celle des expressions préfixées ou une grammaire modifiée des expressions infixes. On dit alors que la grammaire satisfait la condition LL.

Expressions préfixées Nous considérons la grammaire de ces expressions:

$A = \{+, *, (,), a\}$, $\Xi = \{S\}$, 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 $S \to a$, 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 $S \to (+SS)$ qui a été appliquée, si ces deux lettres sont $(*$ alors c'est la règle $S \to (*SS)$. 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.

Cet algorithme reprend les grandes lignes de la descente récursive avec une différence importante: la boucle `while` qui consistait à appliquer chacune des règles de la grammaire est remplacée par un examen de certaines lettres du mot à analyser, examen qui permet de conclure sans retour arrière. On passe ainsi d'une complexité exponentielle à un algorithme en $O(n)$. En effet, une manière efficace de procéder consiste à utiliser une pile pour gérer le mot v_i qui vient de la décomposition $u_i = g_i S v_i$. La consultation de la valeur en tête de la pile et sa comparaison avec la lettre courante de f permet de décider de la règle à appliquer. L'application d'une règle consiste alors à supprimer la tête de la pile (membre gauche de la règle) et à y ajouter le mot formant le membre droit en commençant par la dernière lettre.

Nous avons appliqué cette technique pour construire l'arbre de syntaxe abstraite associé à une expression préfixée. Dans ce qui suit, le mot à analyser f est une variable globale de même que la variable entière `pos` qui indique la position à laquelle on se trouve dans ce mot.

```
static Arbre ArbSyntPref() {
  Arbre a;
  char x;
```

```
    if (f.charAt(pos) == 'a') {
        a = NouvelArbre('a', null, null);
        ++pos;
    } else if (f.charAt(pos) == '(' &&
               (f.charAt(pos+1) == '+' || f.charAt(pos+1) == '*')) {
        x = f.charAt(pos + 1);
        pos = pos + 2;
        a = NouvelArbre(x, ArbSyntPref(), ArbSyntPref());
        if (f.charAt(pos) == ')')
             ++pos;
        else
             erreur(pos);
    } else
        erreur(pos);
    return a;
}
```

L'algorithme d'analyse syntaxique donné ici peut s'étendre à toute grammaire dans laquelle pour chaque couple de règles $S \to u$ et $S \to v$, les mots qui dérivent de u et v n'ont pas des facteurs gauches égaux de longueur arbitraire. Ou de manière plus précise, il existe un entier k tel que tout facteur gauche de longueur k appartenant à A^* d'un mot qui dérive de u est différent de celui de tout mot qui dérive de v. On dit alors que la grammaire est $LL(k)$ et on peut alors démontrer:

Théorème 5 *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 est surtout utile pour $k = 1$. Nous donnons ici ses grandes lignes sous forme d'un programme Pascal qui utilise une fonction `Predicteur`(S, g) calculée au préalable. Pour un élément S de Ξ et un mot g de longueur k, cette fonction indique le numéro de l'unique règle $S \to u$ telle que u se dérive en un mot commençant par g ou qui indique `Omega` si aucune telle règle n'existe. Dans l'algorithme qui suit, on utilise une pile comme variable globale. Elle contient la partie du mot u qui doit engendrer ce qui reste à lire dans f. Nous en donnons ici une forme abrégée.

```
static boolean Analyse(String f, int pos) {
    int i;

    pos = 1;
    while (Pile.valeur(p) == f.charAt(pos)) {
        Pile.supprimer(p);
        ++pos;
    }
    if (Pile.vide(p) && f.charAt(pos) == '$')
        return true;
    else {
        y = Pile.valeur(p);
        if (! estAuxiliaire(y))
           return false;
        else {
           i = Predicteur (y, pos, pos+k-1);
           if (i != Omega) {
                System.out.println (y, '->', regle[y][i]);
                Pile.inserer(regle[y,i], p);
```

```
                    return Analyse (f, pos);
                } else
                    return false;
            }
        }
    }
```

6.6 Analyse ascendante

Les algorithmes d'analyse ascendante sont souvent plus compliqués que ceux de l'analyse descendante. Ils s'appliquent toutefois à un beaucoup plus grand nombre de grammaires. C'est pour cette raison qu'ils sont très souvent utilisés. Ils sont ainsi à la base du système `yacc` 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 bien 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:

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

Cet algorithme tient compte pour effectuer une réduction 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. 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. Noter que l'on a introduit l'opérateur de soustraction qui n'est pas associatif. Ainsi la technique d'analyse décrite au début du paragraphe 6.4 ne peut être appliquée ici.

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. On peut construire aussi un arbre de syntaxe abstraite en utilisant une autre pile qui contient cette fois des arbres (c'est à dire des pointeurs sur des noeuds). Les deux piles sont traitées en parallèle, la réduction par une règle a pour effet sur la deuxième pile de construire un nouvel arbre dont les fils se trouvent en tête de la pile, puis à remettre le résultat dans celle-ci.

6.7 Evaluation

Dans la plupart des algorithmes que nous avons donnés, il a été question d'arbre de syntaxe abstraite d'une expression arithmétique. Afin d'illustrer l'intérêt de cet arbre, on peut examiner la simplicité de la fonction d'évaluation qui permet de calculer la valeur de l'expression analysée à partir de l'arbre de syntaxe abstraite.

```
static int evaluer(Arbre x) {

    if (x.valeur == 'a')
        return x.valeur;
    else if (x.valeur == '+')
        return evaluer(x.filsG) + evaluer(x.filsD);
    else if (x.valeur == '-')
```

```
            return evaluer(x.filsG) - evaluer(x.filsD);
        else if (x.valeur == '*')
            return evaluer(x.filsG) * evaluer (x.filsD);
}
```

Une fonction similaire, qui ne demanderait pas beaucoup de mal à écrire, permet de créer une suite d'instructions en langage machine traduisant le calcul de l'expression. Il faudrait remplacer les opérations +, *, -, effectuées lors de la visite d'un noeud de l'arbre, par la concaténation des listes d'instructions qui calculent le sous-arbre droit et le sous arbre gauche de ce noeud et de faire suivre cette liste par une instruction qui opère sur les deux résultats partiels. Le programme complet qui en résulte dépasse toutefois le but que nous nous fixons dans ce chapitre.

6.8 Programmes en Caml

```
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 = make_string (string_length s) ‘ ‘ in
  blit_string 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 152 *)

```
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 155 *)
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 7

Modularité

Jusqu'à présent, nous n'avons vu que l'écriture de petits programmes ou de procédures suffisant pour apprendre les structures de données et les algorithmes correspondants. La partie la plus importante de l'écriture des vrais programmes consiste à les structurer pour les présenter comme un assemblage de briques qui s'emboitent 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 ou de fonctions, et il devient quasiment impossible de réaliser un programme correct.

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

7.1 Un exemple: les files de caractères

Pour illustrer notre chapitre, nous utilisons un exemple réel tiré du noyau du système Unix. Les files ont été décrites dans le chapitre 3 sur les structures de données élémentaires. Nous avons vu deux manières de les implémenter: par un tableau circulaire ou par une liste. Les files de caractères sont très couramment utilisées, par exemple pour gérer les entrées/sorties d'un terminal (*tty driver*) ou du réseau Ethernet.

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! C'est beaucoup. La représentation par tableau circulaire semble donc 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 circulaire.

Introduisons une troisième réalisation possible de ces files. Au lieu de représenter la file par une liste de tous les caractères la constituant, nous allons regrouper les caractères par blocs contigus de t caractères. Les premier et dernier éléments de la liste pourront être incomplets (comme indiqué dans la figure 7.1). 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).

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

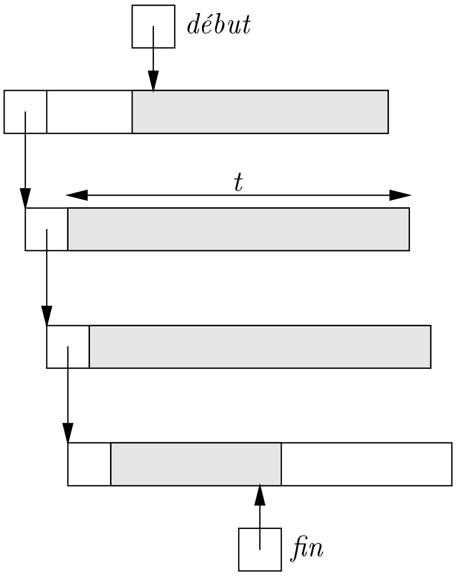


FIG. 7.1 – *File de caractères*

relative dans ce bloc comme indiqué sur la figure 7.2. La déclaration de la classe `FC` d'une file de caractères s'effectue comme suit:

```
class FC {

  int cc;
  Bloc debut_b, fin_b;
  int debut_d, fin_d;

  FC () {
    cc = 0;
  }

  static class Bloc {

    final static int TAILLE = 12;
    Bloc suivant;
    char [] contenu;

    Bloc () {
        suivant = null;
        contenu = new char[TAILLE];
    }
  }
}
```

La file vide est représentée par un compte de caractères nul.

```
static FC vide () {
  return new FC();
}
```

L'ajout et la suppression d'un caractère dans une file s'effectuent comme au chapitre 3. Pour respecter la structure des blocs, il faut tester si le caractère suivant est dans le même bloc ou s'il est nécessaire d'aller chercher le bloc suivant dans la liste des

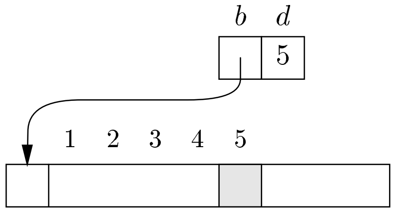


Fig. 7.2 – *Adresse d'un caractère par base et déplacement*

blocs. Lors de l'ajout, il faut allouer un nouveau bloc dans le cas de la file vide ou du franchissement d'un bloc.

```
static FC 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;
  return x;
}
```

La suppression s'effectue au début de file. Pour la suppression, il faut au contraire rendre un bloc si le caractère supprimé (rendu en résultat) libère un bloc. Par convention, nous retournons le caractère nul quand on demande de supprimer un caractère dans une file vide. Une meilleure solution aurait été de retourner une exception.

```
static char supprimer (FC x) {

  char res;
  if (x.cc == 0)
      return -1;
  else {
      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;
  }
}
```

7.2 Interfaces et modules

Reprenons l'exemple précédent. Supposons qu'un programme, comme un gestionnaire de terminaux, utilise des files de caractères, une pour chaque terminal. On ne doit pas mélanger la gestion des files de caractères avec le reste de la logique du programme. Il faut donc regrouper les procédures traitant des files de caractères. Le programme utilisant les files de caractères n'a pas besoin de connaître tous les détails de l'implémentation de ces files. Il ne doit connaître que la déclaration des types utiles dans la classe `FC`: le nom de la classe et les trois procédures pour initialiser une file vide, ajouter un élément au bout de la file et retirer le premier élément. Précisément, on peut se contenter de l'*interface* suivante:

```
class FC {

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

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

  static int supprimer (FC x) {...}
  /* Supprime le premier caractère c de x et rend c comme résultat */
  /* Si x est vide, le résultat est -1 */
```

On ne manipulera les files de caractères 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. Le corps des procédures sur les files seront dans la partie *implémentation* du *module* des files de caractères. Dans l'interface d'un module, on a donc des types, des procédures ou des fonctions que l'on veut exporter ou rendre publiques, et il est bon d'y commenter la fonctionnalité de chaque élément pour comprendre sa signification, puisqu'un utilisateur n'aura besoin que de consulter 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? 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` signifie que seuls les instructions à l'intérieur de la classe où il est définie pourront accéder à ce champ ou méthode. Au contraire `public` dit qu'un champ est accessible par toutes les classes. Jusqu'à présent, nous n'avons pas mis (sauf pour la procédure `main`) de qualificatifs. L'option par défaut est de rendre public les champs (en fait à l'intérieur du *package* où il est défini, mais nous verrons plus loin la notion de *package*). Redéfinissons donc notre classe `FC`.

```
class FC {

  private int cc;
  private Bloc debut_b, fin_b;
  private int debut_d, fin_d;

  public FC () {
    cc = 0;
  }

  private static class Bloc {

    final static int TAILLE = 12;
    Bloc suivant;
```

```
    char [] contenu;

    Bloc () {
        suivant = null;
        contenu = new char[TAILLE];
    }
  }

  public static FC vide () {
    return new FC();
  }

  public static FC 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;
    return x;
  }

  public static char supprimer (FC x) {

    char res;
    if (x.cc == 0)
        return -1;
    else {
        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;
    }
  }
}
```

On peut avoir à cacher non seulement le code des procédures réalisant l'interface, mais aussi des variables et des fonctions qui ne seront pas accessibles de l'extérieur. Supposons dans notre exemple, que, pour être hyper efficace (ce qui peut être le cas dans un *driver* de périphérique), nous voulions avoir notre propre stratégie d'allocation pour les blocs. On construira une liste des blocs libres `listeLibre` à l'initialisation du chargement de la classe `FC` et on utilisera les procédures `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;
  }

 private static class Bloc {

    final static int TAILLE = 12;
    Bloc suivant;
    char [] contenu;

    Bloc () {
        suivant = null;
        contenu = new char[TAILLE];
    }
  }

  public static FC vide () {
    return new FC();
  }

  public static FC 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 = nouveauBloc();
        x.fin_b.suivant = b;
        x.fin_b = b; x.fin_d = -1;
    }
    x.fin_b.contenu[++x.fin_d] = c;
    ++x.cc;
    return x;
  }

  public static int supprimer (FC x) {

    if (x.cc == 0)
return -1;
    else {
char res = x.debut_b.contenu[x.debut_d];
--x.cc;
++x.debut_d;
if (x.cc <= 0)
    libererBloc (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;
    libererBloc (b);
}
return res;
    }
```

```
    }
    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 nouveauBloc () {
      Bloc b = listeLibre;
      listeLibre = listeLibre.suivant;
      b.suivant = null;
      return b;
    }
    private static void libererBloc (Bloc b) {
      b.suivant = listeLibre;
      listeLibre = b;
    }
  }
```

On veut que la variable `listeLibre` reste cachée, puisque cette variable n'a aucun sens dans l'interface des files de caractères. Il en est de même pour les procédures d'allocation ou 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 dans la déclaration de la classe (qui ne sont exécutées qu'une seule fois, au chargement de la classe si le mot-clé `static` figure devant). Deuxièmement, pour ne pas compliquer le programme, nous ne testons pas le cas où la liste des blocs libres devient vide et, donc, l'allocation d'un nouveau bloc libre impossible. Troisièmement, on un créerait un nouveau module séparé pour l'allocation des blocs, si les procédures d'allocation et de libération de blocs étaient très complexes. Alors le module des files de caractères serait lui aussi constitué par plusieurs modules. Essayer de le faire en exercice.

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 pour éviter une programmation trop alambiquée. 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 d'écrire un programme 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.

7.3 Interfaces et modules en Java

Beaucoup de langages de programmation ont une notion explicite de modules et d'interfaces, par exemple Clu, Mesa, Ada, Modula-2, Oberon, Modula-3, et ML. En C ou C++, 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 coincider interfaces et implémentations. Seul le chargeur dynamique `ClassLoader` ou l'interpréteur (la machine virtuelle 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 réservés pour la construction de classes paramétrées ou pour la gestion dynamique simultanée de plusieurs implémentations pour un même interface.

Mais il existe une autre notion pour assurer la 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 avoir à qualifier les noms par le nom de la classe (ou d'un objet pour les méthodes dynamiques). Ainsi on écrit `Liste.ajouter`, `FC.TAILLE`, `args.length()`, `System.out.print` dans les exemples précédents. Parmi ces préfixes, `System` est un nom de *package*. Un *package* permet de regrouper un ensemble de classes qui ont des fonctionalités proches, par exemple les appels système, les entrées/sorties, etc. Un package correspond à un répertoire du système de fichiers, dont les classes sont des fichiers. Par défaut, une classe fait toujours partie d'un *package*, par défaut le répertoire dont le fichier contenant la classe fait partie. On peut spécifier le package où se trouve une classe en la précédent par l'instruction

```
package Nom_de_package;
```

Pour accéder à une classe, on doit la précéder par le nom du *package* dont elle fait partie. Mais, on peut éviter de mettre le nom complet en donnant la liste des noms de *packages* importés.

```
import java.io.*;
import java.lang.*;
import java.util.*;
```

7.4 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. Il s'agit donc d'un code interprété (*byte-code*) 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:

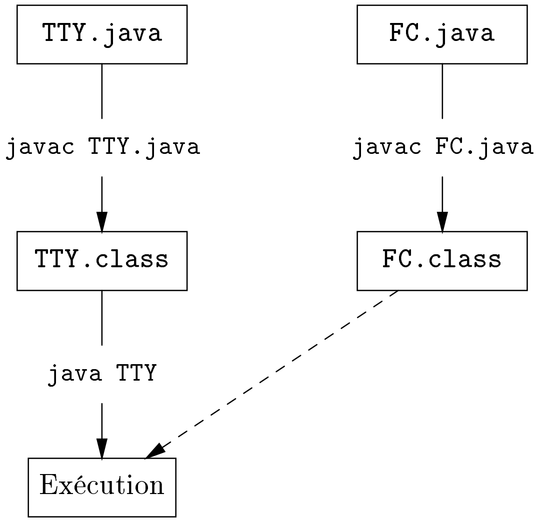


FIG. 7.3 – *Compilation séparée*

```
% javac FC.java
```

permet d'obtenir le fichier *byte-code* `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) { ··· }
}
```

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` n'est pas nécessaire 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 7.3.

Quand il y a un grand nombre de fichiers de code objet, on peut les regrouper dans un fichier d'archive `.jar` (*java archive*) pour en faire une *librairie*, par exemple `libX11.a`

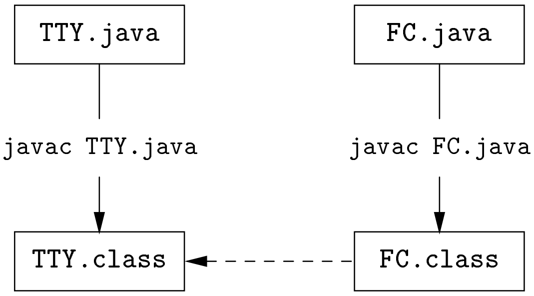


FIG. 7.4 – *Dépendances dans une compilation séparée*

pour X-window. Les diverses classes ou librairies sont retrouvées à des endroits standards, que l'on peut changer en se servant de la variable d'environnement `CLASSPATH`.

Avec CodeWarrior, la notion de *projet* permet de combiner un certain nombre de fichiers Java comme `FC.java` et `TTY.java`, ainsi qu'un certain nombre de librairies. La commande `Run` exécute des commandes de compilation séparée et d'édition de lien analogues à celles d'Unix.

7.5 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 pourra recompiler le strict nécessaire en cas de modification d'un des modules. Dans l'exemple de notre gestionnaire de terminaux, nous voulons indiquer que les dépendances induites par la figure 7.5 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

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, cette opération d'ordonnancement d'un graphe sans cycle s'appelle le *tri topologique*.

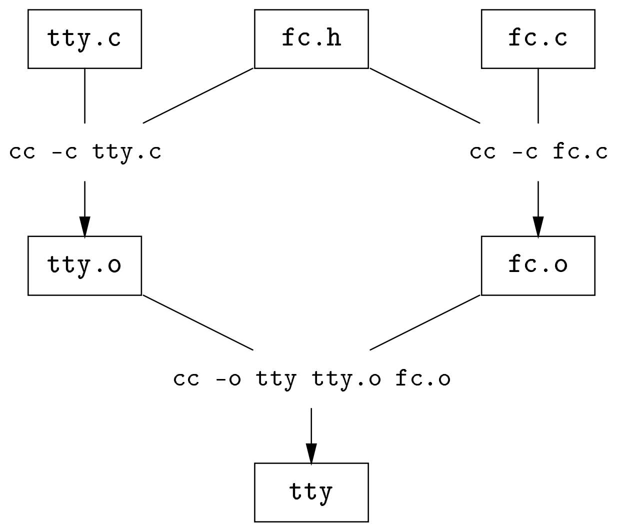


FIG. 7.5 – *Dépendances dans un* Makefile

7.6 Tri topologique

Au début de certains livres, les auteurs indiquent les dépendances chronologiques entre les chapitres en les représentant par un diagramme. Celui qui figure au début du livre de Barendregt sur lambda-calcul [4] est sans doute l'un des plus compliqués. Par exemple, on voit sur ce diagramme que 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 noeuds d'un graphe dirigé sans circuit (souvent appelés sous leur dénomination anglaise *dags* pour *directed acyclic graphs*) est appelée le tri topologique. Comme nous l'avons vu plus haut, elle est aussi bien utile dans la compilation et l'édition de liens des modules

Le tri topologique consiste donc à ordonner les sommets d'un dag en une suite dans laquelle l'origine de chaque arc apparait avant son extrémité. La construction faite ici est une version particulière du tri topologique, il s'agit pour un sommet s donné de construire 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. Pour résoudre ce problème, on applique l'algorithme de descente en profondeur d'abord (Trémaux) sur le graphe opposé. (Au lieu de considérer les successeurs `succ[u,k]` du sommet u, on parcourt ses prédécesseurs.) 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 noeud déjà visité que celui-ci figure dans la liste résultat. Pour ce faire, il faut utiliser un tableau annexe `etat` sur les noeuds qui indique si le noeud est visité, en cours de visite, ou non visité.

```
final static pasVu = 0, enCours = 1, dejaVu = 2;

static Liste triTopologique (int u) {
    for (int i = 0; i < nbSommets; ++i)
        etat[i] = pasVu;
    Liste resultat = Liste.reverse (DFS (u, null));
```

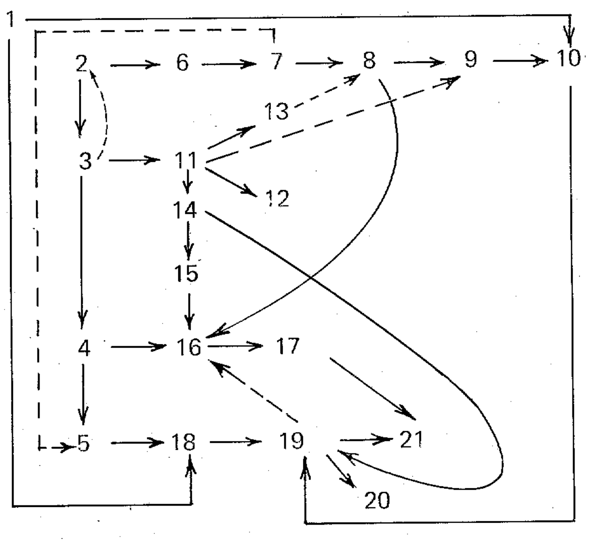


FIG. 7.6 – *Un exemple de graphe acyclique*

```
}
static Liste DFS (int u, Liste a_faire) {
  for (int k = 0; pred[u][k] != Omega; ++k) {
      int v = pred[u][k];
      if (etat[v] == enCours)
          Erreur ("Le graphe a un cycle");
      if (etat[v] == pasVu) {
          etat[v] = enCours;
          a_faire = DFS (v, a_faire);
      }
  }
  etat[u] = dejaVu;
  return Liste.ajouter (u, a_faire);
}
```

Nous avons omis les déclarations des variables `i` et `etat` et du type énuméré des éléments de ce tableau. Nous avons repris les structures développées dans les chapitres sur les graphes et les fonctions sur les listes. Nous supposons aussi que le tableau `succ` est remplacé par `pred` des prédécesseurs de chaque noeud.

7.7 Un exemple de module en C

C est proche de Java par sa non-existence d'un système de modules. Nous reprenons l'exemple des files de caractères (en fait telles qu'elles se trouvaient en Unix version 7 pour les gestionaires 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 de Pascal. 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. Contrairement à Pascal, 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;
    }
}
```

7.8 Modules en Caml

On construit pour tout module deux fichiers `fc.mli` (pour *ML interface*) et `fc.ml` (le module d'implémentation). 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 fc;;
        (* Le type des files de caractères *)

exception FileVide;;
        (* Levée quand [supprimer] est appliquée à une file vide. *)

value vide: unit -> fc
        (* Retourne une nouvelle file, initialement vide. *)
  and ajouter: char -> fc -> fc
        (* [ajouter c x] ajoute le caractère [c] à la fin de la file [x]. *)
  and supprimer: fc -> 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 fc = {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 = make_string 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 symbole `__` [1]

```
let nouveau_tty =
  let x = fc__vide() in
  ...
  let y = fc__ajouter c x ....
```

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 `.zo`, les fichiers `.mli` en `.zi`, et que l'on finit avec un fichier `tty` directement exécutable.

```
tty: tty.zo fc.zo
        camlc -o tty tty.zo fc.zo

tty.zo: tty.ml fc.zi
        camlc -c tty.ml

fc.zo: fc.ml
        camlc -c fc.ml

fc.zi: fc.mli
        camlc -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 `.zi` 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.

1. En OCaml, on est revenu à une notation plus classique avec un point au lieu des deux caractères souligné!

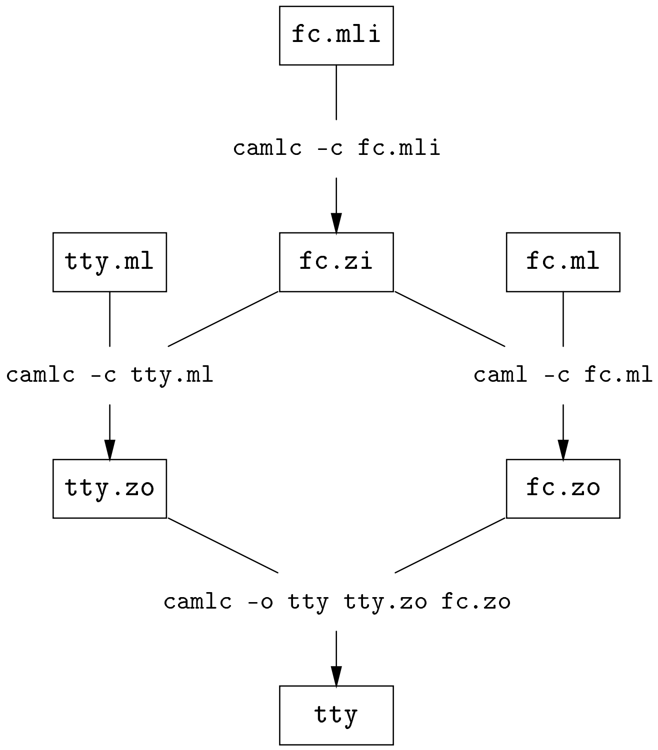


FIG. 7.7 – *Dépendances entre modules Caml*

Chapitre 8

Exploration

Dans ce chapitre, on recherche des algorithmes pour résoudre des problèmes se présentant sous la forme suivante:

> On se donne un ensemble E fini et à chaque élément e de E est affectée une valeur $v(e)$ (en général, un entier positif), on se donne de plus un prédicat (une fonction à valeurs *{vrai, faux}*) 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
> – $\sum_{e \in F} v(e)$ soit maximal (ou minimal, dans certains cas)

Les méthodes développées pour résoudre ces problèmes sont de natures très diverses. Pour certains exemples, il existe un algorithme très simple consistant à initialiser F par $F = \emptyset$, puis à ajouter successivement des éléments suivant un certain critère, jusqu'à obtenir la solution optimale, c'est ce qu'on appelle *l'algorithme glouton*. Tous les problèmes ne sont pas résolubles par l'algorithme glouton mais, dans le cas où il s'applique, il est très efficace. Pour d'autres problèmes, c'est un algorithme dit de *programmation dynamique* qui permet d'obtenir la solution, il s'agit alors d'utiliser certaines particularités de la solution qui permettent de diviser le problème en deux; puis de résoudre séparément chacun des deux sous-problèmes, tout en conservant en table certaines informations intermédiaires. Cette technique, bien que moins efficace que l'algorithme glouton, donne quand même un résultat intéressant car l'algorithme mis en oeuvre est en général polynomial. Enfin, dans certains cas, aucune des deux méthodes précédentes ne donne de résultat et il faut alors utiliser des procédures d'exploration systématique de l'ensemble de toutes les parties de E satisfaisant C, cette exploration systématique est souvent appelée *exploration arborescente* (ou *backtracking* en anglais).

8.1 Algorithme glouton

Comme il a été dit, cet algorithme donne très rapidement un résultat. En revanche ce résultat n'est pas toujours la solution optimale. L'affectation d'une ou plusieurs ressource à des utilisateurs (clients, processeurs, etc.) constitue une classe importante de problèmes. Il s'agit de satisfaire au mieux certaines demandes d'accès à une ou plusieurs ressources, pendant une durée donnée, ou pendant une période de temps définie précisément. Le cas le plus simple de ces problèmes est celui d'une seule ressource, pour laquelle sont faites des demandes d'accès à des périodes déterminées. Nous allons montrer que dans ce cas très simple, l'algorithme glouton s'applique. Dans des cas plus complexes, l'algorithme donne une solution approchée, dont on se contente souvent, vu le temps de calcul prohibitif de la recherche de l'optimum exact.

8.1.1 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 $d(e)$ la date du début de la location et $f(e) > d(e)$ la date de fin. 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. L'algorithme glouton s'exprime comme suit:

- *Etape 1:* Classer les éléments de E par ordre des dates de fins croissantes. Les éléments de E constituent alors une suite $e_1, e_2, \ldots e_n$ telle que $f(e_1) \leq f(e_2), \ldots \leq f(e_n)$
- Initialiser $F := \emptyset$
- *Etape 2:* 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 a bien obtenu ainsi la solution optimale.

Soit $F = \{x_1, x_2, \ldots x_p\}$ la solution obtenue par l'algorithme glouton et soit $G = \{y_1, y_2, \ldots y_q\}, q \geq p$ une solution optimale. Dans les deux cas nous supposons que les demandes sont classées par dates de fins croissantes. Nous allons montrer que $p = q$. Supposons que $\forall i < k$, on ait $x_i = y_i$ et que k soit le plus petit entier tel que $x_k \neq y_k$, alors par construction de F on a: $f(y_k) \geq f(x_k)$. On peut alors remplacer G par $G' = \{y_1, y_2, \ldots y_{k-1}, x_k, y_{k+1}, y_q\}$ tout en satisfaisant à la contrainte de non chevauchement des demandes, ainsi G' est une solution optimale ayant plus d'éléments en commun avec F que n'en avait G. En répétant cette opération suffisamment de fois on trouve un ensemble H de même cardinalité que G et qui contient F. L'ensemble H ne peut contenir d'autres éléments car ceux-ci auraient été ajoutés à F par l'algorithme glouton, ceci montre bien que $p = q$.

Remarques

1. Noter que le choix de classer les demandes par dates de fin croissantes est important. Si on les avait classées, par exemple, par dates de début croissantes, on n'aurait pas obtenu le résultat. On le voit sur l'exemple suivant avec trois demandes e_1, e_2, e_3 dont les dates de début et de fin sont données par le tableau suivant:

	e_1	e_2	e_3
d	2	3	5
f	8	4	8

Bien entendu, pour des raisons évidentes de symétrie, le classement par dates de début décroissantes donne aussi le résultat optimal.

2. On peut noter aussi que 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). De fait, le problème de la maximisation de cette durée totale est NP complet, il est donc illusoire de penser trouver un algorithme simple et efficace.

3. S'il y a plus d'une ressource à affecter, par exemple deux voitures à louer, l'algorithme glouton consistant à classer les demandes suivant les dates de fin et à affecter la première ressource disponible, ne donne pas l'optimum.

8.1.2 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 symétrique, il prend dans ce cas particulier le nom d'algorithme de Kruskal. Décrivons brièvement le problème et l'algorithme.

Un *graphe symétrique* est donné par un ensemble X de sommets et un ensemble A d'arcs tel que, 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. Le couple $\{a, \bar{a}\}$ forme une *arête*. Un *arbre* est un graphe symétrique tel que tout couple de sommets est relié par un chemin (connexité) et qui ne possède pas de circuit (autres que ceux formés par un arc et son opposé). Pour un graphe symétrique $G = (X, A)$ quelconque, un arbre recouvrant est donné par un sous ensemble de l'ensemble des arêtes qui forme un arbre ayant X pour ensemble de sommets (voir figure 8.1). Pour posséder un arbre recouvrant, un graphe doit être connexe. Dans ce cas, les arborescences construites par les algorithmes décrits au chapitre 5 sont des arbres recouvrants. Lorsque chaque arête du graphe est affectée d'un certain poids, se pose le problème de la recherche d'un arbre recouvrant de poids minimal (c'est à dire un arbre dont la somme des poids des arêtes soit minimale). Une illustration de ce problème est la réalisation d'un réseau électrique ou informatique entre différents points, deux points quelconques doivent toujours être reliés entre eux (connexité) et on doit minimiser le coût de la réalisation. Le poids d'une arête est, dans ce contexte, le coût de construction de la portion du réseau reliant ses deux extrémités.

On peut facilement formuler le problème dans le cadre général donné en début de chapitre: E est l'ensemble des arêtes 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 arêtes par ordre de poids croissants. Elles constituent alors une suite
$$e_1, e_2, \dots e_n$$

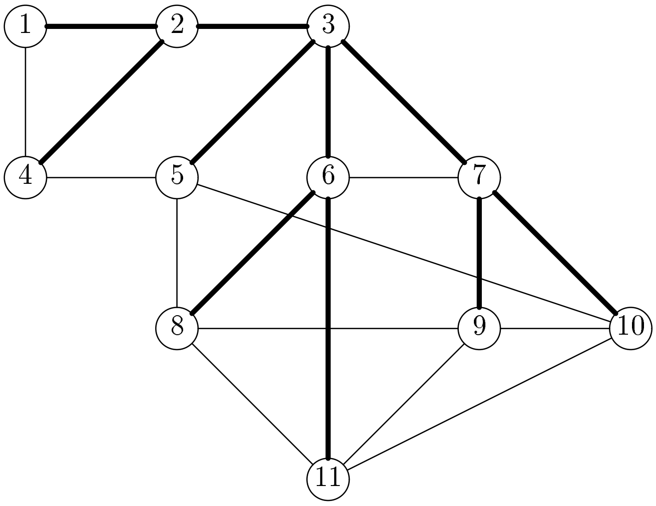


FIG. 8.1 – *Un graphe symétrique et l'un de ses arbres recouvrants*

telle que

$$p(e_1) \leq p(e_2), \ldots \leq p(e_n)$$

– Initialiser $F := \emptyset$

– *Etape 2:* 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:

> 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:

> 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 8.2.

8.2 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.

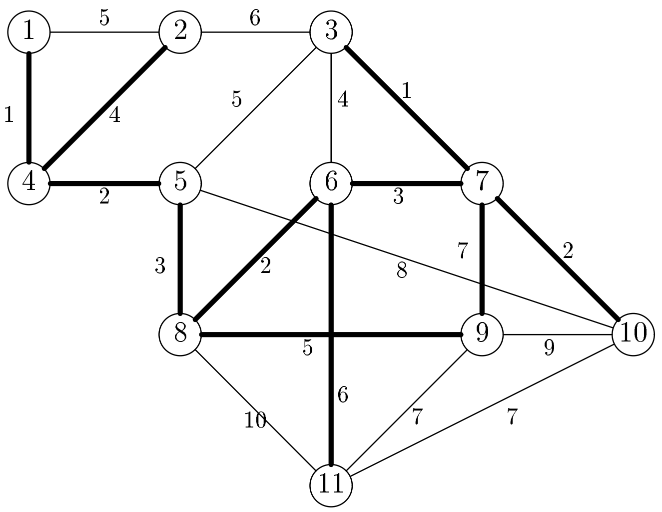


FIG. 8.2 – *Un arbre recouvrant de poids minimum*

8.2.1 Sac à dos

Prenons pour premier exemple le problème dit du *sac à dos*; soit un ensemble E d'objets chacun ayant un certain poids, un entier positif noté $p(e)$, et soit M un réel qui représente la charge maximum que l'on peut emporter dans un sac à dos. La question est de 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. Il est assez facile de trouver des exemples pour lesquels l'algorithme glouton ne donne pas le bon résultat, il suffit en effet de considérer 4 objets de poids respectifs $4, 3, 3, 1$ pour remplir un sac de charge maximum égale à 6. On s'apperçoit que si l'on remplit le sac en présentant les objets en ordre de poids décroissants et en retenant ceux qui ne font pas dépasser la capacité maximale, on obtiendra une charge égale à 5. Si à l'opposé, on classe les objets par ordre de poids croissants, et que l'on applique l'algorithme glouton, la charge obtenue sera égale à 4, alors qu'il est possible de remplir le sac avec deux objets de poids 3 et d'obtenir l'optimum.

Le problème du sac à dos, lorsque la capacité du sac n'est pas un entier,[1] 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. Une bonne programmation de cette exploration systématique consiste à utiliser la récursivité. Notons n le nombre d'éléments de E, nous utiliserons un tableau `sac` permettant de coder toutes les possibilités, un objet i est mis dans le sac si `sac[i] = true`, il n'est pas mis si `sac[i] = false`. Il faut donc parcourir tous les vecteurs possibles de booléens, pour cela on considère successivement toutes les positions $i = 1, \ldots n$ et on effectue les deux choix possibles pour `sac[i]` en ne choisissant pas la dernière possibilité si l'on dépasse la capacité du sac. On utilise un entier `meilleur` qui 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. Un tableau `msac` garde en mémoire le contenu du sac qui réalise ce minimum. La procédure récursive `calcul(i,u)` a pour paramètres d'appel, `i` l'objet pour lequel on doit prendre une décision, et `u` la capacité disponible restante. Elle considère deux possibilités pour l'objet `i` l'une pour laquelle il

1. Dans le cas où M est en entier, on peut trouver un algorithme très efficace fondé sur la programmation dynamique.

est mis dans le sac (si on ne dépasse pas la capacité restante `u`), l'autre pour laquelle il n'y est pas mis. La procédure appelle `calcul(i+1, u)` et `calcul(i+1, u - p[i])`. Ainsi le premier appel de `calcul(i, u)` est fait avec `i = 0` et `u` égal à la capacité `M` du sac, les appels successifs feront ensuite augmenter `i` (et diminuer `u`) jusqu'à atteindre la valeur `n`. Le résultat est mémorisé s'il améliore la valeur courante de `meilleur`.

```
static void calcul (int i, float u) {
    if (i >= n) {
        if (u < meilleur) {
            for (int j = 0; i < n; ++i)
                msac[i] = sac[i];
            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 procédure 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.

8.2.2 Placement de reines sur un échiquier

Le placement de reines sur un échiquier sans qu'aucune d'entre elles ne soit en prise par une autre constitue un autre exemple de recherche arborescente. Là encore il faut parcourir l'ensemble des solutions possibles. Pour les valeurs successives de `i`, on place une reine sur la ligne i et sur une colonne `j = pos[i]` en vérifiant bien qu'elle n'est pas en prise. Le tableau `pos` que l'on remplit récursivement contient les positions des reines déjà placées. Tester si deux reines sont en conflit est relativement simple. Notons i_1, j_1 et i_2, j_2 leurs positions respectives (ligne et colonne) il y a conflit si $i_1 = i_2$ (elles sont alors sur la même ligne), ou si $j_1 = j_2$ (même colonne) ou si $|i_1 - i_2| = |j_1 - j_2|$ (même diagonale).

```
static boolean conflit (int i1, int j1, int i2, int j2) {

  return (i1 == i2) || (j1 == j2) ||
         (Math.abs (i1 - i2) == Math.abs (j1 - j2));
}
```

Celle-ci peut être appelée plusieurs fois pour tester si une reine en position `i, j` est compatible avec les reines précédemment placées sur les lignes $1, \ldots, i-1$:

```
static boolean compatible (int i, int j) {

    for (int k = 0; k < i; ++k)
        if (conflit (i, j, k, pos[k]))
            return false;
    return true;
}
```

La fonction récursive qui trouve une solution au problème des reines est alors la suivante:

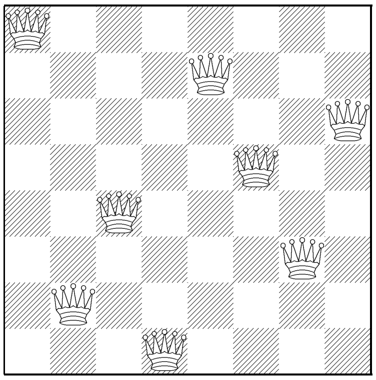


FIG. 8.3 – *Huit reines sur un échiquier*

```
static void reines (int i) {

  if (i >= nbReines)
      imprimerSolution();
  else {
      for (int j = 0; j < nbReines; ++j)
          if compatible (i, j) {
              pos[i] = j;
              reines (i+1);
          }
  }
}
```

La boucle `for` à l'intérieur de la procédure permet de parcourir toutes les positions sur la ligne `i` compatibles avec les reines déjà placées. Les appels successifs de `Reines(i)` modifient la valeur de `pos[i]` déterminée par l'appel précédent. La procédure précédente affiche toutes les solutions possibles, il est assez facile de modifier la procédure en s'arrêtant dès que l'on a trouvé une solution ou pour simplement compter le nombre de solutions différentes. En lançant `Reines(0)`, on trouve ainsi 90 solutions pour un échiquier 8×8 dont l'une d'elles est donnée figure 8.3.

Remarque 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.

8.3 Programmation dynamique

Pour illustrer la technique d'exploration appelée programmation dynamique, le mieux est de commencer par un exemple. Nous considérons ainsi la recherche de chemins de longueur minimale entre tous les couples de points d'un graphe aux arcs valués.

8.3.1 Plus courts chemins dans un graphe

Dans la suite, on considère un graphe $G = (X, A)$ ayant X comme ensemble de sommets et A comme ensemble d'arcs. On se donne une application l de A dans l'ensemble des entiers naturels, $l(a)$ 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 donner un algorithme qui détermine 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 (en fait il suffit dans la suite de remplacer ∞ par un nombre suffisamment grand par exemple la somme des longueurs de tous les arcs du graphe). La construction effective des chemins sera examinée ensuite. On suppose qu'entre deux sommets il y a au plus un arc. En effet, s'il en existe plusieurs, il suffit de ne retenir que le plus court.

Les algorithmes de recherche de chemins les plus courts reposent sur l'observation très simple mais importante suivante:

Remarque *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 $x_1, x_2, \ldots x_n$ 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 $l(a)$ 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.

Lemme Les relations suivantes sont satisfaites par les δ_k:

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

Preuve 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. Il a donc pour longueur: $\delta_k(x_i, x_k) + \delta_k(x_k, x_j)$.

L'algorithme suivant pour la recherche du plus court chemin met à jour une matrice `delta[i,j]` qui a été initialisée par les longueurs des arcs et par un entier suffisamment grand 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.

```
for (k = 0; k < n; ++k)
```

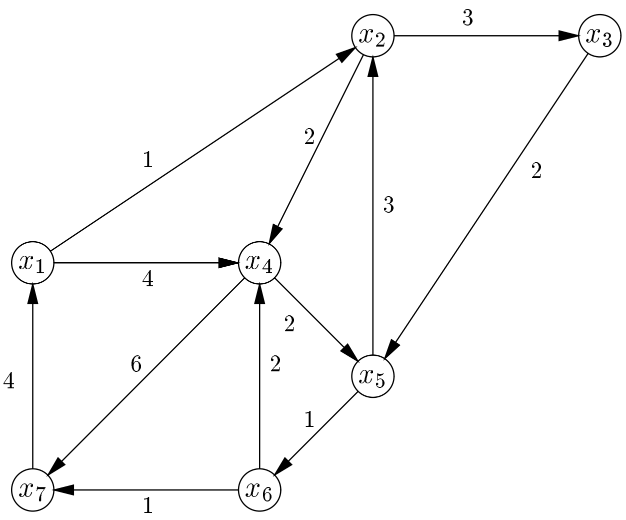


FIG. 8.4 – *Un graphe aux arcs valués*

```
for (i = 0; i < n; ++i)
   for (j = 1; j < n; ++j)
      delta[i][j] = Math.min(delta[i][j], delta[i][k] + delta[k][j]);
```

On note la similitude avec l'algorithme de recherche de la fermeture transitive d'un graphe exposé au chapitre 5. Sur l'exemple du graphe donné sur la figure 8.4, on part de la matrice δ_1 donnée par

$$\delta_1 = \begin{pmatrix} 0 & 1 & \infty & 4 & \infty & \infty & \infty \\ \infty & 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & 0 & 2 & \infty & 6 \\ \infty & 3 & \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 2 & \infty & 0 & 1 \\ 4 & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Après le calcul on obtient:

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

Pour le calcul effectif des chemins les plus courts, on utilise une matrice qui contient $Suiv[i,j]$, le sommet qui suit i dans le chemin le plus court qui va de i à j. Les valeurs $Suiv[i,j]$ sont initialisées à j s'il existe un arc de i vers j et à -1 sinon, $Suiv[i,i]$ est lui initialisé à i. Le calcul précédent qui a donné δ peut s'accompagner de celui de $Suiv$ en procédant comme suit:

```
for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
          if (delta[i][j] > (delta[i][k] + delta[k][j]) {
              delta[i][j] =  delta[i][k] + delta[k][j];
              suivant[i][j] = suivant[i][k];
          }
```

Une fois le calcul des deux matrices effectué on peut retrouver le chemin le plus court qui joint i à j par la procédure:

```
static void plusCourtChemin(int i, int j) {

  for (int k = i; k != j; k = suivant[k][j])
      System.out.print (k + " ");
  System.out.println (j);
}
```

Sur l'exemple précédent on trouve:

$$Suiv = \begin{pmatrix} 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 2 & 3 & 4 & 4 & 4 & 4 \\ 5 & 5 & 3 & 5 & 5 & 5 & 5 \\ 5 & 5 & 5 & 4 & 5 & 5 & 5 \\ 6 & 2 & 2 & 6 & 5 & 6 & 6 \\ 7 & 7 & 7 & 4 & 4 & 6 & 7 \\ 1 & 1 & 1 & 1 & 1 & 1 & 7 \end{pmatrix}$$

8.3.2 Sous-séquences communes

On utilise aussi un algorithme de programmation dynamique pour rechercher des sous-séquences communes à deux séquences données. précisons tout d'abord quelques définitions. Une *séquence* (ou un *mot*) est une suite finie de symboles (ou *lettres*) pris dans un ensemble fini (ou *alphabet*). Si $u = a_1 \cdots a_n$ est une séquence, où $a_1, \ldots, a_n$ sont des lettres, l'entier n est la *longueur* de u. Une séquence $v = b_1 \cdots b_m$ est une *sous-séquence* de $u = a_1 \cdots a_n$ s'il existe des entiers $i_1, \ldots, i_m$, $(1 \leq i_1 < \cdots i_m \leq n)$ 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.

On cherche à déterminer la longueur d'une sous-séquence commune maximale à $u = a_1 \cdots a_n$ et $v = b_1 \cdots b_m$. Pour cela, on note $L(i,j)$ la longueur d'une sous-séquence commune maximale aux mots $a_1 \cdots a_i$ et $b_1 \cdots b_j$, $(0 \leq j \leq m,\ 0 \leq i \leq n)$. On peut montrer que

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

En effet, soit w une sous séquence de longueur maximale, commune à $a_1 \cdots a_{i-1}$ et à $b_1 \cdots b_{j-1}$ si $a_i = b_j$, wa_i est une sous-séquence commune maximale à $a_1 \cdots a_i$ et $b_1 \cdots b_j$. Si $a_i \neq b_j$ alors une sous-séquence commune à $a_1 \cdots a_i$ et $b_1 \cdots b_j$ est ou bien commune à $a_1 \cdots a_i$ et $b_1 \cdots b_{j-1}$ (si elle ne se termine pas par b_j); ou bien à $a_1 \cdots a_{i-1}$ et $b_1 \cdots b_j$, (si elle ne se termine par a_i). On obtient ainsi l'algorithme qui permet de déterminer la longueur d'une sous séquence commune maximale à $a_1 \cdots a_n$ et $b_1 \cdots b_m$

```
static void longueurSSC (String u, String v) {
```

```
    int n = u.length();
    int m = v.length();
    int longueur[][] = new int[n][m];
    for (int i = 0; i < n; ++i) longueur[i, 0] = 0;
    for (int j = 0; j < m; ++j) longueur[0, j] = 0;
    for (int i = 1; i < n; ++i)
        for (int j = 1; j < m; ++j)
            if (u.charAt(i) == v.charAt(j))
                 longueur[i][j] = 1 + longueur[i-1][j-1];
            else if longueur[i][j-1] > longueur[i-1][j]
                 longueur[i][j] = longueur[i][j-1];
            else
                 longueur[i][j] = longueur[i-1][j];
  }
```

Il est assez facile de transformer l'algorithme pour retrouver une sous-séquence maximale commune au lieu de simplement calculer sa longueur. Pour cela, on met à jour un tableau `provient` qui indique lequel des trois cas a permis d'obtenir la longueur maximale.

```
  static void longueurSSC (String u, String v,
                           int[][] longueur, int[][] provient) {
    int n = u.length();
    int m = v.length();
    for (int i = 0; i < n; ++i) longueur[i, 0] = 0;
    for (int j = 0; j < m; ++j) longueur[0, j] = 0;
    for (int i = 1; i < n; ++i)
        for (int j = 1; j < m; ++j)
            if (u.charAt(i) == v.charAt(j)) {
                 longueur[i][j] = 1 + longueur[i-1][j-1];
                 provient[i,j] = 1;
            } else if longueur[i][j-1] > longueur[i-1][j] {
                 longueur[i][j] = longueur[i][j-1];
                 provient[i,j] = 2;
            } else {
                 longueur[i][j] = longueur[i-1][j];
                 provient[i,j] = 3
            }
  }
```

Une fois ce calcul effectué il suffit de remonter à chaque étape de `i,j` vers `i-1, j-1`, vers `i, j-1` ou vers `i-1,j` en se servant de la valeur de `provient[i,j]`.

```
  static String ssc (String u, String v) {

    int n = u.length();
    int m = v.length();
    int longueur[][] = new int[n][m];
    int provient[][] = new int[n][m];
    longueurSSC (u, v, longueur, provient);

    int lg = longueur[n][m];
    StringBuffer res = new StringBuffer(lg);
    int i = n, j = m;
    for (int k = lg-1; k >= 0; )
        switch (provient[i][j]) {
```

```
        case 1: res.setCharAt(k, u.charAt(i-1)); --i; --j; --k;
                break;
        case 2: --j; break;
        case 3: --i; break;
        }
    return new String(res);
}
```

Remarque La recherche de sous-séquences communes à deux séquences intervient parmi les nombreux problèmes algorithmiques posés par la recherche des propriétés des séquences représentant le génome humain.

8.4 Programmes en Caml

```
(* les n reines, voir page 182 *)

let nReines n =
  let pos = make_vect 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 = vect_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 = make_matrix (n+1) (m+1) 0 and
       provient = 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 = create_string 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;;
```

Annexe A

Java

Le langage Java a été conçu par J. Gosling et ses collègues à Sun microsystems. 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. Très rapidement, le langage est devenu populaire. De multiples livres le décrivent, et il est un peu vain de vouloir les résumer ici, nous nous contenterons d'un minimum. Parmi ces livres, les plus intéressants nous semblent dans l'ordre [j1,j2,j3,j4].

Voici trois exemples simples de programmes, ceux qui souhaitent apprendre Java en examinant un programme plus fourni peuvent se reporter directement au paragraphe *Carrés magiques*.

A.1 Exemples simples

Le premier est un classique, il s'agit simplement d'afficher `Bonjour !` à l'écran.

```
/* Voici mon premier programme */
import java.io.*; // On a besoin de la bibliothèque d'entrées et sorties

class PremierProg {

  public static void main (String args[]){

      System.out.println("Bonjour!");
  }
}
```

Pour exécuter ce programme il faut commencer par l'éditer, pour cela on utilise un éditeur de texte (par exemple emacs) pour créer un fichier `Premier.java` qui le contient. Après cela on le compile par la commande :

```
javac Premier.java
```

Ceci a pour effet de faire construire par le compilateur un fichier `PremierProg.class`, enfin il faut exécuter ce programme par la commande `java` suivi du nom de la classe (dans ce cas c'est la fonction `main` de la classe qui est exécutée) :

```
java PremierProg
```

On voit s'afficher :

```
Bonjour!
```

Analysons le texte de ce premier programme, il contient d'abord un commentaire entre les deux symboles `/*` et `*/`, puis un appel à la bibliothèque d'entrées sorties, un nouveau commentaire de fin de ligne qui commence après le symbole `//` et se termine à la fin de la ligne; ces commentaires ne sont utiles qu'à des lecteurs (humains) du texte du programme, ils n'auront aucun effet sur la compilation ou l'exécution.

Dans le programme, il y a une seule classe `PremierProg`, et une seule fonction (on dit aussi méthode) dans cette classe; cette fonction a pour nom `main`, elle affiche la chaîne de caractères `Bonjour!` qui est donnée sous forme d'un paramètre littéral à la fonction `println`. Cette dernière fonction se trouve dans la partie `System.out` de la bibliothèque (d'où la façon de l'invoquer par `System.out.println`).

Un deuxième exemple est le suivant :

```
class Additionner  {

  public static void main (String args[]) {
    if (args.length != 2)
        System.out.println("mauvais nombre d'arguments");
    else {
        int s = Integer.parseInt(args[0]) + Integer.parseInt(args[1]);
        System.out.println (s);}
  }
}
```

Ce programme affiche la somme des deux nombres entiers qui lui sont donnés; dans le cas où il y plus ou moins de 2 nombres, un message d'erreur est affiché. Pour effectuer l'addition des deux nombres il faut les donner en *arguments* à la suite de l'appel de `java Additionner`, ainsi :

```
java Additionner 47 51
```

répond :

```
98
```

On voit de nouvelles choses sur ce programme, d'abord il y a un `if` suivi d'une expression booléenne (on dit aussi logique). Cette expression fait intervenir la variable `args`, qui représente la suite d'arguments suivant la commande `java Additionner`. Plus généralement `args.length` est le nombre d'arguments, `args[0]` est le premier, `args[1]` est le second etc. L'expression est ainsi vraie si le nombre d'arguments est différent de 2 (le symbole `!=` signifie $\neq$ en C et en Java) ; dans ce cas le programme affiche un message et puis s'arrête. Si la condition après le `if` n'est pas satisfaite, c'est ce qui se trouve après `else` qui est exécuté, ici il s'agit d'une suite d'instructions entre `{` et `}`. La première instruction calcule la somme de deux entiers en commençant par transformer les paramètres, qui sont des chaînes de caractères, en des entiers par la fonction `parseInt`. La seconde affiche cette valeur.

Un troisième programme additionne des nombres entiers en quantité quelconque.

```
class Somme  {
     public static void main (String args[]){
         int s = 0;
         for (int i = 0; i < args.length; i++)
             s = s + Integer.parseInt(args[i]);
         System.out.println (s);
         }
}
```

Ainsi,

```
java Somme 34 12 1 5 46
```

affiche aussi 98.

Dans le texte de ce programme, on voit apparaître la boucle `for` qui contient trois sous-instructions, entre parenthèses, séparées par des `;` et un corps de boucle qui est ici : `s = s + Integer.parseInt(args[i])`. La première sous-instruction du `for` n'est exécutée qu'une seule fois au début, elle initialise la variable `i` à 0. La seconde sous-instruction est une expression logique (test) qui est un test de suite de calcul : si celle-ci n'est pas satisfaite on sort de la boucle, sinon on exécute le corps de la boucle et on va à la troisième sous-instruction. Cette troisième sous-instruction est une instruction qui est exécutée après le corps, ici il s'agit d'une incrémentation : `i++;` signifie incrémenter `i` (ajouter 1 à `i`); une fois cette incrémentation effectuée, on revient au test. Lorsqu'on sort de la boucle `for`, une instruction demande que le résultat (contenu dans la variable `s`) soit affiché.

A.2 Symboles

Dans un programme il est indispensable de savoir reconnaître les différents symboles qui y apparaissent : espacements, commentaires, séparateurs, mots réservés, littéraux, opérateurs, identificateurs.

A.2.1 Séparateurs, commentaires

Un *espacement* dans un programme est une suite d'espaces, de tabulations ou de passages à la ligne. Un espacement sert pour séparer les noms des variables des opérations et des autres symboles.

Il y a aussi 9 caractères qui servent de *séparateurs*, il s'agit de

```
(   )   [   ]   {   } .   ,   ;
```

On verra leur utilisation plus loin.

Un *commentaire* est du texte qui se trouve soit entre les deux symboles `/*` et `*/`, soit entre le symbole `//` et la fin d'une ligne. Rappelons que le but du commentaire est d'aider un lecteur à comprendre le programme que l'on a écrit (noter que ce lecteur peut très bien être soi même!) le compilateur ne les considère pas.

A.2.2 Mots réservés, opérateurs

Les mots suivants ont une signification particulière en Java et ne peuvent être utilisés pour désigner autre chose. Il s'agit de mots qui sont utilisés dans une instruction ou une déclaration. Un bon programmeur doit connaître la signification de tous ces mots, on se limitera dans le cours à en expliquer une petite partie, de l'ordre d'une vingtaine.

```
abstract     boolean      break        byte          case
catch        char         class        const         continue
default      do           double       else          extends
final        finally      float        for           goto
if           implements   import       instanceof    int
interface    long         native       new           package
private      protected    public       return        short
static       super        switch       synchronized  this
throw        throws       transient    try           void
volatile     while
```

Les opérateurs sont au nombre de 37, ils servent à exprimer des opérations sur les nombres, caractères, ou les valeurs booléennes (vrai, faux).

Les plus courants sont :

cp cori/x-annee0/

```
+   -   *   /   %   =   ==   !=   &&   ||   <   >   <=   >=   ++   --
```

A.2.3 Identificateurs, littéraux

Les *identificateurs* sont des séquences de lettres et de chiffres commençant par une lettre. Les identificateurs sont séparés par des espacements, des séparateurs ou des opérateurs; les mots réservés ne peuvent pas être utilisés comme identificateurs.

Les *littéraux* sont des constantes qui désignent un entier (comme 1024 ou -67), un flottant (comme 3.14159 ou 5e-9) , un booléen (`true` ou `false`) un caractère (entre deux `'`), une chaîne de caractères (entre deux `"`) Voici quelques exemples de littéraux, on en verra d'autres plus loin.

```
234   -12   3.14  9e-7  true  'c' "Bonjour!"
```

A.3 Types primitifs

En Java les variables et les expressions doivent avoir un *type*, il y a trois sortes de types : les types primitifs sont les plus simples, le type tableau, et les types définis par une classe (il y a aussi des types définis par une interface, mais cette notion sort du cadre d'une initiation). Le type d'une variable doit être déclaré n'importe où avant l'utilisation de la variable, pour cela on indique le nom du type avant le nom de la variable. Par exemple la suite suivante déclare que les variables `a, b, c` sont de type entier, que `x` est de type réel (on dit flottant) et que `mot` est une chaîne de caractères:

```
int a, b, c;
float x;
String mot;
```

A.3.1 Numérique entier

Le type d'un entier est `byte`, `short`, `int` ou `long`, selon qu'on le représente 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` $= 2^{31}-1$, des entiers longs `Long.MIN_VALUE` $= -2^{63}$, `Long.MAX_VALUE` $= 2^{63}-1$, `Byte.MIN_VALUE` $= -128$, `Byte.MAX_VALUE` $= 127$, etc.

A.3.2 Numérique flottant

Le type d'un réel est `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.

A.3.3 Booléens

Les variables booléennes sont déclarées par le type `boolean`, elles ne peuvent prendre que deux valeurs : `true` et `false` qui sont aussi des littéraux, on peut ainsi écrire

```
boolean fini = true;
```

A.3.4 Caractères

Le type des caractères est `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, noter que la classe `String` des chaînes de caractères n'est pas un type primitif.

A.4 Expressions

A.4.1 Expressions arithmétiques

Une expression arithmétique est formée de variables, ou de constantes qui sont des nombres et d'opérateurs qui sont les opérations mathématiques habituelles. La notation est celle couramment utilisée en mathématiques, les opérations usuelles sont `+,-,*,/, %` pour l'addition, soustraction, multiplication, division et le reste de la division (modulo). Il y a des règles de priorité, ainsi l'opération de multiplication a une plus grande priorité que l'addition, cela signifie que les multiplications sont faites avant les additions. La présence de parenthèses permet de mieux contrôler le résultat. Par exemple `3 + 5 * 6` est évalué à 33; par contre `(3 + 5) * 6` est évalué 48. Un expression a toujours un type et le résultat de son évaluation est une valeur ayant ce type. L'évaluation de `2/0` soulève l'exception (voir plus loin) `ArithmeticException`.

A.4.2 Expressions logiques

Une expression logique a pour résultat un booléen. Par exemple, le test d'égalité `a == b` a pour résultat `true` si les deux nombres (ou les deux booléens, ou les deux caractères) `a` et `b` sont égaux, et `false` sinon. La signification de `a == b` dans d'autres cas, nécessite de connaître la notion de référence, ce qui n'est pas notre propos ici. L'opérateur `!=` (différent de) a un résultat opposé à celui de `==`. Les opérateurs de connection logique *et, ou, non* sont représentés respectivement par `&&, ||, !` les deux opérateurs `&&, ||` sont les seuls pour lesquels l'évaluation des deux opérandes n'est pas faite systématiquement pour obtenir le résultat. Ainsi, l'évaluation de `u && v` commence par celle de `u` si on obtient `false` le résultat global est `false` sans qu'il soit fait l'évaluation de `v`. Par exemple `(4 == 5) && (2/0 == 0)` s'évalue à `false` alors que `(2/0 == 0)&& (4 == 5)` soulève une exception. L'opérateur `||` suit une règle similaire.

Pour tout autre opérateur, tous les opérandes sont systématiquement évalués avant d'effectuer l'opération.

A.4.3 Concaténation

Le symbole `+` désigne aussi la concaténation (mise bout à bout) de deux chaînes de caractères, le contexte permet de déterminer s'il s'agit de l'addition de deux nombres ou de la concaténation. On dit qu'il y a surcharge de l'opération `+`.

A.4.4 Expressions sur les bits

Les opérations sur les *bits* peuvent se révéler utiles. On peut faire `&` (*et* logique), `|` (*ou* logique), `^` (*ou* exclusif), `<<` (décalage vers la gauche), `>>` (décalage vers la droite), `~` (complément à 1). 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 booléen des opérations `&` et `|` sur les *bits* qui donnent une valeur entière. Par exemple, si `x` vaut 1 et `y` vaut 2, `x & y` vaut 0.

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.

Le complément à 1 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;
```

A.4.5 Précédences

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, on utilise le parenthésage en cas d'égalité

<table>
<tr><th>Opérateurs</th><th>Associativité</th><th>Signification</th></tr>
<tr><td>() [] .</td><td>gauche à droite</td><td>fonction, tableau, champs</td></tr>
<tr><td>! ~ ++ -- + -</td><td>droite à gauche</td><td>opérateurs unaires</td></tr>
<tr><td>* / %</td><td>gauche à droite</td><td>opérateurs multiplicatifs</td></tr>
<tr><td>+ -</td><td>gauche à droite</td><td>opérateurs additifs</td></tr>
<tr><td><< >></td><td>gauche à droite</td><td>décalages</td></tr>
<tr><td>< <= > >=</td><td>gauche à droite</td><td>comparaisons numériques</td></tr>
<tr><td>== !=</td><td>gauche à droite</td><td>tests d'égalité</td></tr>
<tr><td>&</td><td>gauche à droite</td><td>et sur chaque bit</td></tr>
<tr><td>|</td><td>gauche à droite</td><td>ou sur chaque bit</td></tr>
<tr><td>&&</td><td>gauche à droite</td><td>et logique</td></tr>
<tr><td>||</td><td>gauche à droite</td><td>ou logique</td></tr>
<tr><td>?:</td><td>droite à gauche</td><td>if then else</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>droite à gauche</td><td>affectations</td></tr>
</table>

En règle générale, il est conseillé de mettre des parenthèses si les précédences ne sont pas claires.

A.5 Instructions

Toute expression suivie d'un point-virgule devient une instruction. Ainsi le 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.

A.5.1 Affectations

L'instruction la plus courante est l'affectation, elle s'écrit :

```
a = e;
```

elle consiste à évaluer l'expression `e`, puis à affecter la valeur ainsi obtenue à la variable `a`.

A titre anecdotique, les opérateurs d'affectation peuvent être plus complexes que le simple `=` et permettent des abréviations parfois utiles. Ainsi, si *op* est un des opérateurs `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, ou `|`,

e_1 *op*`=` e_2

est un raccourci pour

e_1 `=` e_1 *op* e_2

L'affectation peut être considérée comme un opérateur dans une expression, le résultat est 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 précaution, car leur effet peut dépendre d'un ordre d'évaluation très complexe. Il est par exemple peu recommandé de mettre plus d'un effet de bord dans une expression.

A.5.2 Incrémentation

L'instruction `i++;` ou `++i;` est une *incrémentation*, elle s'applique à une variable de type entier et a pour effet d'ajouter 1 à sa valeur. En remplaçant `++` par `--` on retranche 1 au lieu d'ajouter (décrémentation). Ainsi :

```
++i;
j++;
```

a le même résultat que :

```
i = i + 1;
j = j + 1;
```

L'incrémentation peut aussi être effectuée à l'intérieur d'une expression, elle a alors un effet de bord. il y a des opérations de pré-incrémentation, de post-incrémentation, de pré-décrémentation, de post-décrémentation qui permettent de fournir la valeur d'une variable tout en l'incrémentant ou la décrémentant (lui ajouter ou retrancher 1) avant ou après . Par exemple, 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.
De manière similaire, 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 séparer l'expression en plusieurs morceaux pour isoler la partie effet de bord.

On doit éviter les effets de bord dans une expression

A.5.3 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. Ce qui 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

$$\begin{tikzcd} \texttt{double} \\ \texttt{float} \arrow[u] \\ \texttt{long} \arrow[u] \\ \texttt{int} \arrow[u] & \texttt{char} \arrow[l] \\ \texttt{short} \arrow[u] \\ \texttt{byte} \arrow[u] \end{tikzcd}$$

FIG. A.1 – *Conversions implicites*

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.

A.5.4 Instructions conditionnelles

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.

Parfois, on peut trouver un peu long d'écrire

```
if (a > b)
    z = a;
```

```
else
    z = b;
```

On peut alors utiliser l'expression conditionnelle ˜ :

e_1 *?* e_2 *:* e_3

Celle-ci évalue e_1 d'abord, si l'on obtient `true`, le résultat est e_2, sinon c'est e_3. Donc le maximum de `a` et `b` peut s'écrire

```
z = (a > b) ? a : b;
```

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 signification bien particulière. Pour sortir de l'instruction, il faut passer par une instruction `break`. Sinon, le reste de l'instruction est exécuté 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.

A.5.5 Itérations

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
```

$\quad S$
while (E);

L'instruction d'itération la plus puissante est l'instruction `for`. Sa syntaxe est

for (E_1; E_2; E_3)
$\quad S$

qui est équivalente à

E_1;
while (E_2) {
$\quad S$;
$\quad E_3$;
}

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 est plus complexe (voir le paragraphe sur le hachage dans le premier chapitre du cours Algorithme et Programmation)

```
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;
      ...
    }
  }
```

A.6 Tableaux

A.6.1 Déclaration, construction

Les tableaux permettent d'utiliser plusieurs variables de même type, ainsi avec la déclaration et la construction :

```
int v[] = new int[101];
```

ou la suivante qui est équivalente :

```
int[]  v = new int[101];
```

on a à sa disposition 101 variables de type `int` que l'on peut affecter et évaluer. Chacune de ces variables est accessible par `a[i]`, où `i` est un entier compris entre 0 et 100. Tous les tableaux commencent par l'indice égal à 0. Si `i` est en dehors de ces limites, une exception `IndexOutOfBoundsException` est soulevée. La notation `v.length` indique la longueur du tableau `v`.

La construction usuelle de parcours des éléments d'un tableau fait itervenir une boucle `for`, voici par exemple le calcul du plus petit élément d'un tableau (non vide)

```
int min = v[0];
for (int i = 0; i < v.length; ++i)
    if (v[i] < min) min = v[i];
```

On verra que les tableaux sont en fait des objets, de ce point de vue `length` est un champs associé à l'objet.

A.6.2 Matrices

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. Pour déclarer une matrice de taille 10×10 on écrit :

```
int[][] mat = new int[10][10];
```

Comme on l'a vu, 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}};
```

A.7 Fonctions, Procédures

Chaque classe contient une suite linéaire de fonctions ou procédures (on dit méthode dans la terminologie des langage à objets). Il n'y a pas de fonctions emboîtées les unes dans les autres. 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 `{ }`. Une fonction obtient un résultat, celui-ci est rendu par l'instruction `return`. 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 ce résultat.

A.7.1 Résultat void

Lorsqu'aucun résultat n'est rendu, on dit que l'on a affaire à une procédure et le mot réservé `void` précède la déclaration de méthode. Dans les deux procédures qui suivent, le résultat est 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).

A.7.2 Variables locales, globales

Il peut y avoir des variables locales dans une procédure, plus exactement dans toute instruction composée entourée par des accolades. Les variables globales sont elles 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.

Les paramètres de type primitif des fonctions sont passés par valeur

Noter que cette règle ne s'applique pas aux types plus complexes, en particulier aux tableaux.

A.8 Classes

Une classe est à la fois la déclaration d'un type non primitif et d'une série de fonctions ou procédures associées. 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`.

A.8.1 Constructeurs

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 pour décrire un constructeur on écrit le nom de la classe suivi des paramètres qui permettent d'effectuer la construction, voici un constructeur de dates :

```
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 (i.e. le nom de la classe où il se trouve) et ses paramètres. Le corps de la fonction est quelconque, mais on ne fait pas figurer d'instruction `return` , puisque son résultat est toujours l'objet en cours de création. Le constructeur est appelé après le mot réservé `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 résérvé `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 réservé n'était pas indispensable, 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;
};
```

A.8.2 Champs statiques

Un champ peut être déclaré statique. Cela signifie qu'il n'existe qu'à un seul exemplaire pour tous les objets de 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.

A.8.3 Méthodes statiques et dynamiques.

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 déclarations de 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 statiques, 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 dynamiquement 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 écrire

```
if (!berlin.equals(bastille))
  berlin.print ();
```

Dans cette 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.

A.8.4 Copie, destruction

Un objet peut être passé comme paramètre d'une procédure ou fonction, 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, il faut faire une fonction particulière, noter que l'on peut souvent utiliser la méthode `clone`.

Les objets ne sont jamais copiés implicitement

D'autre part, 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.

A.8.5 Surcharge

Il est possible de définir plusieurs méthodes ayant le même nom (on dit qu'elles sont surchargées), 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 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 milisecondes depuis le début des temps informatiques, ou sans argument pour avoir la date du jour courant. (Remarque: compter le temps avec des milisecondes 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.9 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= ")`	donne	`x =` *newline*
`System.out.println (100)`		`100` *newline*:
`System.out.print (3.14)`		`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:

```
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.

A.9.1 sur le terminal

Pour lire l'entrée sur le 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.

A.9.2 Sur des fichiers

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 shéma 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.10 Exceptions

Les exceptions sont des objets de toute sous-classe de la classe `Exception` (voir la notion de sous-classe plus loin). 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 déja 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;
}
```

peut être écrit dans l'exemple du carré magique ci-dessous.

A.11 Carrés magiques

A.11.1 Présentation

On souhaite présenter un exemple un peu plus important que ceux vus jusqu'ici, on a choisi la construction 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 calcule un carré magique par l'algorithme suivant : 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?

A.11.2 Texte du programme

```
import java.io.*;
import java.lang.*;
import java.util.*;

class CarreMagique {
```

```
final static int N = 100;
static int     a[][] = new int[N][N];
static int     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, j;

  i = n - 1; 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;
p            j = (j + 1) % n;
      }
  }
}

static void erreur (String s) {

  System.err.println ("Erreur fatale: " + s);
  System.exit (1);
}

static int lire ()  {
  int n;
  BufferedReader in =
      new BufferedReader(new InputStreamReader(System.in));

  System.out.print ("Taille du carré magique, svp?::  ");
  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[]) {

    n = lire();
    init(n);        // inutile, mais pédagogique!
    magique(n);
    imprimer(n);
  }
}
```

A.11.3 Explications

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 fraiche 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 coeur) 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 compte 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 retourne de 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-sortie. 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.12 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;
  }
```

```
}
```

Il y a ici deux méthodes, l'une pour convertir un point en chaîne de caractères (pour l'impression) et l'autre (`move`) pour déplacer 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 fonctionalité.

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 définir 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 dex conversion explicites (déja vu 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 nom 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.13 Fonctions graphiques

A.13.1 Fonctions élémentaires

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).

A.13.2 Rectangles

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.

A.13.3 La classe `Maclib`

```
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 219)

```
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`).

A.13.4 Jeu de balle

Le programme suivant 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.14 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.14.1 Contantes littérales

Literal:
    IntegerLiteral
    FloatingPointLiteral
    BooleanLiteral
    CharacterLiteral
    StringLiteral
    NullLiteral

A.14.2 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.14.3 Noms

Name:
    SimpleName
    QualifiedName

SimpleName:
    Identifier

QualifiedName:
    Name . Identifier

A.14.4 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.14.5 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.14.6 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.14.7 Tableaux

ArrayInitializer:
{ VariableInitializers$_{opt}$,$_{opt}$ }

VariableInitializers:
VariableInitializer
VariableInitializers , VariableInitializer

A.14.8 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.14.9 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

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 AT&T 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] ou Monasse [5]. 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 Caml:

```
#open "printf";;

let magique a =
 let n = vect_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 vect_length a - 1 do
   for j = 0 to vect_length a - 1 do
    printf "%4d " a.(i).(j)
   done;
   printf "\n"
  done;;

 let main () =
  let n = lire () in
  let a = 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 = vect_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 `vect_length a` plutôt que `vect_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 `make_matrix`. La ligne

```
let c = 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 Caml Light propose un compilateur indépendant `camlc` 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% camlc 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 `camllight`.

```
poly% camllight
>       Caml Light version 0.71

#let rec fact x = if x <= 1 then 1 else x * fact (x - 1);;
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 fonction `include`. 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:

```
#include "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);;
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;;
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.

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 `make_string` 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 `sub_string` permet d'extraire une sous-chaîne, et la procédure `blit_string` 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 `vect`. 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 |];;
v : int vect = [|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 `make_vect` l c, on obtient un vecteur de longueur l, initialisé avec l'élément c. On dispose aussi des fonctions `sub_vect` pour extraire des sous-chaînes et `blit_vect` 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);;
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;;
x : int ref = ref 1
#let y = ref 1;;
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;;
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";;
La fonction successeur est dorénavant tracée.
- : 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;;
Entrée interactive:
>if true then 1;;
>             ^
Cette expression est de type int,
mais est utilisée avec le 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`
`|` v_1 `->` e_1
`|` v_2 `->` e_2
`...`
`|` v_n `->` e_n

```
| _ -> 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 =` e_1 `to` e_2 `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 =` e_1 `downto` e_2 `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` e_1 `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__Sys_error` est lancée avec pour argument la chaîne de caractères "*fichier*: `No such file or directory`".

```
#open_in "essai";;
Exception non rattrapée: sys__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;;
Exception non rattrapée: 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__Sys_error s ->
      close_in ic;
      printf "Impossible d'ouvrir le ficher %s \n" copie;
      raise (sys__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__Sys_error s ->
    printf "Le ficher %s n'existe pas\n" origine;
    raise (sys__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 $\texttt{p} = 1 + 2x^3$ 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;;
Le type t est défini.
#let int_of_t x =
  match x with C i -> i;;
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;;
Le type t est défini.
#int_of_t (C 2);;
Entrée interactive:
>int_of_t (C 2);;
>          ^^^
Cette expression est de type t,
mais est utilisée avec le 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];;
l : int list = [1; 2; 3]
#let ll = 0 :: l;;
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;;
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;;
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;;
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. *)

value new: unit -> 'a t
        (* Return a new stack, initially empty. *)
  and push: 'a -> 'a t -> unit
        (* [push x s] adds the element [x] at the top of stack [s]. *)
  and pop: 'a t -> 'a
        (* [pop s] removes and returns the topmost element in stack [s],
           or raises [Empty] if the stack is empty. *)
  and clear : 'a t -> unit
        (* Discard all elements from a stack. *)
  and length: 'a t -> int
        (* Return the number of elements in a stack. *)
  and 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 new () = { 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 = do_list f s.c;;
```

La compilation de l'interface du module `stack` produit un fichier `stack.zi` et celle de l'implémentation le fichier `stack.zo`. 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` et `sys__Sys_error` du module `sys`. 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 Caml Light, 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=camlc -W -g -I .

OBJS= module1.zo module2.zo
EXEC= prog

all: $(OBJS)
        $(CAMLC) -o $(EXEC) $(OBJS)

clean:
        rm -f *.z[io] *.zix *~ #*#

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.zi:
        $(CAMLC) -c $<
.ml.zo:
        $(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. La bibliothèque `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, qui s'obtient par la commande `camllight camlgraph`. 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://pauillac.inria.fr/caml/index-fra.html`: site principal de Caml.

`http://pauillac.inria.fr/caml/man-caml/index.html`: manuel en anglais.

`http://pauillac.inria.fr/caml/contribs-fra.html`: bibliothèque et outils.

`http://pauillac.inria.fr/caml/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.

Annexe C

Initiation au système Unix

Le système Unix fut développé à Bell laboratories (*research*) de 1970 à 1980, puis à l'université de Berkeley. C'est un système maintenant standard dans le milieu scientifique. Les machines de l'X en Unix sont des serveurs Dec Alpha ou HP et des stations Dec, HP, PC, MacIntosh pour les élèves, Linux est une variante du système Unix qui fonctionne sur les PC. Cette annexe reprend des notes de cours du magistère de l'ENS écrites par Damien Doligez et Xavier Leroy. Son adaptation pour l'X est due à Dominique Moret. Il y a deux parties: une trousse de survie pour connaître les manipulations simples, un approfondissement pour apprendre les commandes élémentaires. Enfin, le béaba du réseau figure dans une troisième section.

C.1 Trousse de Survie

C.1.1 Se connecter

- Depuis les stations Alpha, Hp, PC, Mac et les terminaux X des salles de TD: taper son nom d'utilisateur *(login)*, puis son mot de passe.

- Depuis une autre machine, interne à l'Ecole: taper `telnet 192.104.247.103` ou simplement `telnet poly` puis répondre à `login/password` comme précédemment.

- Depuis une machine externe à l'Ecole mais reliée au réseau Internet il faut alors taper `telnet 192.104.247.14` ou simplement `telnet sil.polytechnique.fr`[1], puis `login/password` comme précédemment.

- Par Minitel (il faut un Minitel modèle 1B ou 2): se connecter au 01 69 33 35 12. Appuyer sur `fnct-T`, puis `A` pour passer le Minitel en mode 80 colonnes avec défilement. Appuyer plusieurs fois sur la touche retour chariot (↩, pas `Envoi`), jusqu'à obtenir le prompt
  ```
  Entrez dans Communication Server...
  [1 ]CS >
  ```
  Répondre `c poly`, puis retour chariot. (On peut mettre le nom d'une autre machine des salles Dec, Hp ou Pc à la place de `poly`.) Puis login/password comme précédemment.

- Par terminal ou émulation terminal, on peut aussi appeler le 01 69 33 38 38: il faut régler ses paramètres de communication à 8 bits, sans parité, sans Xon/Xoff, 1 bit stop, puis `login/passwd` comme précédemment.

1. `sil` signifie passerelle en breton

C.1.2 Se déconnecter

- Depuis un terminal ou une autre machine: taper `exit`
- Depuis un minitel: taper `exit` puis `logout`.
- Depuis une station ou un terminal X: appuyer sur le bouton droit ou gauche de la souris, le pointeur étant sur le fond d'écran, et sélectionner `DECONNEXION` dans le menu qui apparaît alors. (Taper `exit` dans une fenêtre shell ferme cette fenêtre, mais ne vous déconnecte pas de la machine.)
- Ne jamais éteindre une station ou un terminal X.

C.1.3 Le système de fenêtres par défaut

Ce paragraphe ne concerne que les stations (postes de travail) ou les terminaux X.

Boutons Souris

	Bouton de la Souris		
Position de la souris	Gauche	Milieu	Droite
Titre de fenêtre	déplacer	menu opérations	
Titre (bouton gauche)	menu opérations		
Titre (bouton milieu)	iconifier		
Titre (bouton droite)	maximiser		
Fenêtre Shell (xterm)	sélectionner	coller	modifier la sélection
Ascenseur	descendre	positionner	monter
Icône	déplacer	ouvrir	
Fond d'écran	Sessions xterm poly rafraîchir ecran (re)démarrer Fvwm DECONNEXION		
Fenêtre Emacs	positionner curseur	coller	

Par défaut, la fenêtre active est celle contenant le pointeur de la souris.[2]

Quelques outils du système de fenêtres

`xterm`	ouverture d'une fenêtre shell sur la station où on travaille
`poly`	ouverture d'une fenêtre shell sur poly
`Refresh Screen`	redessine l'écran
`Restart Fvwm`	redémarre le gestionaire de fenêtres

C.1.4 Obtenir de l'aide

`man` *commande* Montre page par page le manuel de *commande*. Faire `Espace` pour passer à la page suivante, `q` pour quitter avant la fin. Pour quitter, on peut aussi faire `Ctrl-C`, qui interrompt la plupart des commandes Unix.

2. Cet environnement est celui qui a été installé par le SITX, mais il est bien sûr entièrement reconfigurable avec un peu d'habileté, ce que nous déconseillons vivement aux débutants.

`man -k` *mot* Donne la liste des commandes indexées sur le mot-clé *mot*, avec un résumé de ce qu'elles font en une ligne.

Dans les programmes interactifs (`elm`, `maple`), on peut souvent obtenir de l'aide en tapant ? ou `h`. Enfin, on peut aussi poser des questions aux utilisateurs habituels des salles informatiques; certains en savent très long.

C.1.5 Changer son mot de passe

La commande est `passwd`. Critères de choix d'un mot de passe:

- utiliser 8 caractères (si vous en proposez plus, seuls les 8 premiers caractères sont pris en compte).
- ni un prénom, ni un nom propre, ni un mot du dictionnaire.
- au moins une lettre majuscule, ou un chiffre, ou un caractère spécial (non alphabétique).
- Pas de caractìre accentué (comme *é*).

C.1.6 Courrier électronique

La commande `elm` permet de lire son courrier et d'en envoyer. Résumé des commandes de `elm`:

Espace	Affiche le message sélectionné (`Espace` pour afficher la page suivante, `i` pour sortir du message)
↑, ↓	Sélectionne un message
`m`	Compose et envoie un message
`r`	Compose et envoie une réponse au message sélectionné
`s`	Sauve le message sélectionné dans un fichier
`d`	Efface le message sélectionné
`q`	Pour sortir

En jargon `elm`, un `folder` est simplement le nom d'un fichier qui contient des messages.

Les adresses de courrier

Une adresse de courrier électronique est de la forme la forme `prénom.nom.@machine`). La `machine` est le nom complet de la machine de destination. Il ne faut pas mettre d'espaces dans une adresse de courrier: `truc @ machine` ne marche pas. Votre adresse de courrier sur les machines de l'X est:

prénom.nom`@polytechnique.fr`

ou

nom-de-login`@poly.polytechnique.fr`

On omet les accents; on met un tiret - pour les prénoms ou noms composés. Exemples (purement imaginaires):

Henri Poincaré	`poincare@poly.polytechnique.fr`
	`Henri.Poincare@polytechnique.fr`
Anne–Sophie de Barreau de Chaise	`barreau@poly.polytechnique.fr`
	`Anne-Sophie.de-Barreau-de-Chaise@polytechnique.fr`
Fulgence l'Hôpital	`lhopital@poly.polytechnique.fr`
	`Fulgence.l-Hopital@polytechnique.fr`

Pour du courrier destiné à un utilisateur de `poly` vous pouvez utiliser une adresse plus simple, réduite au nom d'utilisteur (ou nom de `login`). Pour envoyer du courrier à un utilisateur d'une marchine connectée au réseau de l'X, vous pouvez utiliser l'adresse: *nom-utilisateur*@*nom-de-machine* par exemple : `mail cori@lix`

Si vous n'êtes pas sûr de votre adresse, expérimentez en vous envoyant du courrier à vous-même.

Pour ce qui est des noms complets de machines, ils sont généralement en trois parties: `machine.institution.pays`. Quelques exemples de noms complets:

`poly.polytechnique.fr`	la machine principale de l'X
`clipper.ens.fr`	la machine des "frères" de l'ENS
`ftp.inria.fr`	une machine avec du logiciel à l'INRIA
`research.att.com`	une machine à courrier des AT&T Bell Laboratories
`cs.princeton.edu`	le département d'informatique de Princeton
`src.dec.com`	un labo de Digital Equipment
`jpl.nasa.gov`	le Jet Propulsion Laboratory

Quelques exemples de noms de "pays":

`au`	Australie	`br`	Brésil	`ca`	Canada	`de`	Allemagne
`dk`	Danemark	`fi`	Finlande	`fr`	France	`it`	Italie
`jp`	Japon	`nl`	Hollande	`nz`	Nouvelle-Zélande	`se`	Suède
`su`	CEI	`th`	Thaïlande	`uk`	Royaume-Uni	`za`	Afrique du Sud

`com`	Etats-Unis, réseau des entreprises privés
`edu`	Etats-Unis, réseau des universités
`gov`	Etats-Unis, réseau des labos gouvernementaux
`mil`	Etats-Unis, réseau de l'armée
`bitnet`	Un réseau de machines IBM

C.1.7 Éditeur de texte

Plusieurs éditeurs sont disponibles: `emacs vi textedit ....` Nous recommandons chaudement `emacs`. On le lance par `emacs` ou `emacs` *nom-de-fichier*. Les commandes vitales sont:

$\leftarrow$, $\rightarrow$, $\uparrow$, $\downarrow$	déplace le curseur
`Delete`	efface le caractère précédent
`Control-V`	avance d'une page
`Meta-V`	recule d'une page (`Meta`, ce sont les touches marquées $\diamond$ ou `esc`ou `Alt`)
`Control-X Control-S`	écrit le fichier modifié
`Control-G`	interruption (arrêt) de la commande en cours
`Control-X Control-C`	quitte Emacs

Pour apprendre à se servir d'Emacs, on peut lancer `emacs` et taper `Control-h`, puis `t`; ça affiche un petit cours assez bien fait.

C.1.8 Manipulations simples de fichiers

- `ls` affiche la liste des fichiers.
- `more` *nom* affiche le contenu du fichier *nom*, page par page. La touche espace passe à la page suivante; `q` quitte.
- `cp` *nom1 nom2* copie le fichier *nom1* dans un fichier *nom2*.
- `mv` *nom1 nom2* change le nom d'un fichier de *nom1* en *nom2*.
- `rm` *nom* détruit le fichier *nom*.

C.1.9 Cycles de mise au point

Programmes Java

`emacs prog.java`	On crée le texte source.
`javac prog.java`	On compile le source `prog.java` en l'exécutable `NomClasse.class`
`java NomClasse`	On exécute le programme.
`emacs prog.java`	On corrige le source et on recommence jusqu'à ce que ça marche.

Programmes C

`emacs prog.c`	On crée le texte source.
`cc -o prog prog.c`	On compile le source `prog.c` en l'exécutable `prog`.
`prog`	On exécute le programme.

Textes LaTeX

`emacs memoire.tex`	On crée le texte source.
`latex memoire.tex`	On transforme le source `memoire.tex`, le résultat est dans `memoire.dvi`
`xdvi memoire.dvi`	On regarde à quoi le résultat ressemble.
`emacs memoire.tex`	On corrige le source et on recommence jusqu'à ce qu'ils oit satisfaisant.
⋮	
`dvips -o memoire.ps memoire.dvi`	On transforme `memoire.dvi` en un fichier PostScript.
`lpr -Pqms memoire.ps`	On imprime le résultat sur l'imprimante recto-verso.
`lprm -Pqms` *imprimante user*	Pour stopper l'impression de *user*

C.1.10 Types de machines

Il y a deux types de machines Unix à l'X: des HP et des sations PC Linux. Chacune a un processeur différent: HP-PA pour les premières et Pentium II ou III pour les PC, les fichiers exécutables sont donc différents sur ces deux types de machines. Tout l'environnement par défaut fait que les commandes vont exécuter les exécutables de bon type. Il faut noter que les fichiers des élèves eux se trouvent toujours au même endroit, car Unix permet de partager les fichiers entre machines différentes grâce à NFS (*Network File System*). De même, toutes les imprimantes sont accessibles depuis toute machine. Enfin, il existe aussi des terminaux X qui n'ont pas de processeur (autre que la gestion locale de l'écran) et qui peuvent atteindre toute machine sur le réseau.

C.2 Approfondissement

C.2.1 Système de fichiers

Répertoires

On les appelle aussi *directories*. Un répertoire est une boîte qui peut contenir des fichiers et d'autres répertoires (comme les catalogues de MS-DOS, ou les dossiers du Macintosh). Exemples de répertoires:

```
/users  /bin   /usr/local/bin
```

On désigne les fichiers (et les répertoires) contenus dans un répertoire par: *nom de répertoire*/*nom de fichier*. Exemple: `/bin/sh` est le fichier `sh` contenu dans le répertoire `/bin`. Les répertoires sont organisés en arbre, c'est-à-dire qu'ils sont tous contenus dans un répertoire appelé la *racine*, et désigné par `/`. Chaque répertoire contient deux répertoires spéciaux:

`.`	désigne le répertoire lui-même
`..`	désigne le père du répertoire

Exemples: `/users/cie1/.` est le même répertoire que `/users/cie1`. `/users/cie1/..` est le même répertoire que `/users`.

Chaque utilisateur a un *home-directory*. C'est l'endroit ou il range ses fichiers. Le home-directory a pour nom `/users/cie`*n*`/`*nom*.
Exemples: `/users/cie7/joffre`, `/users/cie5/foch`.

On peut aussi désigner le home-directory d'un autre utilisateur par le nom de login de l'utilisateur précédé d'un tilde (le caractère `~`). Exemple: `~foch`.

Noms de fichiers

Un nom de fichier qui commence par `/` est dit *absolu*. Il est interprété en partant de la racine, et en descendant dans l'arbre. Un nom de fichier qui ne commence pas par `/` est *relatif*. Il est interprété en partant du *répertoire courant*. Le répertoire courant est initialement (au moment où vous vous connectez) votre home-directory.

Exemples: `/users/cie7/joffre/foo` est un nom (ou chemin) absolu. `bar` est un nom relatif. Il désigne un fichier appelé `bar` et situé dans le répertoire courant. Le fichier exact dont il s'agit dépend donc de votre répertoire courant.

Remarque: Le seul caractère spécial dans les noms de fichiers est le slash `/`. Un nom de fichier peut avoir jusqu'à 255 caractères, et contenir un nombre quelconque de points.

Commandes pour manipuler le système de fichiers

- `pwd` affiche le répertoire courant. Exemple:

```
poly% pwd
/users/cie5/foch
```

- `cd` change le répertoire courant. Si on ne lui donne pas d'argument, on retourne dans le home-directory. Exemple:

```
poly% cd ..
poly% pwd
/users/cie5
poly% cd
poly% pwd
/users/cie5/foch
```

- `mkdir` crée un nouveau répertoire, (presque) vide. Il ne contient que `.` et `..`

- `rmdir` supprime un répertoire vide. Si le répertoire contient autre chose que `.` et `..` ça ne marche pas.

- `mv` renomme un fichier, mais peut aussi le déplacer d'un répertoire à un autre. Exemple:

```
poly% cd
poly% mkdir foo
poly% emacs bar
```

```
poly% mv bar foo/bar2
poly% cd foo
poly% pwd
/users/cie5/foch/foo
poly% ls
bar2
```

- `rm -i foo` supprime le fichier `foo`, attention il faut manipuler cette commande avec précaution car on risque de détruire un fichier utile et il est alors impossible de le récupérer par la suite.
- `ls` liste les fichiers et les répertoires qu'on lui donne en arguments, ou le répertoire courant si on ne lui donne pas d'argument. `ls` ne liste pas les fichiers dont le nom commence par `.` C'est pourquoi `.` et `..` n'apparaissent pas ci-dessus.

Les droits d'accès

Chaque fichier a plusieurs propriétés associées: le *propriétaire*, le *groupe propriétaire*, la date de dernière modification, et les *droits d'accès*. On peut examiner ces propriétés grâce à l'option `-lg` de `ls`. Exemple:

```
poly% ls -lg

drw-r--r--  1  foch  cie5  512  Sep 30 17:56  foo
-rw-r--r--  1  foch  cie5    7  Sep 30 17:58  bar
                                              ___ nom du fichier
                                ____________ date de dernière modif.
                           ___ taille
                     ____ groupe propriétaire
               ____ propriétaire
        ___ droits des autres
     ___ droits du groupe
  ___ droits du propriétaire
_ type
```

Type `-` pour les fichiers, `d` pour les répertoires.

Droits du propriétaire

`r` ou `-`	droit de lire le fichier (`r` pour oui, `-` pour non)
`w` ou `-`	droit d'écrire dans le fichier
`x` ou `-`	droit d'exécuter le fichier ou de visite pour un répertoire

Droits du groupe Comme les droits du propriétaire, mais s'applique aux gens qui sont dans le groupe propriétaire.

Droits des autres Comme les droits du propriétaire, mais s'applique aux gens qui sont ni le propriétaire, ni dans le groupe propriétaire.

Propriétaire Le nom de login de la personne à qui appartient ce fichier. Seul le propriétaire peut changer les droits ou le groupe d'un fichier.

Groupe propriétaire Le nom du groupe du fichier. Les groupes sont des ensembles d'utilisateurs qui sont fixés par l'administrateur du système.

Taille En octets.

Pour changer les droits d'un fichier, la commande est `chmod`. Exemples:

`chmod a+x foo`	ajoute (`+`) le droit d'exécution (`x`) pour tout le monde (`a`ll) au fichier `foo`
`chmod g-r bar`	enlève (`-`) le droit de lecture (`r`) pour les gens du groupe (`g`roup) sur le fichier `bar`
`chmod u-w gee`	enlève (`-`) le droit d'écriture (`w`) pour le propriétaire (`u`ser) sur le fichier `gee`

C.2.2 Raccourcis pour les noms de fichiers

Il est ennuyeux d'avoir à taper un nom complet de fichier comme `nabuchodonosor`. Il est encore plus ennuyeux d'avoir à taper une liste de fichier pour les donner en arguments à une commande, comme: `cc -o foo bar.c gee.c buz.c gog.c`. Pour éviter ces problèmes, on peut utiliser des *jokers* (*wildcards* en anglais.)

Une étoile `*` dans un nom de fichier est interprétée par le shell comme "n'importe quelle séquence de caractères qui ne commence pas par un point." Exemple: `cc -o foo *.c`.

Pour interpréter l'étoile, le shell va faire la liste de tous les noms de fichiers du répertoire courant qui ne commencent pas par `.` et qui finissent par `.c` Ensuite, il remplace `*.c` par cette liste (triée par ordre alphabétique) dans la ligne de commande, et exécute le résultat, c'est-à-dire par exemple: `cc -o foo bar.c buz.c foo.c gee.c gog.c`.

On a aussi le `?`, qui remplace un (et exactement un) caractère quelconque. Par exemple, `ls ?*` liste tous les fichiers, y compris ceux dont le nom commence par un point.

La forme `[abcd]` remplace un caractère quelconque parmi `a`, `b`, `c`, `d`. Enfin, `[^abcd]` remplace un caractère quelconque qui ne se trouve pas parmi `a`, `b`, `c`, `d`.

Exemple: `echo /users/*` affiche la même chose que `ls /users`. (La commande `echo` se contente d'afficher ses arguments.)

Attention:

- C'est le shell qui fait le remplacement des arguments contenant un joker. On ne peut donc pas faire `mv *.c *.bak`, car le shell va passer à `mv` les arguments `foo.c bar.c foo.bak bar.bak`, et `mv` ne sait pas quel fichier remplacer.
- Attention aux espaces. Si vous tapez `rm * ~`, le shell remplace l'étoile par la liste des fichiers présents, et ils seront tous effacés. Si vous tapez `rm *~`, seuls les fichiers dont le nom finit par un tilde seront effacés.

Interlude: comment effacer un fichier nommé `?*`? On ne peut pas taper `rm ?*` car le shell remplace `?*` par la liste de tous les fichiers du répertoire courant. On peut taper `rm -i *` qui supprime tous les fichiers, mais en demandant confirmation à chaque fichier. On répond `no` à toutes les questions sauf `rm: remove ?*?`. Autre méthode: utiliser les mécanismes de quotation (voir ci-dessous).

C.2.3 Variables

Le shell a des variables. Pour désigner le contenu d'une variable, on écrit le nom de la variable précédé d'un dollar. Exemple: `echo $HOME` affiche le nom du home-directory de l'utilisateur.

On peut donner une valeur a une variable avec la commande `setenv`:

```
poly% setenv foo bar
```

```
poly% echo $foo
bar
```

Les valeurs des variables sont accessibles aux commandes lancées par le shell. L'ensemble de ces valeurs constitue l'*environnement*. On peut aussi supprimer une variable de l'environnement avec `unsetenv`.

Quelques variables d'environnement:

`PRINTER` Pour les commandes d'impression. Contient le nom de l'imprimante sur laquelle il faut envoyer vos fichiers.

`EDITOR` Utilisée par `elm`, `polyaf`, et beaucoup d'autres commandes. Contient le nom de votre éditeur de textes préféré.

`VISUAL` La même chose qu'`EDITOR`.

`SHELL` Contient le nom de votre shell préféré.

`HOME` Contient le nom de votre home-directory.

`USER` Contient votre nom de login.

`LOGNAME` La même chose que `USER`.

`PATH` Contient une liste de répertoires dans lesquels le shell va chercher les commandes exécutables.

`DISPLAY` Contient le nom de la machine qui affiche.

C.2.4 Le chemin d'accès aux commandes

La variable `PATH` contient le chemin d'accès aux commandes. Le shell l'utilise pour trouver les commandes. Il s'agit d'une liste de répertoires séparés par des `:`. La plupart des commandes sont en fait des programmes, c'est-à-dire des fichiers qu'on trouve dans le système de fichiers. Quand vous tapez `ls`, par exemple, le shell exécute le fichier `/bin/ls`. Pour trouver ce fichier, il cherche dans le premier répertoire du `PATH` un fichier qui s'appelle `ls`. S'il ne trouve pas, il cherche ensuite dans le deuxième répertoire et ainsi de suite. S'il ne trouve la commande dans aucun répertoire du `PATH`, le shell affiche un message d'erreur. Exemple:

```
poly% sl
sl: Command not found.
```

Exercice: Assurez-vous que `/usr/games` se trouve bien dans votre `PATH`.

C.2.5 Quotation

Avec tous ces caractères spéciaux, comment faire pour passer des arguments bizarres à une commande? Par exemple, comment faire afficher un point d'interrogation suivi d'une étoile et d'un dollar par `echo`? Le shell fournit des mécanismes pour ce faire. Ce sont les *quotations*. Le plus simple est le backslash `\`. Il suffit de précéder un caractère spécial d'un backslash, et le shell remplace ces deux caractères par le caractère spécial seul. Evidemment, le backslash est lui-même un caractère spécial.

Attention, tout ceci est très dépendant de la machine et les exemples ci-dessous ne sont pas garantis.

Exemples:

```
poly% echo \*\$
```

```
*$
poly% echo \\\\\*\\\$
\\*\$
```

Un autre moyen est d'inclure une chaîne de caractères entre apostrophes (simple quotes) `'`. Tout ce qui se trouve entre deux apostrophes sera passé tel quel par le shell à la commande. Exemple:

```
poly% echo '$?*\'
$?*\
```

Enfin, on peut utiliser des guillemets (double quotes) `"`. Les guillemets se comportent comme les apostrophes, à une exception près: les dollars et les backslashes sont interprétés entre les guillemets. Exemple:

```
poly% echo "$HOME/*"
/users/cie5/foch/*
```

Une technique utile: Quand on juxtapose deux chaînes de caractères quotées, le shell les concatène, et elles ne forment qu'un argument. Exemple:

```
poly% echo "'"'"'
'"
```

Quant aux interactions plus compliquées (backslashes à l'intérieur des guillemets, guillemets à l'intérieur des apostrophes, etc.), le meilleur moyen de savoir si ça donne bien le résultat attendu est d'essayer. La commande `echo` est bien utile dans ce cas.

Dernière forme de quotation: `` ` ``*commande*`` ` ``. Le shell exécute la *commande*, lit la sortie de la commande mot par mot, et remplace `` ` ``*commande*`` ` `` par la liste de ces mots. Exemple:

```
poly% echo `ls`
Mail News bin foo g7 lib misc marne.aux marne.dvi marne.log marne.tex
poly% ls -lg `which emacs`
-rwxr-xr-x  1 root     system     765952 Dec 17  1992 /usr/local/bin/emacs
```

La commande `which` *cmd* employée ci-dessus affiche sur sa sortie le nom absolu du fichier exécuté par le shell quand on lance la commande *cmd*.

```
poly% which emacs
/usr/local/bin/emacs
```

C.2.6 Redirections et filtres

Chaque commande a une *entrée standard*, une *sortie standard*, et une *sortie d'erreur*. Par défaut, l'entrée standard est le clavier, la sortie standard est l'écran, et la sortie d'erreur est aussi l'écran.

On peut *rediriger* la sortie standard d'une commande vers un fichier (caractère `>`). Le résultat de la commande sera placé dans le fichier au lieu de s'afficher sur l'écran. Exemple:

```
poly% ls -l >foo
```

Le résultat de `ls -l` ne s'affiche pas à l'écran, mais il est placé dans le fichier `foo`. On peut alors taper

```
poly% more foo
```

pour lire le fichier page par page.

On peut aussi rediriger l'entrée standard d'une commande (caractère `<`). La commande lira alors le fichier au lieu du clavier. Exemple:

```
poly% elm joffre <foo
```

envoie par mail à Joseph Joffre le résultat de la commande `ls -l` de tout à l'heure.

On peut aussi taper `more <foo` qui est équivalent à `more foo` car `more` sans argument lit son entrée standard et l'affiche page par page sur le terminal.

On peut aussi se passer du fichier intermédiaire grâce à un *pipe* (caractère `|`). Un pipe connecte directement la sortie standard d'une commande sur l'entrée standard d'une autre commande. Exemple: pour afficher page par page la liste des fichiers du répertoire courant, faire

```
ls -l | more
```

La panoplie complète des redirections est la suivante:

`>` change la sortie standard de la commande pour la placer dans un fichier.

`<` change l'entrée standard de la commande pour la prendre dans un fichier.

`>&` place la sortie standard et la sortie erreur dans un fichier.

`|` branche la sortie standard de la commande de gauche sur l'entrée standard de la commande de droite.

`|&` branche la sortie standard et la sortie erreur de la commande de gauche sur l'entrée standard de la commande de droite.

>> change la sortie standard pour l'ajouter à la fin d'un fichier existant.

>>& place la sortie standard et la sortie erreur à la fin d'un fichier existant.

Remarques: Normalement, une redirection avec `>` sur un fichier qui existe déjà efface le contenu du fichier avant d'y placer le résultat de la commande. Il existe une option qui dit au shell `tcsh` de refuser d'effacer le fichier.

Le pipe avec `|&` est utile pour capturer tout ce qui sort d'une commande. Exemple: `ls -R / |& more` affiche page par page la liste de tous les fichiers du système, sans que les messages d'erreur dérangent l'affichage.

Une ligne de commandes contenant des `|` s'appelle un pipe-line. Quelques commandes souvent utilisées dans les pipe-lines sont:

more à la fin du pipe-line, affiche le résultat page par page, pour laisser le temps de le lire.

wc compte le nombre de caractères, de mots et de lignes de son entrée.

grep cherche dans son entrée les lignes contenant un mot donné, et les écrit sur sa sortie.

sort lit toutes les lignes de son entrée, les trie, et les écrit dans l'ordre sur sa sortie

tail écrit sur sa sortie les dernières lignes de son entrée.

head écrit sur sa sortie les premières lignes de son entrée.

cat copie plusieurs fichiers sur sa sortie.

fold coupe les lignes de son entrée à 80 caractères et écrit le résultat sur sa sortie.

Exemples:

```
poly% cat glop buz >toto
```

Concatène les fichiers `glop` et `buz` et place le résultat dans `toto`.

```
poly% wc -w /usr/dict/words
```

Affiche le nombre de mots du dictionnaire Unix.

```
poly% grep gag /usr/dict/words | tail
```

Affiche les 20 derniers mots du dictionnaire qui contiennent la chaîne `gag`.

C.2.7 Processus

Si on lance une commande qui prend beaucoup de temps, on peut l'interrompre par `Control-C`. Ceci interrompt (définitivement) la commande. On peut aussi exécuter une commande en *tâche de fond*. Le shell rend alors la main avant la fin de la commande. Pour le faire, on ajoute un `&` à la fin de la commande:

```
poly% cc -o grosprogramme grosfichier.c &
```

Cette commande lance le compilateur `cc` en parallèle avec le shell. On reprend la main immédiatement, sans attendre la fin de l'exécution de la commande. On peut donc taper d'autres commandes pendant que la précédente d'exécute. La commande `ps` ou `ps -x` montre où en sont les tâches de fond:

```
poly% ps
  PID TT STAT  TIME COMMAND
 4450 p9 S     0:00 /usr/local/bin/tcsh
 4782 p9 S     0:02 cc -o grosprogramme grosfichier.c
 4841 p9 R     0:00 ps
```

Unix est un système *multi-tâches*, c'est-à-dire qu'il peut exécuter plusieurs programmes à la fois. Un processus est un programme en train de s'exécuter. La commande `ps` affiche la liste des processus que vous avez lancés. Chaque processus a un numéro. C'est la colonne PID ci-dessus. Le shell crée un nouveau processus pour exécuter chaque commande. Pour une commande "normale" (sans `&`), il attend que le processus termine, indiquant que la commande a fini de s'exécuter. Pour une commande en tâche de fond (avec `&`), le shell n'attend pas. On peut interrompre ("tuer") un processus avant la fin, avec la commande `kill -9` (plus le numéro du processus).

```
poly% kill -9 4782
poly% ps
  PID TT STAT  TIME COMMAND
 4450 p9 S     0:00 /usr/local/bin/tcsh
 4851 p9 R     0:00 ps
```

C.2.8 Programmation du shell

Le shell peut aussi exécuter des commandes prises dans un fichier. Un fichier contenant des commandes pour le shell est appelé un *script*. C'est en fait un programme écrit dans le langage du shell. Ce langage comprend non seulement les commandes que nous avons déjà vues, mais aussi des structures de contrôle (constructions conditionnelles et boucles).[3] Pour être un script, un fichier doit commencer par la ligne:

3. Il existe en fait plusieurs shells, ayant des langages de commandes différents. Jusqu'ici, on a pris comme exemple le shell `csh` et sa variante `tcsh`. Pour la programmation du shell, nous allons utiliser le shell `sh`, qui a un meilleur langage de commandes. Ce que nous avons vu jusqu'ici s'applique aussi bien à `sh` qu'à `csh`, à l'exception de `setenv` et de certaines redirections.

```
#!/bin/sh
```

Il doit aussi avoir le droit d'exécution (bit `x`). (Le `#!/bin/sh` sur la première ligne indique que ce script doit être exécuté par le shell `sh`.)

Structures de contrôle

- `for` *var* `in` *liste de chaînes* `;` `do` *commandes* `;` `done`
  Affecte successivement à la variable de nom *var* chaque chaîne de caractères dans la *liste de chaînes*, et exécute les *commandes* une fois pour chaque chaîne. Rappel: `$`*var* accède à la valeur courante de *var*. La partie *commandes* est une suite de commandes, séparées par des `;` ou des retours à la ligne. (Tous les `;` dans cette syntaxe peuvent aussi être remplacés par des retour à la ligne.)

  Exemple: `for i in *; do echo $i; done` affiche tous les fichiers du répertoire courant, un par ligne.

- `if` *commande* `;` `then` *commandes* `;` `else` *commandes* `;` `fi`
  Exécute l'une ou l'autre des listes de *commandes*, suivant que la première *commande* a réussi ou non (voir ci-dessous).

- `while` *commande* `;` `do` *commande* `;` `done`
  Exécute les *commandes* de manière répétée tant que la première *commande* réussit.

- `case` *chaîne* `in`
  pattern `)` *commande* `;;`
  ...
  pattern `)` *commande* `;;`
  `esac`
  Exécute la première *commande* telle que la *chaîne* est de la forme *pattern*. Un *pattern* est un mot contenant éventuellement les constructions `*`, `?`, `[a-d]`, avec la même signification que pour les raccourcis dans les noms de fichiers. Exemple:

```
case $var in
  [0-9]* ) echo 'Nombre';;
  [a-zA-Z]* ) echo 'Mot';;
  * ) echo 'Autre chose';;
esac
```

Code de retour

On remarque que la condition des commandes `if` et `while` est une commande. Chaque commande renvoie un code de retour (qui est ignoré en utilisation normale). Si le code est 0, la commande a réussi; sinon, la commande a échoué. Par exemple, le compilateur `cc` renvoie un code d'erreur non nul si le fichier compilé contient des erreurs, ou s'il n'existe pas.

Les commandes `if` et `while` considèrent donc le code de retour 0 comme "vrai", et tout autre code comme "faux".

Il existe une commande `test`, qui évalue des expressions booléennes passées en argument, et renvoie un code de retour en fonction du résultat. Elle est bien utile pour les scripts. Exemple:

```
if test $var = foo
then echo 'La variable vaut foo'
else echo 'La variable ne vaut pas foo'
fi
```

Variables

Dans les scripts, on peut utiliser des variables définies à l'extérieur (avec `setenv`), mais aussi définir ses propres variables locales au script. On donne une valeur à une variable avec une commande de la forme *nom-de-variable* = *valeur*.

On a aussi des variables spéciales, initialisées automatiquement au début du script:

`$*`	La liste de tous les arguments passés au script.
`$#`	Le nombre d'arguments passés au script.
`$1`, `$2`, ...	Les arguments passés au script.
`$?`	Le code de retour de la dernière commande lancée.
`$!`	Le numéro de process de la dernière commande en tâche de fond.
`$$`	Le numéro de process du shell lui-même.

Commandes internes

Certaines commandes du shell ne sont pas des programmes mais des commandes *internes*. Elles sont directement reconnues et exécutées par le shell. Un exemple de commande interne est `cd`. C'est le répertoire courant du shell qui est modifié par `cd`, ce qui signifie que le script suivant:

```
#! /bin/sh
cd $*
```

ne marche pas, car le shell lance un autre shell pour exécuter le script. C'est ce sous-shell qui change son répertoire courant, et ce changement est perdu quand le sous-shell meurt.

Fichier de démarrage

Il existe un script spécial, qui est exécuté au moment où on se connecte. Ce script est contenu dans le fichier `$HOME/.login`. C'est ce fichier qui vous dit s'il y a de nouveaux messages dans polyaf, si vous avez du courrier, etc .... Chacun peut ainsi personnaliser son environnement au début de chaque session. On a quelques informations sur la "customization" en utilisant le menu Aide (bouton de droite de la souris sur fond d'écran).

C.3 Unix et le réseau de l'X

Le réseau Internet relie près de 100 millions d'utilisateurs dans le monde en juillet 99, ce nombre est en croissance très rapide il a été multiplié par 5 en 4 ans. Avec le système Unix, les machines s'interfacent facilement à l'Internet, certains services sont aussi disponibles sur Macintosh ou PC. Le réseau local de l'X contient plusieurs sous-réseaux pour les élèves, pour les laboratoiress et pour l'administration. Le réseau des élèves relie les chambres, les salles de TD (salles Dec, Sun ou Hp), la salle DEA, la machine sil (passerelle vers l'Internet).

Physiquement, dans une chambre, on connecte sa machine par un petit cable muni d'une prise RJ45. Une ligne en paires torsadées 10Base/T part de la chambre vers une boîte d'interconnexion avec une fibre optique FDDI (*Fiber Data Distributed Interface*) qui arrive dans chaque casert. La fibre repart des caserts vers les salles de TD et les autres machines centrales. Le tout est certifié à 10 Méga bit/s, et fonctionne vraisemblablement à 100 Mbit/s. Dans une salle de TD, les stations Unix et les imprimantes sont connectées au réseau. Pour les Mac ou PC, une prise différente existe devant chaque chaise.

Logiquement, la partie 10Base/T supporte le protocole Ethernet à 10 Mbit/s (pour les PC) ou le protocole PhoneNet à 230 kbit/s (pour les Mac). Tout est transparent pour l'utilisateur, qui signale en début d'année au Centre Info s'il désire une prise PhoneNet ou Ethernet dans sa chambre. L'interconnexion avec la fibre optique sera alors positionné correctement.

Toute machine a une adresse Internet, de chiffres (`192.104.247.103` pour `poly`) ou symbolique (`poly.polytechnique.fr`) qui sont les mêmes adresses que pour le courrier électronique. A l'intérieur de l'X, le suffixe `polytechnique.fr` est inutile. Les PC de la salle 35 ont des noms d'os (`radius`, `cubitus`, ...), les stations Hp de la salle 34 des noms de poissons (`carpe`, `lieu`, ...), celles de la salle 36 des noms de voitures (`ferrari`, `bugatti`, ...) Les PC de la salle 33 ont des noms de département (`loire`, `marne`, ...). Les Mac obtiennent leur adresse Internet dynamiquement de la boite d'interconnexion à Ethernet. C'est en principe automatique. Il peut être nécessaire de se servir de l'utilitaire MacTCP pour donner la zone à laquelle on appartient, par exemple son nom de casert + le numéro d'étage. Pour les PC, il faut rentrer l'adresse Internet manuellement. Dans les salles TD, elle est écrite en face de chaque chaise.

Voici une liste de services courants (cf. la référence [12] pour beaucoup plus de détails).

telnet Pour se connecter sur une autre machine et y exécuter des commandes.

rlogin Idem.

ftp Pour transférer des fichiers depuis une autre machine. Sur certaines machines, on peut faire des transferts sans y avoir un compte, en se connectant sous le nom `anonymous`; d'où le nom de "anonymous FTP", ou "FTP anonyme", donné à cette méthode de transfert de fichiers.

xrn Pour lire les "News" (le forum à l'échelle mondiale).

xwais Pour interroger des bases de données par mots-clé.

xarchie Pour obtenir les sources domaine public sur le réseau.

netscape Pour consulter les sites multi-media du *World Wide Web*. Quelques adresses qui peuvent vous être utiles

`http://www.polytechnique.fr/`	le serveur officiel de l'X,
`http://www.polytechnique.fr/eleves/`	le serveur des élèves de l'X,
`http://www.polytechnique.fr/bibliotheque.html/`	la bibliothèque de l'X,
`http://www.ens.fr/`	le serveur de l'ENS,
`http://www.stanford.edu/`	*Stanford University*,
`http://www.labri.u-bordeaux.fr/`	le Labri de Bordeaux,
`http://www.meteo.fr/`	Météo France,
`http://www.sncf.fr/`	la SNCF,
`http://www.delta-air.com/`	*Delta Air Lines*,
`http://www.airfrance.fr/`	*Air France*,
`http://pauillac.inria.fr/caml/`	les langages Caml,
`http://www.research.digital.com/SRC/modula-3/html/`	Modula-3,
`http://java.sun.com/`	et Java,
`http://inferno.bell-labs.com/inferno/`	système Inferno,
`http://www.lmet.fr/Homefr.html`	la librairie Le Monde en Tique,
`http://www.technical.powells.portland.or.us/`	*Powell's Technical Books*,
`http://www.radio-france.fr/`	Radio France,
`http://www.sdv.fr/orthonet/recherche.html`	Dictionnaire de langue française,
`http://www.msnbc.com/`	MSNBC,

`http://www-groups.dcs.st-and.ac.uk/%7Ehistory/Mathematicians/Godel.html`	une biographie de Kurt Gödel,
`http://www-cs-faculty.stanford.edu/~knuth/`	la page de D. Knuth,
`http://www.netfrance.com/Libe/`	le journal Libération,
`http://www.mapquest.com/`	un atlas interactif des *USA*,
`http://www.apple.com/`	*Apple computers*,
`http://www.cern.ch/`	le CERN à Genève,
`http://www.culture.fr/louvre/`	le Louvre
`http://www.bordeaux-wines.com/CIVB/Francais/Encyclopedie/`	Les vins de Bordeaux
`http://www.yahoo.com/`	un logiciel de recherche de sites
`http://webcrawler.com/`	une belle araignée chercheuse,
`http://altavista.digital.com/`	l'index automatique de l'Internet

irc Pour perdre son énergie à discuter avec le monde entier.

eudora Pour lire son courrier sur Mac.

fetch Pour faire `ftp` depuis les Mac.

Certains services (connexions internes, courrier, *news*) sont disponibles depuis toute machine du réseau des élèves. Les autres (et en particulier les connexions à l'Internet) ne sont disponibles que depuis la machine sil. Il convient donc d'ouvrir une session sur sil pour accéder à tous les services de l'Internet.

C.4 Bibliographie Unix

– **Utiliser Unix**

[1] Documentation Ultrix. Une dizaine de volumes, deux exemplaires, en salle Sun ou Dec. Contient des manuels de référence très techniques, mais aussi de bon guides pour débutants: “Getting started with Unix”, “Doing more with Unix”, “SunView 1 beginner's guide”, “Mail and messages: beginner's guide”, “Basic troubleshooting”.

[2] Steve Bourne, “The Unix system”, Addison-Wesley. Traduction française: “Le système Unix”, Interéditions. Une vue d'ensemble du système Unix. Les chapitres sur l'édition et le traitement de texte sont dépassés.

– **Programmer sous Unix**

[3] Brian Kernighan, Rob Pike, “The Unix programming environment”, Addison-Wesley. Traduction française: “L'environnement de la programmation Unix”, Interéditions. Une autre vue d'ensemble du système Unix, plus orientée vers la programmation en C et en shell.

[4] Jean-Marie Rifflet, “La programmation sous Unix”. Programmation en C et en shell (**pas mal**).

– **Traitement de texte**

[5] Leslie Lamport, “LaTeX user's guide and reference manual”. Addison-Wesley, 1986. Tout sur le traitement de texte LaTeX. Un exemplaire se trouve dans la salle Sun.

[6] Donald E. Knuth. “The TeXbook”. Addison-Wesley, 1984. Tout, absolument tout sur le traitement de texte TeX. LaTeX est en fait une extension de TeX, un peu plus simple à utiliser; mais beaucoup de commandes sont communes aux deux, et non documentés dans [5]. Un exemplaire se trouve dans la salle Sun.

[7] Raymond Seroul. “Le petit livre de TeX”. Interéditions. Petit sous-ensemble de [6]; plus accessible, cependant.

[8] Francis Borceux. “LaTeX, la perfection dans le traitement de texte”. (Hum?) Editions Ciaco. Resucée de [5]. Moins complet et guère plus accessible.

– **Le langage C**

[9] Brian Kernighan, Dennis Ritchie, “The C programming language”, Prentice-Hall. Traduction française: “Le langage C”, Interéditions. Le livre de référence sur le langage C.

[10] Samuel Harbison, Guy Steele. “C: a reference manual”. Une autre bonne description du langage C.

– **Utiliser le réseau**

[11] Brendan P. Kehoe, “Zen and the art of the Internet — A beginner’s guide to the Internet”. (Non publié.) Un exemplaire se trouve en salle Sun.

[12] Ed Krol, “The whole Internet user’s guide & catalog”, O’Reilly & Associates, 1992.

Bibliographie

[1] Harold Abelson, Gerald J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[2] Adobe Systems Inc., *PostScript Language, Tutorial and Cookbook*, Addison Wesley, 1985.

[3] 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.

[4] Henk Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, North Holland, 1981.

[5] Danièle Beauquier, Jean Berstel, Philippe Chrétienne, *Eléments d'algorithmique*, Masson, Paris, 1992.

[6] Jon Bentley, *Programming Pearls*, Addison Wesley, 1986.

[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] 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

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Algorithms*, MIT Press, 1990.

[11] Patrick Cousot, *Introduction à l'algorithmique et à la programmation*, Ecole Polytechnique, Cours d'Informatique, 1986.

[12] Shimon Even, *Graph Algorithms*, Computer Science Press, Potomac, Md, 1979.

[13] Adele Goldberg, *Smalltalk-80: the language and its implementation*, Addison-Wesley 1983.

[14] David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmeticc*, Computing Surveys, 23(1), Mars 1991.

[15] Gaston H. Gonnet, Riccardo Baeza-Yates, *Handbook of Algorithms and Data Structures, In Pascal and C*, Addison Wesley, 1991.

[16] 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.

[17] Ron L. Graham, Donald E. Knuth, Oren Patashnik, *Concrete mathematics: a foundation for computer science*, Addison Wesley, 1989.

[18] Samuel P. Harbison, *Modula-3*, Prentice Hall, 1992.

[19] John H. Hennessy, David A. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishers, Inc. , 1990.

[20] Kathleen Jensen, Niklaus Wirth, *PASCAL user manual and report : ISO PASCAL standard*, Springer, 1991. (1ère édition en 1974).

[21] Gerry Kane, *Mips, RISC Architecture*, MIPS Computer Systems, Inc., Prentice Hall, 1987.

[22] 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.

[23] Brian W. Kernighan, *PIC–a language for typesetting graphics*, Software Practice & Experience 12 (1982), 1-20.

[24] Dick B. Kieburtz, *Structured Programming And Problem Solving With Algol W*, Prentice Hall, 1975.

[25] Stephen C. Kleene, *Introduction to Metamathematics*, North Holland, 6ème édition, 1971. (1ère en 1952).

[26] Donald E. Knuth, *The TEXbook*, Addison Wesley, 1984.

[27] Donald E. Knuth, *The Metafont book*, Addison Wesley, 1986.

[28] Donald E. Knuth, *Fundamental Algorithms. The Art of Computer Programming*, vol 1, Addison Wesley, 1968.

[29] Donald E. Knuth, *Seminumerical algorithms, The Art of Computer Programming*, vol 2, Addison Wesley, 1969.

[30] Donald E. Knuth, *Sorting and Searching. The Art of Computer Programming*, vol 3, Addison Wesley, 1973.

[31] Leslie Lamport, *LATEX, User's guide & Reference Manual*, Addison Wesley, 1986.

[32] 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.

[33] Jan van Leeuwen, *Handbook of theoretical computer science*, volumes A et B, MIT press, 1990.

[34] M. Lothaire, *Combinatorics on Words*, Encyclopedia of Mathematics, Cambridge University Press, 1983.

[35] Udi Manber, *Introduction to Algorithms, A creative approach*, Addison Wesley, 1989

[36] Bob Metcalfe, D. Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks*, Communications of the ACM 19,7, Juillet 1976, pp 395–404.

[37] Robin Milner, *A proposal for Standard ML*, In ACM Symposium on LISP and Functional Programming, pp 184-197, 1984, ACM Press, New York.

[38] Robin Milner, Mads Tofte, Robert Harper, *The definiton of Standard ML*, The MIT Press, 1990.

[39] Greg Nelson, *Systems Programming with Modula-3*, Prentice Hall, 1991.

[40] Eric Raymond, *The New Hacker's Dictionary*, dessins de Guy L. Steele Jr., MIT Press 1991.

[41] Brian Randell, L. J. Russel, *Algol 60 Implementation*, Academic Press, New York, 1964.

[42] Martin Richards, *The portability of the BCPL compiler*, Software Practice and Experience 1:2, pp. 135-146, 1971.

[43] Martin Richards, Colin Whitby-Strevens, *BCPL : The Language and its Compiler*, Cambride University Press, 1979.

[44] 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).

[45] Hartley Rogers, *Theory of recursive functions and effective computability*, MIT press, 1987, (édition originale McGraw-Hill, 1967).

[46] A. Sainte-Laguë, *Les réseaux (ou graphes)*, Mémoire des Sciences Mathématiques (18), 1926.

[47] Bob Sedgewick, *Algorithms*, 2nd edition, Addison-Wesley, 1988. En français: *Algorithmes en langage C*, trad. par Jean-Michel Moreau, InterEditions, 1991.

[48] Ravi Sethi, *Programming Languages, Concepts and Constructs*, Addison Wesley, 1989.

[49] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, 1986.

[50] Robert E. Tarjan, *Depth First Search and linear graph algorithms*, Siam Journal of Computing, 1, pages 146-160, 1972.

[51] 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).

[52] Pierre Weis, *The Caml Reference Manual, version 2-6.1*, Rapport technique 121, INRIA, Rocquencourt, 1990.

[53] Pierre Weis, Xavier Leroy, *Le langage Caml*, InterEditions, 1993.

Table des figures

Index