Algorithmes et Programmation

Ecole Polytechnique

Robert Cori Jean-Jacques Lévy

Table des matières

Introduction **5**

Complexité des algorithmes **12**

Scalaires **13**
Les entiers 15
Les nombres flottants 15

1 Tableaux **19**
1.1 Le tri 19
1.1.1 Méthodes de tri élémentaires 20
1.1.2 Analyse en moyenne 24
1.1.3 Le tri par insertion 24
1.2 Recherche en table 27
1.2.1 La recherche séquentielle 28
1.2.2 La recherche dichotomique 29
1.2.3 Insertion dans une table 30
1.2.4 Hachage 31
1.3 Programmes en C 37

2 Récursivité **43**
2.1 Fonctions récursives 43
2.1.1 Fonctions numériques 43
2.1.2 La fonction d'Ackermann 46
2.1.3 Récursion imbriquée 46
2.2 Indécidabilité de la terminaison 47
2.3 Procédures récursives 48
2.4 Fractales 50
2.5 Quicksort 53
2.6 Le tri par fusion 56
2.7 Programmes en C 58

3 Structures de données élémentaires **63**
3.1 Listes chaînées 64
3.2 Piles 70
3.3 Evaluation des expressions arithmétiques préfixées 71
3.4 Files 74
3.5 Opérations courantes sur les listes 78
3.6 Programmes en C 81

4 Arbres **89**
4.1 Files de priorité . . . 90
4.2 Borne inférieure sur le tri . . . 94
4.3 Implémentation d'un arbre . . . 95
4.4 Arbres de recherche . . . 98
4.5 Arbres équilibrés . . . 100
4.6 Programmes en C . . . 103

5 Graphes **111**
5.1 Définitions . . . 111
5.2 Matrices d'adjacence . . . 113
5.3 Fermeture transitive . . . 115
5.4 Listes de successeurs . . . 118
5.5 Arborescences . . . 120
5.6 Arborescence des plus courts chemins. . . . 124
5.7 Arborescence de Trémaux . . . 126
5.8 Composantes fortement connexes . . . 130
5.8.1 Définitions et algorithme simple . . . 130
5.8.2 Utilisation de l'arborescence de Trémaux . . . 131
5.8.3 Points d'attache . . . 133
5.9 Programmes en C . . . 138

6 Analyse Syntaxique **145**
6.1 Définitions et notations . . . 146
6.1.1 Mots . . . 146
6.1.2 Grammaires . . . 147
6.2 Exemples de Grammaires . . . 148
6.2.1 Les systèmes de parenthèses . . . 148
6.2.2 Les expressions arithmétiques préfixées . . . 149
6.2.3 Les expressions arithmétiques . . . 149
6.2.4 Grammaires sous forme BNF . . . 150
6.3 Arbres de dérivation et arbres de syntaxe abstraite . . . 152
6.4 Analyse descendante récursive . . . 153
6.5 Analyse LL . . . 158
6.6 Analyse ascendante . . . 161
6.7 Evaluation . . . 162
6.8 Programmes en C . . . 163

7 Modularité **167**
7.1 Un exemple: les files de caractères . . . 167
7.2 Interfaces et modules . . . 170
7.3 Interfaces et modules en Pascal . . . 172
7.4 Compilation séparée et librairies . . . 172
7.5 Dépendances entre modules . . . 175
7.6 Tri topologique . . . 177
7.7 Programmes en C . . . 178

8 Exploration **181**
8.1 Algorithme glouton . . . 181
8.1.1 Affectation d'une ressource . . . 182
8.1.2 Arbre recouvrant de poids minimal . . . 183
8.2 Exploration arborescente . . . 185
8.2.1 Sac à dos . . . 185
8.2.2 Placement de reines sur un échiquier . . . 186
8.3 Programmation dynamique . . . 188
8.3.1 Plus courts chemins dans un graphe . . . 188
8.3.2 Sous-séquences communes . . . 191

A Pascal **195**
A.1 Un exemple simple . . . 195
A.2 Quelques éléments de Pascal . . . 199
A.2.1 Symboles, séparateurs, identificateurs . . . 199
A.2.2 Types de base . . . 199
A.2.3 Types scalaires . . . 200
A.2.4 Expressions . . . 201
A.2.5 Types tableaux . . . 202
A.2.6 Procédures et fonctions . . . 202
A.2.7 Blocs et portée des variables . . . 204
A.2.8 Types déclarés . . . 204
A.2.9 Instructions . . . 205
A.2.10 Chaînes de caractères . . . 207
A.2.11 Ensembles . . . 208
A.2.12 Arguments fonctionnels . . . 209
A.2.13 Entrées – Sorties . . . 210
A.2.14 Enregistrements . . . 213
A.2.15 Pointeurs . . . 215
A.2.16 Fonctions graphiques . . . 216
A.3 Syntaxe BNF de Pascal . . . 220
A.4 Diagrammes de la syntaxe de Pascal . . . 224

B Le langage C **233**
B.1 Un exemple simple . . . 233
B.2 Quelques éléments de C . . . 236
B.2.1 Symboles, séparateurs, identificateurs . . . 236
B.2.2 Types de base . . . 236
B.2.3 Types scalaires . . . 237
B.2.4 Expressions . . . 238
B.2.5 Instructions . . . 243
B.2.6 Procédures, fonctions, structure d'un programme . . . 245
B.2.7 Pointeurs et tableaux . . . 247
B.2.8 Structures . . . 250
B.2.9 Entrées-Sorties . . . 251
B.2.10 Fonctions graphiques . . . 253
B.3 Syntaxe BNF de C . . . 255
B.4 Diagrammes de la syntaxe de C . . . 261

C Caml **271**
C.1 Un exemple simple . . . 271
C.2 Quelques éléments de Caml . . . 276
C.2.1 Fonctions . . . 277
C.2.2 Symboles, séparateurs, identificateurs . . . 278
C.2.3 Types de base . . . 279
C.2.4 Expressions . . . 280
C.2.5 Blocs et portée des variables . . . 282
C.2.6 Correction des programmes . . . 283
C.2.7 Instructions . . . 283
C.2.8 Exceptions . . . 285
C.2.9 Entrées – Sorties . . . 285
C.2.10 Définitions de types . . . 287
C.2.11 Modules . . . 292
C.2.12 Fonctions graphiques . . . 295
C.3 Syntaxe BNF de Caml . . . 298
C.4 Programmes en Caml . . . 303

D Initiation au système Unix **315**
D.1 Trousse de Survie . . . 315
D.1.1 Se connecter . . . 315
D.1.2 Se déconnecter . . . 316
D.1.3 Le système de fenêtres par défaut . . . 316
D.1.4 Obtenir de l'aide . . . 317
D.1.5 Changer son mot de passe . . . 317
D.1.6 Courrier électronique . . . 317
D.1.7 Polyaf . . . 318
D.1.8 Éditeur de texte . . . 319
D.1.9 Manipulations simples de fichiers . . . 319
D.1.10 Cycles de mise au point . . . 319
D.1.11 Types de machines . . . 320
D.2 Approfondissement . . . 320
D.2.1 Système de fichiers . . . 320
D.2.2 Raccourcis pour les noms de fichiers . . . 322
D.2.3 Variables . . . 323
D.2.4 Le chemin d'accès aux commandes . . . 324
D.2.5 Quotation . . . 324
D.2.6 Redirections et filtres . . . 325
D.2.7 Processus . . . 327
D.2.8 Programmation du shell . . . 327
D.3 Unix et le réseau de l'X . . . 329
D.4 Bibliographie Unix . . . 331

Bibliographie **333**

Table des figures **338**

Index **339**

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, 47], 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, notamment pour les dessins, 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 C sur le langage Caml, Damien Doligez et Xavier Leroy, auteurs de l'annexe D sur le système Unix, tant réclamée par les élèves, et Dominique Moret du Centre Informatique pour l'adaptation de cette annexe au contexte de l'X.

Enfin, merci à Pascal Brisset, auteur de la version électronique de ce cours consultable grâce au *World Wide Web* sur le réseau Internet à l'adresse

`http://www.polytechnique.fr/poly/~www/poly/`

Polycopié, version 1.4

Introduction

En 30 ans, l'informatique est devenue une industrie importante: 10% des investissements hors bâtiments des sociétés françaises. Au recensement de 1982, 50000 cadres techniciens se déclaraient informaticiens, 150000 en 1991. Une certaine 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* [39], 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 [30] de L. Lamport (jeu de macros TeX de D. Knuth [25]), les caractères ont été calculés en METAFONT de D. Knuth [26], les dessins en `gpic` version Gnu (R. Stallman)

de `pic` de B. Kernighan [22]. On ne comptera ni le système Macintosh (qui dérive fortement de l'Alto et du Dorado faits à Xerox PARC [50, 31]), ni OzTeX (TEX sur Macintosh), ni les éditeurs QED, Emacs, Alpha, ni le système Unix fait à Bell laboratories [43], les machines Dec Stations 3100, 5000, Sun Sparc2, ni leurs composants MIPS 2000/3000 [20] ou Sparc, ni le réseau Ethernet (Xerox-PARC [35]) 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 [18]). 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 exac-

tement 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 leur analyse de 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 [46]. 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 [14], Berstel-Pin-Pocchiola [8], Manber [34], Graham-Knuth-Patashnik [16].... Il faut aussi bien sûr signaler les livres de Knuth [27, 28, 29]. 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 [21], sur le livre de Wirth pour Pascal [19], sur le livre de Nelson [38] ou Harbison pour Modula-3[17].

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 Pascal (langage enseigné en classes préparatoires) et une autre dans le langage C à la fin de chaque chapitre. Un rappel de Pascal figurera en annexe, ainsi qu'une introduction au langage C fortement inspirée de [21]. 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 Pascal ou en C devra plutôt commencer par les appendices.

Le choix de considérer aussi le langage C est dicté par des considérations techniques, plus que philosophiques. Pascal est un langage vieillissant. Son successeur potentiel Modula-3 [38] ne fonctionne pas encore sur des petites machines et n'a pas de bon environnement de programmation. L'avantage de C est qu'il existe pratiquement sur toutes les machines. D'autres solutions plus avancées technologiquement, comme C++ [48], Scheme [1] ou ML [15, 36, 51], nous semblent trop distinctes de Pascal et imposent donc un choix. Il faudrait alors passer une bonne partie du cours à expliquer le langage, ce que nous nous refusons à faire. Par C, nous entendrons C ANSI avec la convention ANSI pour les paramètres des procédures. Le langage C a ses inconvénients, le principal est de ne pas avoir de garde-fous. On peut donc écrire des programmes inesthétiques, sous couvert d'efficacité. Pourtant, cet argument n'est même plus vrai avec l'avènement des machines RISC. On peut simplement écrire en C comme en Pascal. C'est le style que nous adopterons ici, et espérons que C pourra être considéré comme du "sucre syntaxique" autour de Pascal. On gagnera certainement un point par rapport à Pascal: l'intégration du langage au système d'exploitation pour accéder aux fichiers ou autres ressources de la machine, en particulier s'il s'agit d'une machine Unix.

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 [39] est interdit dans notre cours.

```
#define _ -F<00 || --F-00--;
int F=00,00=00;
main(){F_00();printf("%1.3f\n", 4.*-F/00/00);}F_00()
{
            _-_-_-_
       _-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_
            _-_-_-_
 }
```

Ce programme fait un nombre inconsidéré d'appels au préprocesseur et beaucoup trop d'effets de bord (cf. page 240). 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, et notamment du programmeur C, 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 le Vax 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);

3.1415926535897932384626433832795028841971693993751058209749445923078\
1640628620899862803482534211706798214808651328230664709384460955058223\
1\
7253594081284811174502841027019385211055596446229489549303819644288109\
7566593344612847564823378678316527120190914564856692346034861045432664\
8213393607260249141273724587006606315588174881520920962829254091715364\
3678925903600113305305488204665213841469519415116094330572703657595919\
5309218611738193261179310511854807446237996274956735188575272489122793\
8183011949129833673362440656643086021394946395224737190702179860943702\
7705392171762931767523846748184676694051320005681271452635608277857713\
4275778960917363717872146844090122495343014654958537105079227968925892\
3542019956112129021960864034418159813629774771309960518707211349999998\
3729780499510597317328160963185950244594553469083026425223082533446850\
3526193118817101000313783875288658753320838142061717766914730359825349\
0428755468731159562863882353787593751957781857780532171226806613001927\
876611195909216420199

> 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. En dehors de toutes les représentations symboliques des scalaires, via les types simples, il y a deux grandes catégories d'objets scalaires dans les programmes: les entiers en virgule fixe, les réels en virgule flottante.

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	$4\ 294\ 967\ 296 = 4 \times 10^9$	Vax
64	$18\ 446\ 744\ 073\ 709\ 551\ 616 = 2 \times 10^{19}$	Alpha

Figure *i*.1 : Bornes supérieures des nombres entiers

Ainsi, les processeurs de 1992 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 tout de même désigner 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 Pascal, `maxint` $= 2^{n-1} - 1$, le plus grand entier positif. En C, c'est `INT_MAX` obtenu dans le fichier *include* `<limits.h>` (cf. l'annexe sur le langage C).

Une opération entre 2 nombres peut créer un débordement, c'est-à-dire atteindre 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. En fait, le langage C ne fait jamais ce test, et Pascal pratiquement jamais.

Les nombres flottants

La notation flottante sert à représenter les nombres réels pour permettre d'obtenir 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 Pascal, vaut

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

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

Figure *i*.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

Figure *i*.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 les 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 [29]. 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:=1;
for j:= 2 to N do
    if a[j] < a[m] then 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.

```
program TriParSelection;

const N = 10;
type  T = array [1..N] of integer; (* Le tableau à trier *)

var a: T;

procedure Initialisation;         (* On tire au sort des nombres *)
    var i: integer;               (* entre 0 et 100 *)
    begin
    for i:= 1 to N do
        a[i] :=  50 + round((random / maxint) * 50);
    end;

procedure Impression;
    var i: integer;
    begin
    for i:= 1 to N do
        write(a[i], ' ');
    writeln;
```

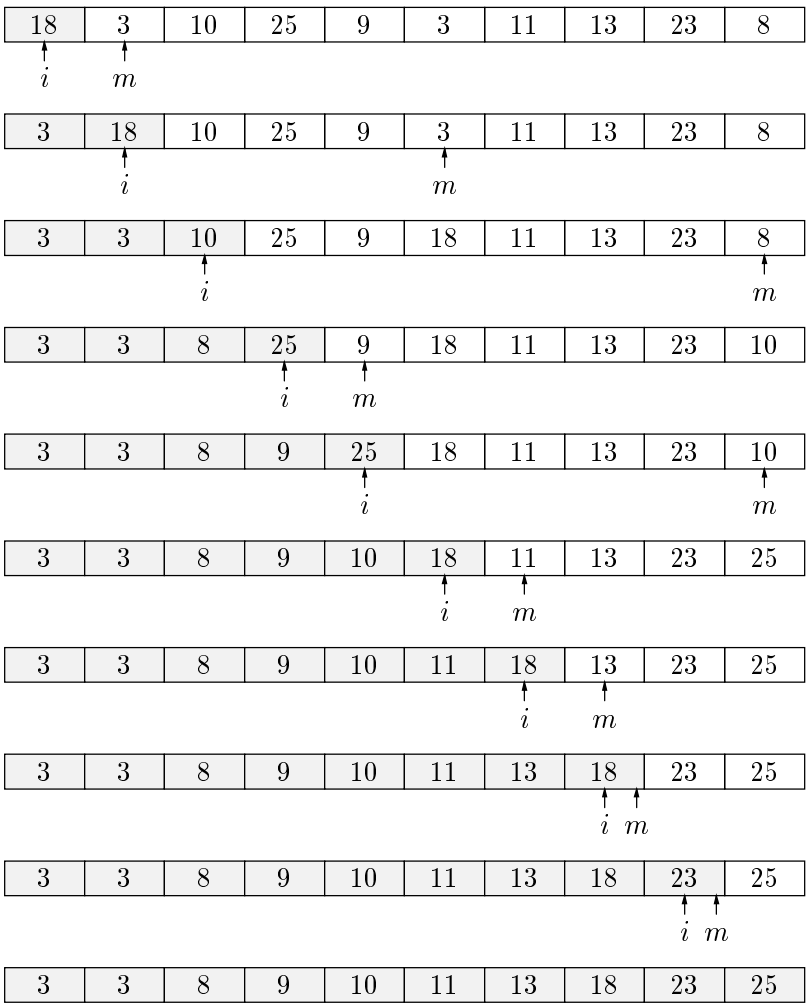


Figure 1.1 : Exemple de tri par sélection

```
    end;

procedure TriSelection;

    var i, j, m, t: integer;

    begin
    for i := 1 to N-1 do
        begin
        m := i;
        for j := i+1 to N do
            if a[j] < a[m] then m := j;
        t := a[m]; a[m] := a[i]; a[i] := t;
        end;
    end;

begin
Initialisation;    (* On initialise le tableau *)
TriSelection;      (* On trie *)
Impression;        (* On imprime le résultat *)
end.
```

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).

```
procedure TriBulle;

    var i, j, t: integer;

    begin
    for i:= N downto 1 do
      for j := 2 to i do
        if a[j-1] > a[j] then
           begin
           t := a[j-1]; a[j-1] := a[j]; a[j] := t;
           end;
```

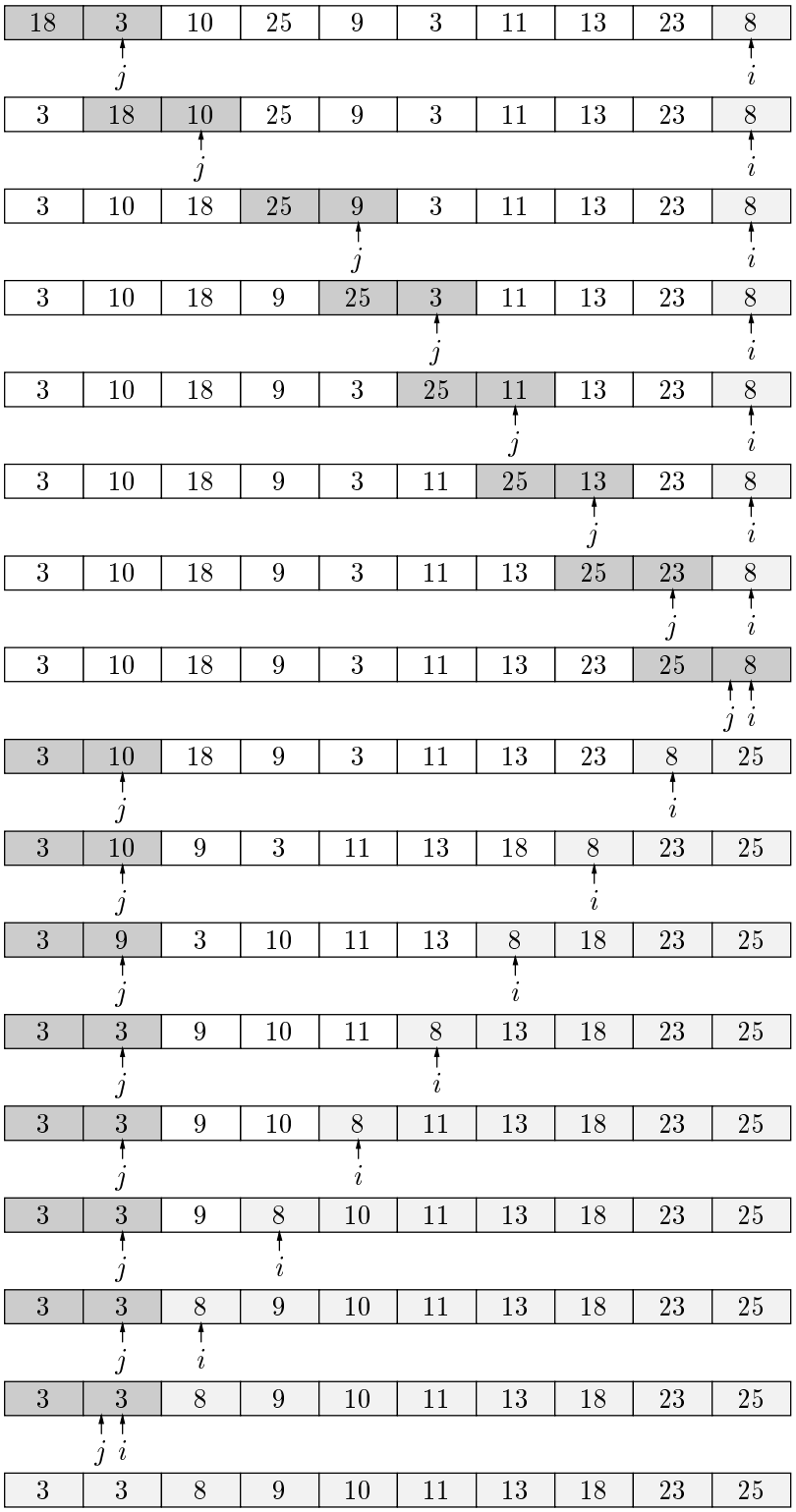


Figure 1.2 : Exemple de tri bulle

```
    end;
```

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 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 $\{1, 2 \ldots n\}$ 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 $\alpha = \langle a_1, a_2 \ldots a_N \rangle$ est la permutation $\beta = \langle a_N, \ldots, a_2, a_1 \rangle$. 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

```
procedure TriInsertion;

    var i, j, v: integer;
```

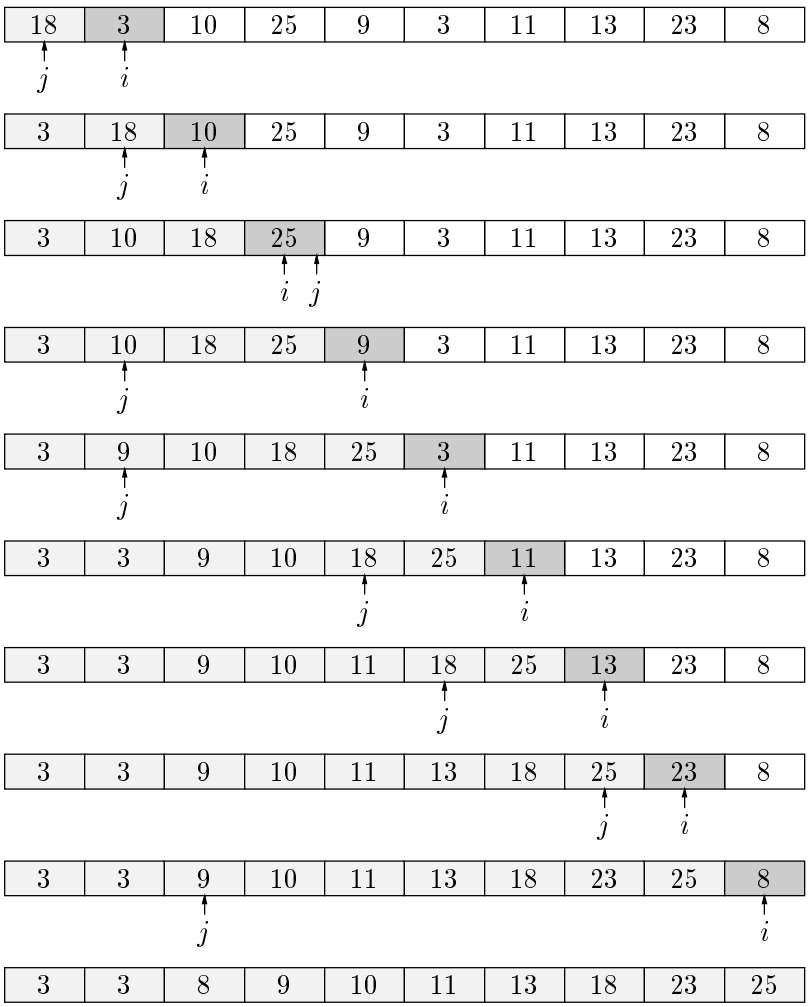


Figure 1.3 : Exemple de tri par insertion

```
begin
for i := 2 to N do
    begin
    v := a[i]; j := i;
    while a[j-1] > v do
        begin a[j] := a[j-1]; j:= j-1 end;
    a[j] := v;
    end;
end;
```

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 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

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

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

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

Figure 1.4 : Un exemple de table pour la recherche en table

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).

```
procedure TriShell;
    label 0;

    var i, j, h, v: integer;

    begin
    h := 1; repeat h := 3*h + 1 until h > N;
    repeat
        h := h div 3;
        for i := h + 1 to N do
            begin
            v := a[i]; j := i;
            while a[j-h] > v do
                begin
                a[j] := a[j-h]; j := j - h;
                if j <= h then goto 0;
                end;
          0:a[j] := v;
            end
    until h = 1;
    end;
```

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

```
function Recherche (x: Chaine): integer;
    label 1;
    var i: integer;
    begin
    for i := 1 to N do
        if x = nom[i] then
            begin
            Recherche := tel[i];
            goto 1;
            end;
    Recherche := -1;
  1:end;
```

Ce programme, avec une boucle à deux sorties, peut être écrit sans instruction `goto` de la manière suivante

```
function Recherche (x: Chaine): integer;
    var i: integer;
        succes: boolean;
    begin
    i := 1;
    succes := false;
    while not succes and (i <= N) do
        begin
        succes := x = nom[i];
        i := i + 1;
        end;
    if succes then
        Recherche := tel[i-1]
    else
        Recherche := -1;
    end;
```

Un booléen permet de faire le test sur l'égalité des noms à un endroit différent du test sur le débordement de la table. Ce mécanisme est une méthode générale pour éviter les `goto` dans des boucles à double sortie. Elle peut être plus ou moins élégante. Si on a la place, une autre possibilité est de mettre une *sentinelle* au bout de la table.

```
function Recherche (x: Chaine): integer;
    var i: integer;
    begin
    i := 1;
    nom[N+1] := x; tel[N+1] := -1;
    while (x <> nom[i]) do
        i := i + 1;
    Recherche := tel[i];
    end;
```

L'écriture de la procédure de recherche dans la table des noms est donc plus simple et 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.

```
program RechercheLineaire;

    const  N = 6;
           N1 = 7;
    type   Chaine = string[20];
    var    nom: array[1..N1] of Chaine;
           tel: array[1..N1] of integer;
           x: Chaine;

procedure Initialisation;
    begin
    nom[1] := 'paul';  tel[1] := 2811;
    nom[2] := 'roger';  tel[2] := 4501;
    nom[3] := 'laure';  tel[3] := 2701;
    nom[4] := 'anne';  tel[4] := 2702;
    nom[5] := 'pierre';  tel[5] := 2805;
    nom[6] := 'yves';  tel[6] := 2806;
    end;

function Recherche (x: Chaine): integer;
    var  i: integer;
    begin
    i := 1;
    nom[N + 1] := x;
    tel[N + 1] := -1;
    while (x <> nom[i]) do
        i := i + 1;
    Recherche := tel[i];
    end;

begin
Initialisation;
while true do
    begin
    readln(x);
    writeln(Recherche(x));
    end;
end.
```

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

```
procedure Initialisation;
    begin
    nom[1] := 'anne'; tel[1] := 2702;
    nom[2] := 'laure'; tel[2] := 2701;
    nom[3] := 'paul'; tel[3] := 2811;
    nom[4] := 'pierre'; tel[4] := 2805;
    nom[5] := 'roger'; tel[5] := 4501;
    nom[6] := 'yves'; tel[6] := 2806;
    end;

function RechercheDichotomique (x: Chaine): integer;
label 1;
var i, g, d: integer;
begin
    g := 1; d := N;
    RechercheDichotomique := -1;
    while g <= d do
        begin
        i := (g + d) div 2;
        if x = nom[i] then
            begin
            RechercheDichotomique := tel[i];
            goto 1;
            end;
        if x < nom[i] then
            d := i-1 else
            g := i+1;
        end;
    1:
end;
```

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 271). Ainsi

```
procedure Insertion (x: Chaine; val: integer);
begin
    n := n + 1;
    if n > Nmax then
        Erreur ('De''bordement de la table');
    nom[n] := x;
    tel[n] := val;
end;
```

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`,

```
function h (x: Chaine; l: integer): integer;
    var i, r: integer;
    begin
```

```
    r := 0;
    for i := 1 to l do
        r := ((r * B) + x[i]) mod N;
    h := r;
    end;
```

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 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

```
function Recherche (x: Chaine; l: integer): integer;
    var i: integer;
    begin
    i := h(x, l);
    while (nom[i] <> x) and (col[i] <> -1) do
        i := col[i];
    if (x = nom[i]) then
        Recherche := tel[i]
    else
        Recherche := -1;
    end;
```

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 68, 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

```
procedure Insertion (x: Chaine; l: integer; val: integer);
```

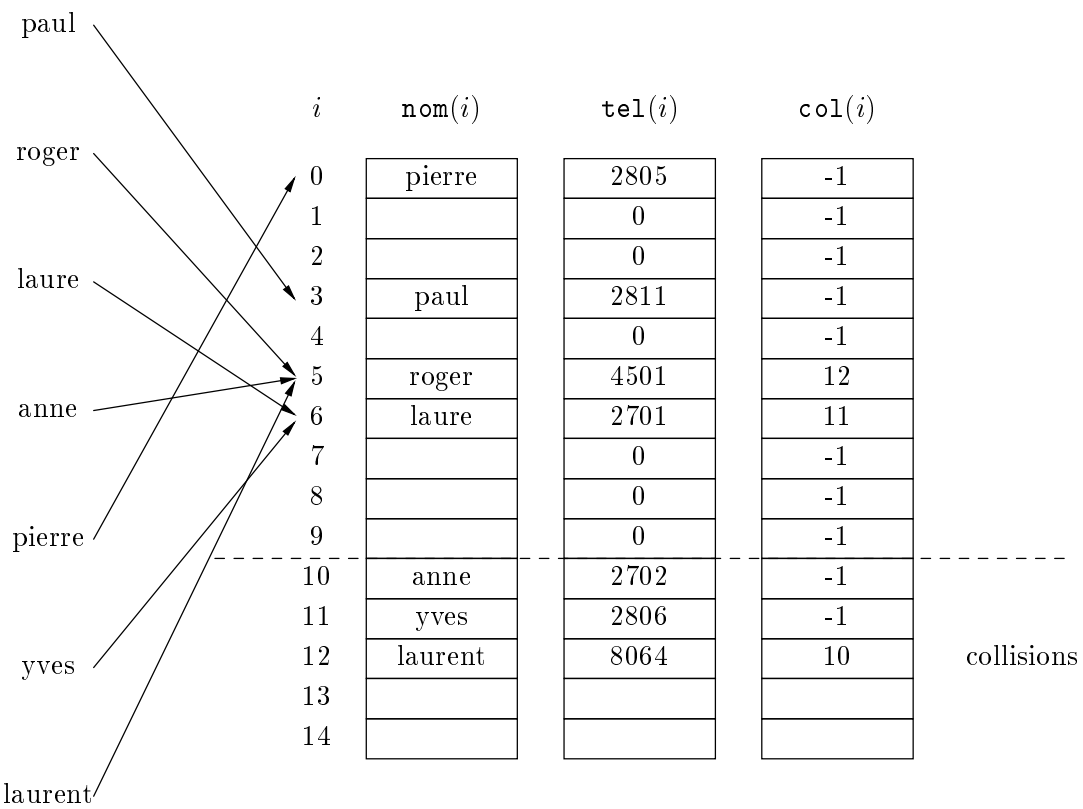


Figure 1.5 : Hachage par collisions séparées

```
    var i: integer;
    begin
    i := h(x, l);
    if nom[i] = '' then
        begin
        nom[i] := x;
        tel[i] := val
        end
    else
    if nNoms >= Nmax then
        Erreur ('De''bordement de la table');
    else
        begin
        nom[nNoms] := x; tel[nNoms] := val;
        col[nNoms] := col[i];     (* On met la nouvelle entrée en tête *)
        col[i] := nNoms;          (* de la liste des collisions de sa *)
        nNoms := nNoms + 1;       (* classe d'équivalence. *)
        end;
    end;
```

Pascal ne faisant malheureusement pas de différences entre minuscules et majuscules, la variable globale `nNoms` désignera n et permettra de le distinguer de `N` !! Au début, on suppose `nNoms` $=$ `N`, `nom`$[i] =$ `''` (chaîne vide) et `col`$[i] = -1$ pour $0 \leq i <$ `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

```
function Recherche (x: Chaine; l: integer): integer;
    var i: integer;
    begin
    i := h(x);
    while (nom[i] <> x) and (nom[i] <> '') do
        i := (i+1) mod Nmax;
    if nom[i] = x then
        Recherche := tel[i]
    else
        Recherche := -1;
    end;

procedure Insertion (x: Chaine; l: integer; val: integer);
    var i, r: integer;
    begin
    if nNoms >= Nmax then
        Erreur ('De''bordement de la table');
    nNoms := nNoms + 1;
    i := h(x);
    while (nom[i] <> x) and (nom[i] <> '') do
        i := (i+1) mod Nmax;
```

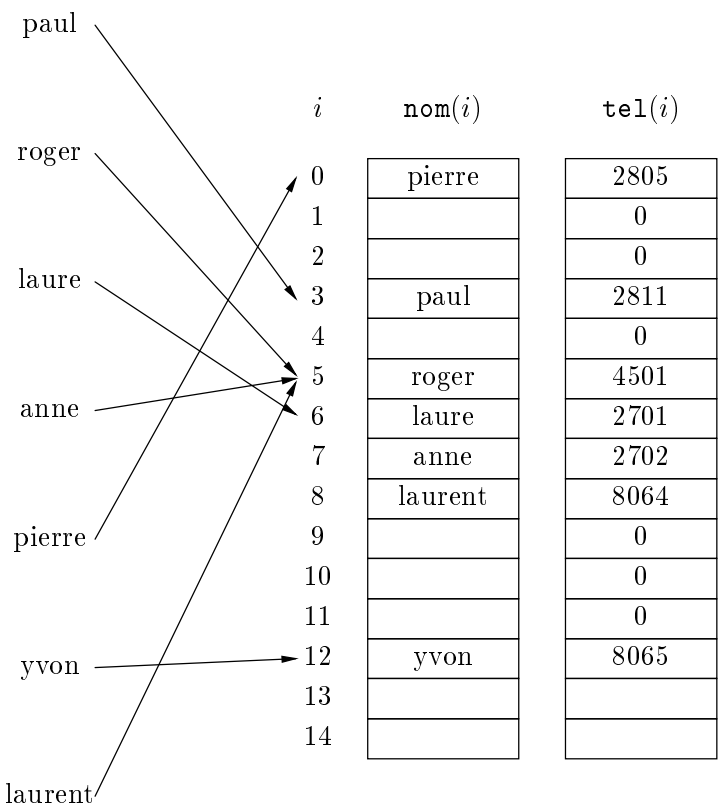


Figure 1.6 : Hachage par adressage ouvert

```
nom[i] := x;
tel[i] := val;
end;
```

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 $\alpha = \texttt{n/Nmax}$ 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 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

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

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 C

```
                        /* Tri par sélection, page 20 */
#include <stdio.h>      /* contient la signature de printf */
#include <stdlib.h>     /* contient la signature de rand */
#include <time.h>       /* contient la signature de clock */

#define N  10

int a[N];               /* Le tableau à trier */

void Initialisation()   /* On tire au sort des nombres */
{                       /* entre 0 et 127, en initialisant */
    int   i, s;         /* le tirage au sort sur l'heure */

    s = (unsigned int) clock();
    srand(s);
    for (i = 0; i < N; ++i)
        a[i] = rand() % 128;
}

void Impression()
{
    int   i;
    for (i = 0; i < N; ++i)
        printf ("%3d ", a[i]);
    printf ("\n");
}

void TriSelection()
{
    int   i, j, min, t;

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

[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).

```
        for (j = i+1; j < N; ++j)
           if (a[j] < a[min])
               min = j;
        t = a[min]; a[min] = a[i]; a[i] = t;
    }
}

int main()
{
    Initialisation();   /* On initialise le tableau */
    TriSelection();     /* On trie */
    Impression();       /* On imprime le résultat */
    return 0;
}
```

```
void TriBulle()                 /* Tri bulle, voir page 22 */
{
    int   i, j, t;

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

```
void TriInsertion()             /* Tri par insertion, voir page 24 */
{
    int   i, j, v;

    for (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;
    }
}
```

```
void TriShell()                 /* Tri Shell, voir page 27 */
{
    int   i, h;

    h = 1; do
       h = 3*h + 1;
    while ( h <= N );
    do {
        h = h / 3;
        for (i = h; i < N; ++i)
```

```
            if (a[i] < a[i-h]) {
                int   v = a[i], j = i;
                do {
                    a[j] = a[j-h];
                    j = j - h;
                } while (j >= h  &&  a[j-h] > v);
                a[j] = v;
            }
    } while ( h > 1);
}
```

```
int Recherche (char x[])          /* Recherche 1, voir page 28 */
{
    int   i;
    for (i = 0; i < N; ++i)
        if (strcmp(x, nom[i]) == 0)
            return tel[i];
    return -1;
}
```

```
int Recherche (char x[])          /* Recherche 2, voir page 28 */
{
    int   i = 0;
    while (i < N  &&  strcmp(x, nom[i]) != 0)
        ++i;
    if (i < N)
        return tel[i];
    else
        return -1;
}
```

```
int Recherche (char x[])          /* Recherche 3, voir page 28 */
{
    int   i = 0;
    nom[N] = x; tel[N] = -1;
    while (strcmp (x, nom[i]) != 0)
        ++i;
    return tel[i];
}
```

```
#include <string.h>
#include <stdio.h>
#define  N   6                    /* Recherche Linéaire, voir page 29 */
char   *nom[N+1];
int     tel[N+1];
char    x[100];

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;
}

int Recherche (char x[])
{
    int  i = 0;

    nom[N] = x; tel[N] = -1;
    while (strcmp (x, nom[i]) != 0)
        ++i;
    return tel[i];
}

int main()
{
    Initialisation();
    for (;;) {
        scanf("%s", x);
        printf("%d\n", Recherche(x));
    }
    return 0;
}
```

```
void Initialisation()            /* Recherche dichotomique, voir page 30 */
{
    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;
}

int RechercheDichotomique (char x[])
{
    int   i, g, d;

    g = 0; d = N-1;
    while (g <= d) {
        i = (g + d) / 2;
        if (strcmp (x, nom[i]) == 0)
            return tel[i];
        if (strcmp(x, nom[i]) < 0)
            d = i - 1;
        else
            g = i + 1;
```

```
    }
    return -1;
}
```

```
#include <stdlib.h>              /* pour la signature de malloc */
#include <string.h>              /* et des fonctions sur les chaînes  */

void Insertion (char x[], int val)    /* Insertion 1, voir page 31 */
{
    if (n >= N)
        Erreur ("De'bordement de la table");
    nom[n] = (char *) malloc (strlen (x) + 1);
    strcpy (nom[n], x);
    tel[n] = val;
    ++n;
```

```
#define B    128

int H (char x[])                 /* Fonction de hachage, voir page 31 */
{
    int   i, r;

    r = 0;
    for (i = 0; x[i] != 0; ++i)
        r = ((r * B) + x[i]) % N;
    return r;
}
```

```
int col[Nmax];

int Recherche (char x[])         /* Recherche avec hachage, voir page 32 */
{
    int   i;

    for (i = H(x); i != -1; i = col[i])
        if (strcmp (x, nom[i]) == 0)
            return tel[i];
    return -1;
}
```

```
void Insertion (char x[], int val)
{                                /* Insertion avec hachage, voir page 32 */

    int   i = H(x);

    if (nom[i] == NULL) {
        nom[i] = (char *) malloc (strlen(x) +1);
        strcpy (nom[i], x);
        tel[i] = val;
    } else
    if (n >= Nmax)
```

```
            Erreur ("De'bordement de la table");
        else {
            nom[n] = (char *) malloc (strlen(x) + 1);
            strcpy (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. */
        }
    }
```

```
int Recherche (char x[], int l)  /* Hachage avec adressage ouvert, voir page 34 */
{
    int   i = H(x);

    while (nom[i] != NULL) {
        if (strcmp(nom[i], x) == 0)
            return tel[i];
        i = (i+1) % N;
    }
    return -1;
}

void Insertion (char x[], int val)
{
    int   i;

    if (n >= N)
        Erreur ("De'bordement de la table");
    ++n;
    i = H(x);
    while ((nom[i] != NULL) && (strcmp (nom[i], x) != 0))
        i = (i+1) % N;
    nom[i] = (char *) malloc (strlen (x) + 1);
    strcpy (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:

```
function Fib(n:integer): integer;
    var r: integer;
    begin
    if n <= 1 then
        r := 1
    else
        r := Fib(n-1) + Fib(n-2);     (* on fait 2 appels récursifs *)
    Fib := r;
    end;
```

`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 Pascal, 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 Pascal, comme le confirment les exemples numériques suivants: factorielle et le triangle de Pascal.

```
function Fact(n: integer): integer;
    var r: integer;
    begin
    if n <= 1 then
```

```
        r := 1
    else
        r := n * Fact(n-1);
    Fact := r;
    end;

function C(n, p: integer): integer;
    var r: integer;
    begin
    if (n = 0) or (p = n) then
        r := 1
    else
        r := C(n-1, p-1) + C(n-1, p);
    C := r;
    end;
```

Remarque de syntaxe: la variable `r` n'est pas nécessaire dans les exemples précédents. Elle a été introduite pour rendre les programmes plus clairs, et éviter la confusion entre les appels récursifs à droite de l'affectation et la convention Pascal pour retourner un résultat en mettant le nom de la fonction à gauche du symbole d'affectation. En fait, on pouvait aussi écrire la fonction de Fibonacci de la façon suivante:

```
function Fib(n:integer): integer;
    begin
    if n <= 1 then
        Fib := 1
    else
        Fib := Fib(n-1) + Fib(n-2);   (* sinon on fait 2 appels récursifs *)
    end;
```

Par ailleurs, on peut se demander comment Pascal 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

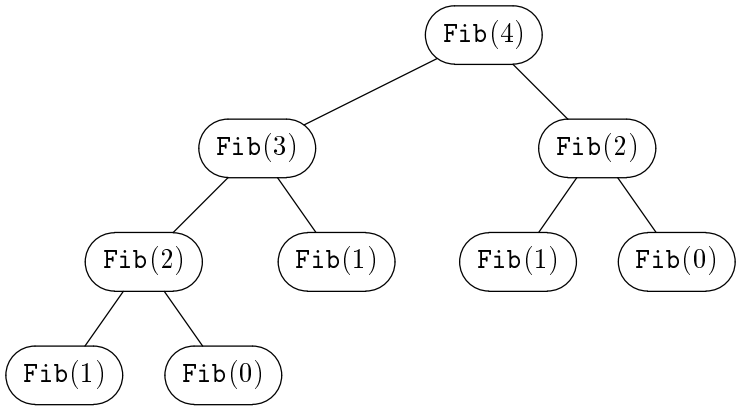


Figure 2.1 : Appels récursifs pour `Fib`(4)

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 `Fib`(n) et `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

```
function Fib (n: integer): integer;
    var u, v: integer;
        u0, v0: integer;
        i: integer;
    begin
    u := 1; v := 1;
    for i := 2 to n do
        begin
        u0 := u; v0 := v;
        u := u0 + v0;
        v := u0;
        end;
    Fib := u;
    end;
```

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 Pascal

```
function Ack (m, n: integer): integer;
    begin
    if m = 0 then
        Ack := n + 1
    else
        if n = 0 then
            Ack := Ack (m - 1, 1)
        else
            Ack := Ack (m - 1, Ack (m, n - 1));
    end;
```

On peut vérifier que

$$\begin{aligned}
\texttt{Ack}(0,n) &= n+1 \\
\texttt{Ack}(1,n) &= n+2 \\
\texttt{Ack}(2,n) &\simeq 2*n \\
\texttt{Ack}(3,n) &\simeq 2^n \\
\texttt{Ack}(4,n) &\simeq 2^{\overset{\overset{\overset{2}{\iddots}}{2^{2}}}{\left.\right\}n}}
\end{aligned}$$

Donc $\texttt{Ack}(5,1) \simeq \texttt{Ack}(4,4) \simeq 2^{65536} > 10^{80}$, 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

```
function f (n: integer): integer;
    begin
    if n > 100 then
        f := n - 10
    else
        f :=  f(f(n+11))
    end;
```

Ainsi, le calcul de $\texttt{f}(96)$ donne $\texttt{f}(96) = \texttt{f}(\texttt{f}(107)) = \texttt{f}(97) = \cdots \texttt{f}(100) = \texttt{f}(\texttt{f}(111)) = \texttt{f}(101) = 91$. 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

```
function g (m, n: integer): integer;
    begin
    if m = 0 then
```

[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

```
        g := 1
    else
        g :=  g(m - 1, g(m, n));
    end;
```

Que vaut alors $g(1,0)$? En effet, on a $g(1,0) = g(0, g(1,0))$. Il faut se souvenir que Pascal 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 Pascal 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(u)`*. Elle prend comme paramètre le nom de votre procédure et donne pour résultat* `true` *si la procédure 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:*
>
> ```
> procedure F;
>     var x: integer;
>     begin
>        x := 1;
>        writeln(x);
>     end;
> ```
>
> ```
> procedure G;
>     var x: integer;
>     begin
>        x := 1;
>        while (x > 0) do x:= x + 1;
>     end;
> ```
>
> *pour lesquels* `Termine(F) = true` *et* `Termine(G) = 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 sur la fonction suivante:

```
function Termine (procedure u): boolean;
    begin
```

```
        (* Contenu breveté par le vendeur *)
        end;

    procedure Absurde;
        begin
        while Termine(Absurde) do
            ;
        end;
```

Si la procédure `Absurde` boucle indéfiniment, alors `Termine(Absurde)` = `false`. Donc la boucle

```
        while Termine(Absurde) do
            ;
```

s'arrête, et la procédure `Absurde` termine. Sinon, si la procédure `Absurde` ne boucle pas indéfiniment, alors `Termine(Absurde)` = `true`. La boucle `while` précédente boucle indéfiniment, et la procédure `Absurde` boucle indéfiniment. Il y a donc une contradiction sur les valeurs possibles de `Termine(Absurde)`. 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 une fonction (en fait une "adresse mémoire"), désassemble la procédure correspondante, et décide de sa terminaison. Sinon, elle ne peut que lancer l'exécution de son argument et ne peut pas 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 Pascal 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

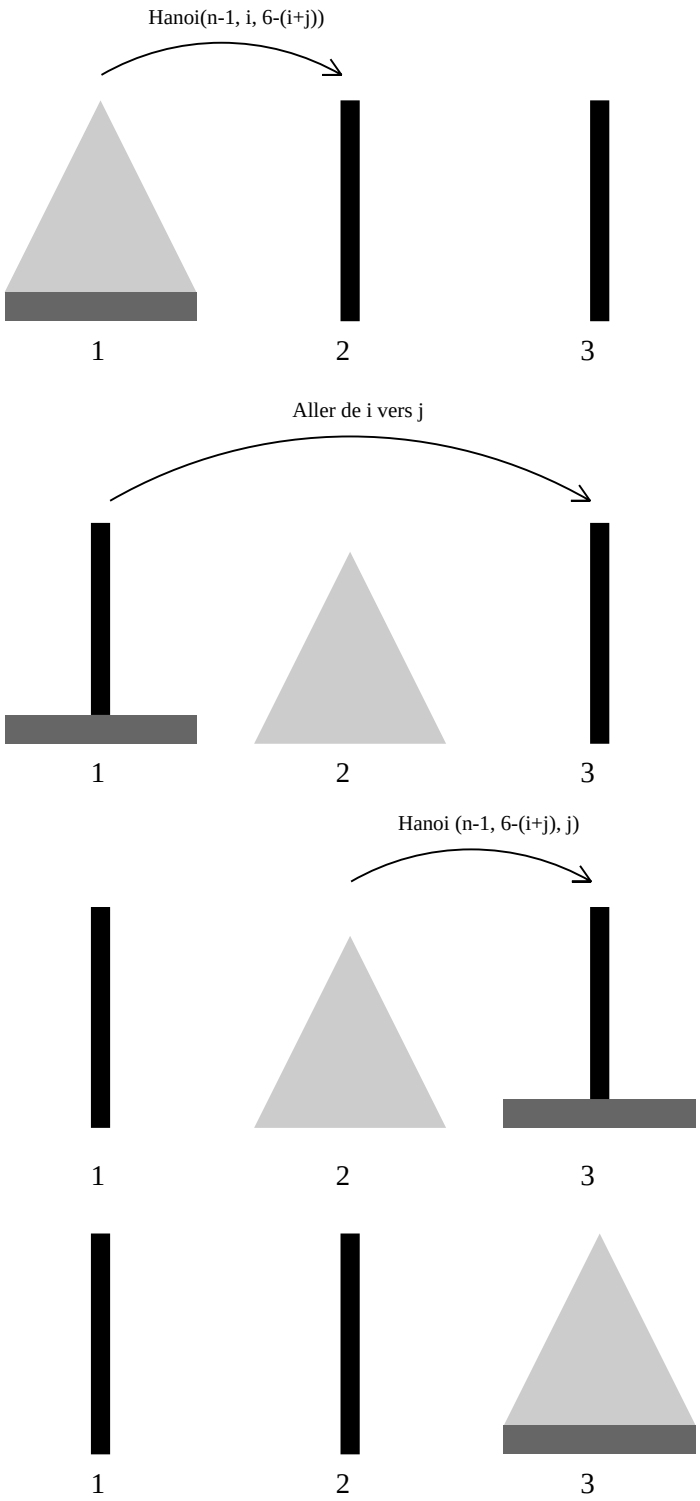


Figure 2.2 : Les tours de Hanoi

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

```
procedure Hanoi (n:integer; i, j: integer);
    begin
    if n > 0 then
        begin
        Hanoi (n-1, i, 6-(i+j));
        writeln (i:2, '->', j:2);
        Hanoi (n-1, 6-(i+j), j);
        end;
    end;
```

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 [24], Rogers [44]). 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 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 295). 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

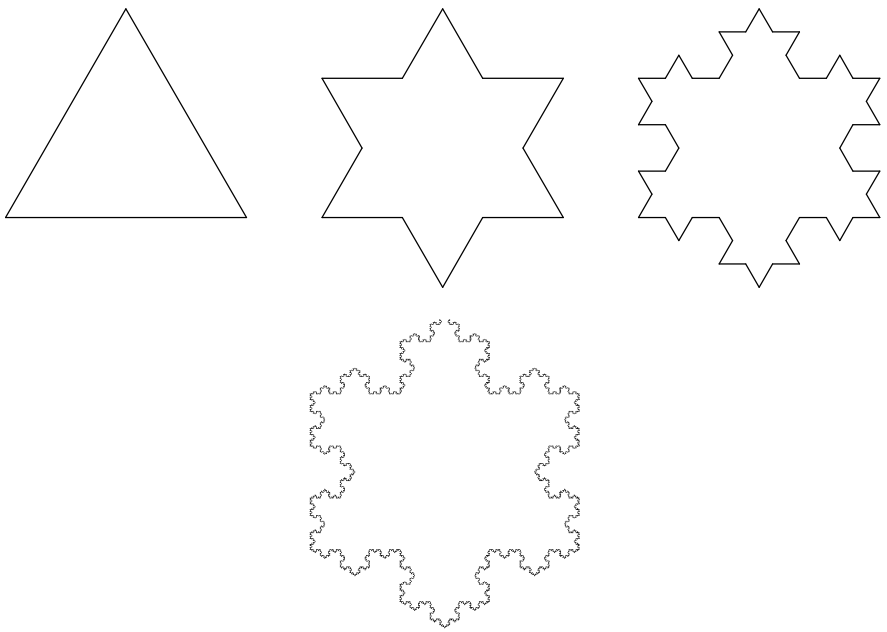


Figure 2.3 : Flocons de von Koch

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

La courbe se programme simplement par

```
procedure Dragon (n: integer; x, y, z, t: integer);
    var u, v: integer;
    begin
    if n = 1 then
        begin
        MoveTo (x, y);
        LineTo (z, t);
        end
    else
        begin
        u := (x + z + t - y) div 2;
        v := (y + t - z + x) div 2;
        Dragon (n-1, x, y, u, v);
        Dragon (n-1, z, t, u, v);
        end;
    end;
```

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

Figure 2.4 : La courbe du Dragon

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*. En Pascal, on utilise le mot clé `forward` pour cela.

```
procedure DragonBis (n: integer; x, y, z, t: integer);
    forward;

procedure Dragon (n: integer; x, y, z, t: integer);
    var u, v: integer;
    begin
    if n = 1 then
        begin
        MoveTo (x, y);
        LineTo (z, t);
        end
    else
        begin
        u = (x + z + t - y) div 2;
        v = (y + t - z + x) div 2;
        Dragon (n-1, x, y, u, v);
        DragonBis (n-1, u, v, z, t);
        end;
    end;

procedure DragonBis;
    var u, v: integer;
    begin
```

```
    if n = 1 then
        begin
        MoveTo (x, y);
        LineTo (z, t);
        end
    else
        begin
        u = (x + z - t + y) div 2;
        v = (y + t + z - x) div 2;
        Dragon (n-1, x, y, u, v);
        DragonBis (n-1, u, v, z, t);
        end;
    end;
```

Remarque de syntaxe: Pascal (comme C) exige que les types des procédures soient définis avant toute référence à cette procédure, d'où la déclaration `forward`. Mais Pascal (à la différence de C) exige aussi que la vraie définition de la procédure se fasse sans redéclarer la signature de la fonction (pour éviter de vérifier la concordance de types entre la signature des deux déclarations de la même fonction ou procédure!)

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

```
procedure QSort (g, d: integer);
    begin
    if g < d then
        begin
        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);
        end;
    end;
```

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

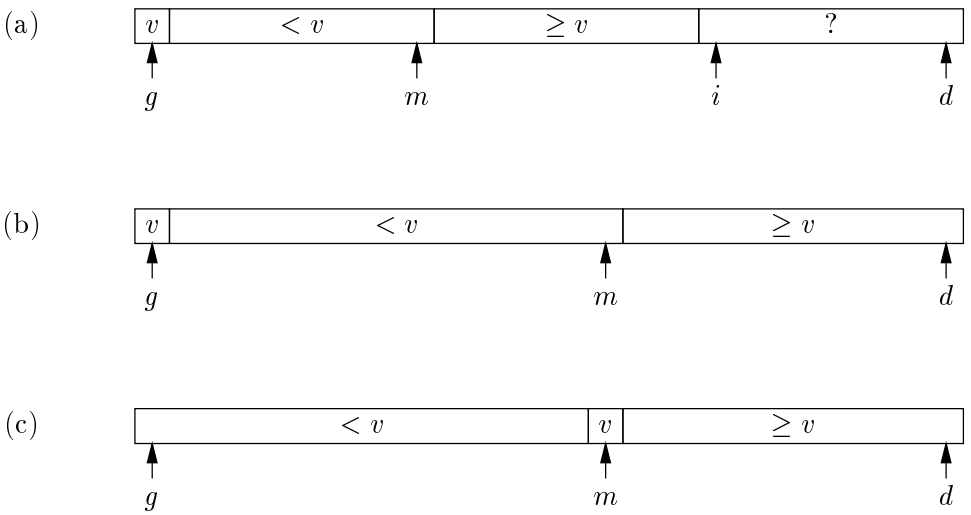


Figure 2.5 : Partition de Quicksort

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 to d do
    if a[i] < v then
        begin
        m := m + 1;
        x := a[m]; a[m] := a[i]; a[i] := x; (* Echanger a_m et a_i *)
        end;
```

ce qui donne la procédure suivante de Quicksort

```
procedure QSort (g, d: integer);
    var i, m, v, x: integer;
    begin
    if g < d then
        begin
        v := a[g];
        m := g;
        for i := g+1 to d do
            if a[i] < v then
                begin
                m := m + 1;
                x := a[m]; a[m] := a[i]; a[i] := x; (* Echanger a_m et a_i *)
                end;
        x := a[m]; a[m] := a[g]; a[g] := x; (* Echanger a_m et a_g *)
        QSort (g, m-1);
        QSort (m+1, d);
```

```
        end;
    end;
```

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 [46].

```
procedure QuickSort(g, d: integer);
var v,t,i,j:integer;
begin
   if g < d then
      begin
      v := a[d]; i := g-1; j := d;
      repeat
         repeat i := i+1 until a[i] >= v;
         repeat j := j-1 until a[j] <= v;
         t := a[i]; a[i] := a[j]; a[j] := t;
      until j <= i;
      a[j] := a[i]; a[i] := a[d]; a[d] := t;
      QuickSort (g, i-1);
      QuickSort (i+1, d);
      end;
end;
```

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 [46] 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. (Il y a un léger problème dans ce programme, puisque Pascal ne sait pas tester deux conditions en séquence — cf A.2.4). On suppose donc que l'on met deux sentinelles $a_{M+1} = \infty$ et $b_{N+1} = \infty$ pour ne pas compliquer la structure du programme.) On pose `M1` $= M + 1$, `N1` $= N + 1$, $P = M + N$, `P1` $= P + 1$.

```
var a: array [1..M1] of integer;
    b: array [1..N1] of integer;
    c: array [1..P1] of integer;
    i, j, k: integer;

    begin
    i := 1; j := 1;
    a[M+1] := maxint; b[N+1] := maxint;
    for k := 1 to P do
         if a[i] <= b[j] then
             begin
             c[k] := a[i];
             i := i + 1;
             end
         else
```

```
            begin
            c[k] := b[j];
            j := j + 1;
            end;
    end;
```

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 puisqu'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

```
procedure TriFusion (g, d: integer);
    var i, j, k, m: integer;
    begin
    if g < d then
        begin
        m := (g + d) div 2;
        TriFusion (g, m);
        TriFusion (m + 1, d);
        for i := m downto g do b[i] := a[i];
        for j := m+1 to d do b[d+m+1-j] := a[j];
        i := g; j := d;
        for k := g to d do
            if b[i] < b[j] then
                begin a[k] := b[i]; i := i + 1 end
            else
                begin a[k] := b[j]; j := j - 1 end;
        end;
    end;
```

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 C

```
int Fib(int n)                    /* Fibonacci, voir page 43 */
{
    if (n <= 1)
        return 1;
    else
        return Fib (n-1) + Fib (n-2);
}
```

```
int Fact(int n)                   /* Factorielle, voir page 43 */
{
    if (n <= 1)
        return 1;
    else
        return n * Fact (n-1);
}
```

```
int C(int n, int p)               /* Triangle de Pascal, voir page 43 */
{
    if ((n == 0) || (p == n))
        return 1;
    else
        return C(n-1, p-1) + C(n-1, p);
}
```

```
int Fib(int n)                    /* Fibonacci itératif, voir page 45 */
{
    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;
}
```

```
int Ack(int m, int n)         /* La fonction d'Ackermann, voir page 46 */
{
    if (m == 0)
        return n + 1;
    else
        if (n == 0)
```

```
            return Ack (m - 1, 1);
        else
            return Ack (m - 1, Ack (m, n - 1));
}
```

```
int f(int n)                /* La fonction 91, voir page 46 */
{
    if (n > 100)
        return n - 10;
    else
        return f(f(n+11));
}
```

```
int g(int m, int n)         /* La fonction de Morris, voir page 46 */
{
    if (m == 0)
        return 1;
    else
        return g(m - 1, g(m, n));
}
```

```
void Hanoi(int n, int i, int j)       /* Les tours de Hanoi, voir page 50 */
{
    if (n > 0) {
        Hanoi (n-1, i, 6-(i+j));
        printf ("%d -> %d\n", i, j);
        Hanoi (n-1, 6-(i+j), j);
}
```

```
void Dragon(int n, int x, int y, int z, int t)
{                                    /* La courbe du dragon, voir page 51 */
    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);
        Dragon (n-1, z, t, u, v);
    }
}
```

```
void Dragon(int n, int x, int y, int z, int t)
{                                     /* La courbe du dragon, voir page 52 */
    void DragonBis(int, int, int, int, int);
```

```
    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);
    }
}

void DragonBis(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);
    }
}
```

```
void QSort(int g, int d)            /* QuickSort, voir page 54 */
{
    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);
    }
}
```

```
void QuickSort(int g, int d)        /* Quicksort, voir page 55 */
{
    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);
    }
}
```

```
int   b[N];

void TriFusion(int g, int d)        /* Tri par fusion, voir page 57 */
{
    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;
            }
    }
}
```

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,

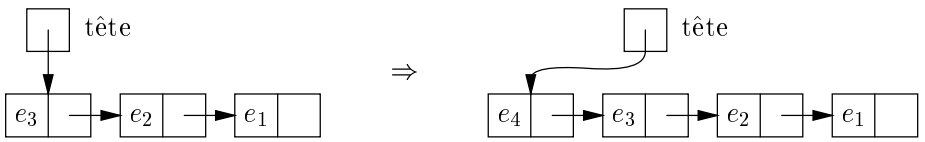


Figure 3.1 : Ajout d'un élément dans une liste

nous verrons que malgré ces restrictions les structures en question ont de nombreuses applications.

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 Pascal permet cette réalisation à l'aide de pointeurs: les cellules sont des enregistrements (`record`) dont un des champs contient l'adresse de la cellule suivante. L'adresse de la première cellule est elle contenue dans une variable de tête de liste. Les déclarations correspondantes sont les suivantes où l'on suppose que le type `Element` qui décrit la nature des éléments de l'ensemble considéré a déjà été déclaré.

```
type
    Liste = ^Cellule;
    Cellule = record
        contenu: Element;
        suivant: Liste;
        end;
var
    a: Liste;
```

Tout pointeur en Pascal peut prendre la valeur `nil` qui n'est l'adresse d'aucune cellule et qu'on utilise pour indiquer la fin de liste. Il existe aussi une procédure `new(u)` qui donne à son paramètre d'appel `u` l'adresse d'une cellule libre. Les opérations sur les ensembles que nous avons considérées ci-dessus s'expriment alors comme suit si on gère ceux-ci par des listes:

```
procedure FaireLvide (var a : Liste); (* Création de la liste a *)
    begin
    a := nil;
    end;

function Lvide (a: Liste): boolean;
     (* Teste si l'ensemble pointé par a est vide *)
     begin
```

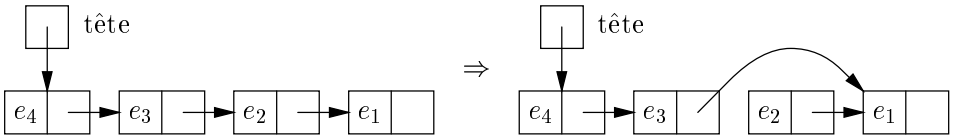


Figure 3.2 : Suppression d'un élément dans une liste

```
    Lvide:= a = nil;
    end;
```

La procédure `Ajouter` insère l'élément `x` en tête de liste. Ce choix de mettre l'élément en tête a été fait pour limiter le nombre d'opérations, il suffit en effet simplement de modifier la valeur de la tête de liste. La procédure a son deuxième argument passé par référence, et non par valeur (voir page 204).

```
procedure Lajouter(x: Element; var a: Liste);
    var u: Liste;
    begin
    new(u);    (* u est l'adresse où on va ajouter l'élément x *)
    u^.contenu := x;
    u^.suivant := a;
    a := u;
    end;
```

La fonction `Recherche` effectue un parcours de liste pour rechercher l'élément `x` dans la liste, la variable `adr` est modifiée itérativement par `adr := adr^.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 (`adr = nil`).

```
function Lrecherche (x: Element; a: Liste): boolean;
    var existe: boolean;
    begin
    existe := false;
    while (a <> nil) and (not existe) do
        begin
        existe := a^.contenu = x;
        a := a^.suivant;
        end;
    Lrecherche := existe;
    end;
```

La fonction `Recherche` peut aussi être écrite plus simplement de manière récursive

```
function Lrecherche (x: Element; a: Liste): boolean;
    begin
    if a = nil then
        Lrecherche := false        (* a est une liste vide *)
    else
    if a^.contenu = x then         (* a est une liste non vide *)
        Lrecherche := true
    else
        Lrecherche := Lrecherche (x, a^.suivant);
```

```
    end;
```

Cette écriture récursive est systématique. 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

```
function Llongueur (a: Liste): integer;
    begin
    if a = nil then                (* a = Liste_vide *)
        Llongueur := 0
    else                           (* a ∈ Element × Liste *)
        Llongueur := 1 + Llongueur (a^.suivant);
    end;
```

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

```
function Llongueur (a: Liste): integer;
    var resultat: integer;
    begin
    resultat := 0
    while a <> nil do
        begin
        resultat := 1 + resultat;
        a := a^.suivant;
        end;
    Llongueur := resultat;
    end;
```

Choisir entre la manière récursive ou itérative est affaire de goût. Autrefois, on disait que l'écriture itérative était plus efficace. C'est de moins en moins vrai, et l'occupation mémoire est maintenant 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. La fonction `dispose` redonne la place utilisée, celle ci pourra être réutilisée lors d'un `new`. Il faut toutefois noter qu'elle n'est pas toujours correctement réalisée par les compilateurs.

```
procedure Lsupprimer (x: Element; var a : Liste);
(* Cette procédure supprime l'élément x de la liste si celui ci y figure, *)
(* la liste est rendue telle quelle si l'élément x ne figure pas dans celle-ci *)
    var b: Liste;
    begin
    if a <> nil then
        if a^.contenu = x then
            begin
            b := a;
            a := a^.suivant;
```

```
            dispose(b)
            end
        else
            Lsupprimer (x, a^.suivant);
    end;
```

Une procédure itérative demande, comme souvent, beaucoup plus d'attention.

```
procedure LsupprimerIter (x: Element; var a :Liste);
    var b, c: Liste;
    function EstFini (x: Element; b: List): boolean;
        begin
        if b = nil then
            EstFini := true
        else
            EstFini := b^.contenu = x;
        end;

    begin
    if a <> nil then
        if a^.contenu = x then
            begin
            c := a;
            a := a^.suivant;
            dispose(c);
            end
        else
            begin
            b := a;
            while not EstFini(x, b^.suivant) do
                b := b^.suivant;
            if b^.suivant <> nil then
                begin
                c := b^.suivant;
                b^.suivant := c^.suivant;
                dispose(c);
            end;
    end;
```

Une autre version possible est la suivante

```
procedure LsupprimerIter (x: Element; var a :Liste);
    var b, c: Liste;
    begin
    if a <> nil then
        if a^.contenu = x then
            begin
            c := a;
            a := a^.suivant;
            dispose(c);
            end
        else
            begin
            b := a;
```

```
            c := a^.suivant;
            while c <> nil do
                if c^.contenu <> x then
                    begin
                    b := c;
                    c := c^.suivant;
                    end
                else
                    begin
                    b^.suivant := c^.suivant;
                    dispose(c);
                    c := nil;
                    end;
            end;
    end;
```

Une technique élégante peut permettre d'éviter la quantité de tests effectués 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 relevante 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 76) . 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 Les procédures *vide* et *ajouter* effectuent respectivement 1 et 4 opérations élémentaires. Elles sont donc particulièrement efficaces. 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 en on supprime tous leurs multiples stricts. Ceci se traduit par la procédure Pascal suivante :

```
procedure ListePremiers (n: integer; var a : Liste);
    var a,b: Liste;
        i,j,k: integer;
    begin
    a := nil;
    for i:= n downto 2 do Ajouter (i,a);
    b := a;
    k:= b^.contenu;
    while k*k <= n do
        begin
        for j:= k to (n div k) do
            Lsupprimer (j*k, a);
        b := b^.suivant;
        k := b^.contenu;
        end;
     end;
```

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, bien plus utile, d'application des listes est la gestion des collisions dans le hachage dont il a été question au chapitre 1 (voir page 34). Il s'agit pour chaque entier i de l'intervalle $[1 \ldots N]$ de construire une liste chaînée L_i formée de toutes les clés x telles que $h(x) = i$. 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. Par contre, si h est bien choisie, la recherche est rapide.

```
var al: array[0 .. N − 1] of Liste;

procedure Insertion (x: Chaine; l: integer; val: integer);
    var i: integer;
    begin
    i := h(x, l);
    Lajouter (x, al[i]);
    end;

function Recherche (x: Chaine; l: integer): boolean;
    var existe: boolean; i: integer; a: Liste;
    begin
    i := h(x,l);
    a := al[i];
    existe := false;
    while (a <> nil) and (not existe) do
```

```
        begin
        existe := a^.nom = x;
        a := a^.suivant;
        end;
    if existe then
        Recherche := a^.tel
    else
        Recherche := -1
    end;
```

3.2 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*:

- *vide* est une application de $Pil(E)$ dans (*vrai, faux*), $vide(P)$ est égal à *vrai* si et seulement si la pile P est vide.
- *ajouter* est une application de $E \times Pil(E)$ dans $Pil(E)$, $ajouter(x, P)$ est la pile obtenue à partir de la pile P en insérant l'élément x au sommet.
- *valeur* est une application de $Pil(E) \setminus P_0$ dans E qui à une pile P non vide associe l'élément se trouvant en son sommet.
- *supprimer* est une application de $Pil(E) \setminus P_0$ dans $Pil(E)$ qui associe à une pile P non vide la pile obtenue à partir de P en supprimant l'élément qui se situe en son sommet.

Les opérations sur les piles satisfont les relations suivantes

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

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:

$$supprimer\ (ajouter\ (7,\ supprimer\ (ajouter\ (valeur\ (ajouter\ (5,\ ajouter\ (3, P_0)))),\ ajouter\ (9, P_0))))$$

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. Ceci s'effectue comme suit:

```
const MaxP = 100;
type Pile = record
    hauteur:integer;
    contenu: array[1..MaxP] of Element;
    end;
var p: Pile;

procedure FairePvide (var p: Pile);
    begin
    p.hauteur:=0;
    end;

function Pvide (var p: Pile): boolean;
    begin
    Pvide := p.hauteur = 0;
    end;

function Pvaleur (var p: Pile): Element;
    begin
    Pvaleur := p.contenu[p.hauteur];
    end;

procedure Pajouter (x: Element; var p: Pile);
    begin
    p.hauteur := p.hauteur + 1;
    p.contenu[p.hauteur] := x;
    end;

procedure Psupprimer (var p: Pile);
    begin
    p.hauteur := p.hauteur - 1;
    end;
```

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. On peut aussi constater que les tests de débordement ont été délaissés, et que 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.

3.3 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, 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))
(* (+ 5 (* 2 3)) (+ (* 10 10) (* 9 9)))
```

Leur évaluation donne respectivement 47, 6, 20, 26 et 1991.

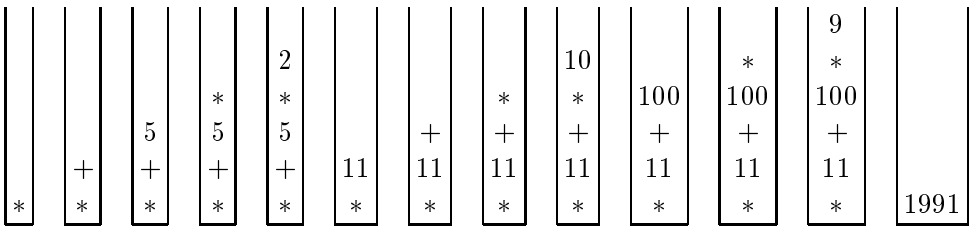
Pour représenter une expression préfixée en Pascal, on utilise ici un tableau dont chaque élément représente une entité de l'expression. Ainsi les expressions ci-dessus seront représentées par des tableaux de tailles respectives 1, 5, 5, 13, 25. Les éléments du tableau sont des enregistrements (`record`) 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.

```
type Element = record
    nature: (Nombre, Symbole);
    valeur: integer;
    valsymb: char;
    end;
    Expression = array[1..Nmax] of Element;
```

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

```
function Calculer (a: char; x, y: integer): integer;
    begin
    case a of
        '+' : Calculer := x + y;
        '*' : Calculer := x * y;
        end;
    end;
```

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.


* + 5 * 2 11 + * 10 100 * 9 1991
* * + + * * + + + + * * *

```
    y: Element;
begin
FairePvide(p);
for i := 1 to l do
    if (x[i].nature = Nombre) or  (* Supression des parenthèses *)
       (x[i].valsymb = '+') or    (* ouvrantes et fermantes *)
       (x[i].valsymb = '*') then
        Inserer (x[i],p);
    y := Pvaleur (p);
    Evaluer := y.valeur;
end;
```

3.4 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. Sur les files, on définit à nouveau les opérations *vide*, *ajouter*, *valeur*, *supprimer* sur les files comme sur les piles. Cette fois, les relations satisfaites sont les suivantes (où F_0 dénote la file vide).

Si $F \neq F_0$

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

Pour toute file F

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

Pour la file F_0

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

Une première idée de réalisation sous forme de programmes des opérations sur les files est empruntée à une technique mise en œuvre 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.

Dans la suite, on a représenté toutes ces opérations en Pascal en optimisant la place prise par la file en utilisant la technique suivante: on réattribue le numéro 1 à un nouveau client lorsque l'on atteint un certain seuil pour la valeur de *fin*. On dit qu'on a un tableau circulaire. Une vérification systématique du fait que la file n'est pas pleine doit alors être effectuée à chaque ajout.

```
const MaxF = 100;
type Intervalle = 1..MaxF;
    Fil = record                    (* file est un mot réservé *)
    fin, debut: Intervalle;
    contenu: array[Intervalle] of Element;
    end;

procedure FaireFvide (var f:Fil);
    begin
    f.debut := 0;
    f.fin := 0;
    end;

function Successeur (x: Intervalle): Intervalle;
    begin
    if x = MaxF then
        Successeur := 1
    else
        Successeur := x+1;
    end;

function Fvide (var f: Fil): boolean;
    begin
    Fvide := f.fin = f.debut;
    end;

function Fpleine (var f: Fil): boolean;
    begin
    Fpleine := f.debut = Successeur (f.fin);
    end;

function Fvaleur (var f: Fil): Element;
    var i: Intervalle;
    begin
    i := Successeur (f.debut);
    Fvaleur := f.contenu[i];
    end;

procedure Fajouter (x: Element; var f: Fil);
    begin
    f.fin := Successeur (f.fin);
    f.contenu[f.fin] := x;
    end;

procedure Fsupprimer (var f: Fil);
    begin
```

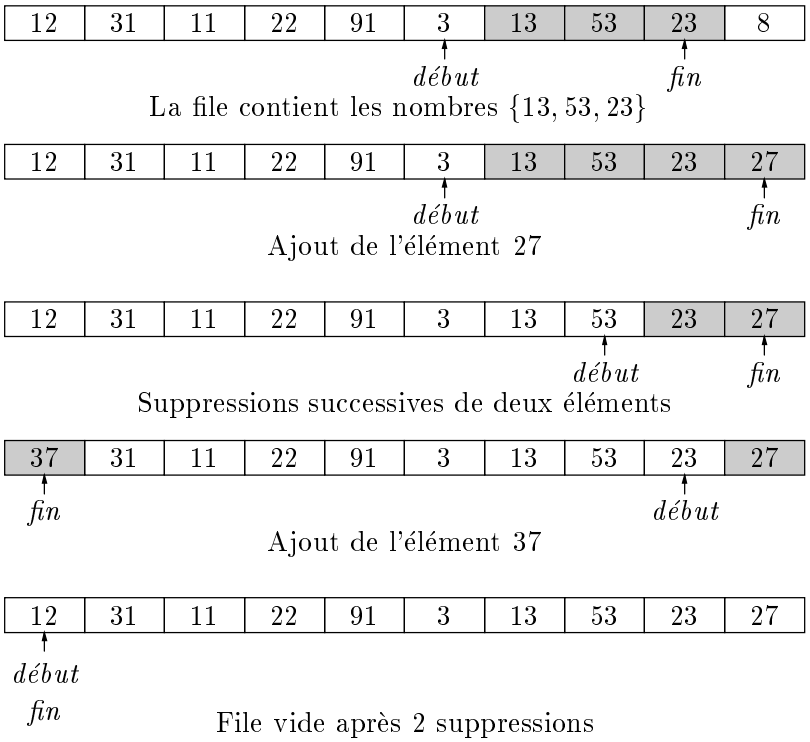


Figure 3.4 : File gérée par un tableau circulaire

```
    f.debut := Successeur (f.debut);
    end;
```

Pour ne pas alourdir les programmes ci dessus, nous n'avons pas ajouté des tests vérifiant si la file `f` n'est pas vide lorsqu'on supprime un élément et si elle n'est pas pleine lorsqu'on en ajoute un. Il est par contre vivement conseillé de le faire dans un programme d'utilisation courante. Cette gestion des files en tableau est souvent appelée *tampon circulaire*.

Une autre façon de gérer des files consiste à utiliser des listes chaînées gardées (voir page 68) dans lesquelles on connaît à la fois l'adresse du premier et du dernier élément. Cela donne les opérations suivantes:

```
type
    Liste = ^Cellule;
    Cellule = record
        contenu: Element;
        suivant: Liste;
        end;
    Fil = record
        debut: Liste;
        fin: Liste;
        end;
procedure FaireFvide(var f: Fil);
    begin
```

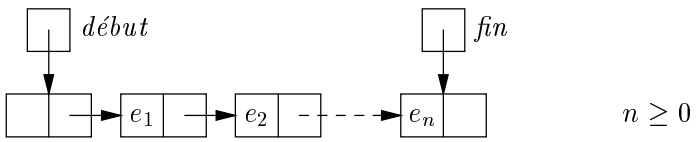


Figure 3.5 : File d'attente implémentée par une liste

```
    New(f.debut);
    f.fin := f.debut;
    end;

function Successeur (a: Liste): Liste;
    begin
    Successeur := a^.suivant;
    end;

function Fvide (f: Fil): boolean;
    begin
    Fvide := f.fin = f.debut;
    end;

function Fvaleur (f: Fil): Element;
    var b: Liste;
    begin
    b := Successeur (f.debut);
    Fvaleur := b^.contenu;
    end;

procedure Fajouter (x: Element; var f: Fil);
    var b: Liste;
    begin
    new (b);
    b^.contenu := x;
    b^.suivant := nil;
    f.fin^.suivant := b;
    f.fin := b;
    end;

procedure Fsupprimer (var f: Fil);
    var b: Liste;
    begin
    if not Fvide(f) then
        begin
        b := f.debut;
        f.debut := Successeur (f.debut);
        dispose (b)
        end
    end;
```

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

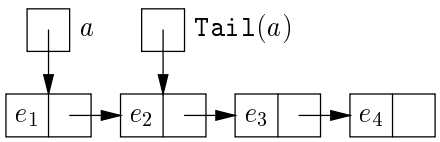


Figure 3.6 : Queue d'une liste

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 plus tard.

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. Les fonctions `Tail` et `Cons` sont des primitives classiques, la première supprime le premier élément d'une liste, la seconde est la version sous forme de fonction de la procédure `Ajouter` vue à la section 3.1.

```
type Liste = ^Cellule;
     Cellule = record
        contenu : Element;
        suivant: Liste;
    end;

function Tail (a: Liste): Liste;
    begin
    if x <> nil then
        Tail := a^.suivant
    else
        Erreur('Tail d'une liste vide');
    end;

function Cons (v: integer; a: Liste): Liste;
    var b: Liste;
    begin
    new (b);
    b^.contenu := v;
    b^.suivant := a;
    Cons := b;
    end;
```

Des procédures sur les listes construisent une liste à partir de deux autres, la première appelée `Append` consiste à 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

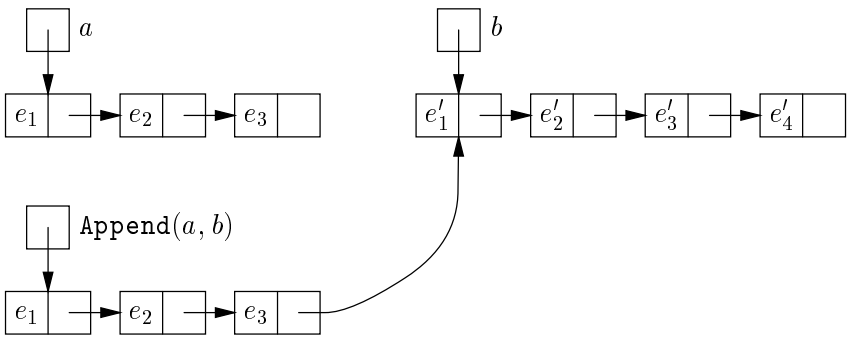


Figure 3.7 : Concaténation de deux listes par *Append*

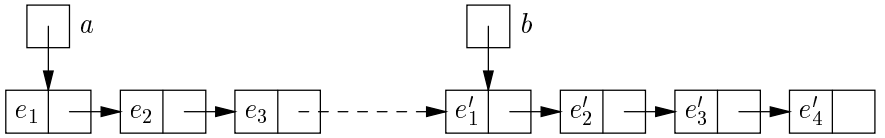


Figure 3.8 : Concaténation de deux listes par *Nconc*

`Append` copie son premier argument, il partage la fin de liste de son résultat avec son deuxième argument.

```
function Append (a: Liste; b: Liste): Liste;
    begin
    if a = nil then
        Append := b
    else
        Append := Cons (a^.contenu, Append (a^.suivant, b));
    end;

procedure Nconc (var a: Liste; b: Liste);
    var c: Liste;
    begin
    if a = nil then
        a := b
    else
        begin
        c := a;
        while c^.suivant <> nil do
            c := c^.suivant;
        c^.suivant := b;
        end;
    end;
```

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, quadratique mais classique. A nouveau, `Nreverse` modifie son argument, alors que `Reverse` ne le modifie pas et copie une nouvelle liste pour son résultat.

```
procedure Nreverse (var a: Liste);
    var b, c: Liste;
    begin
    b := nil;
    while a <> nil do
        begin
        c := a^.suivant;
        a^.suivant := b;
        b := a;
        a := c;
        end;
    a := b;
    end;

function Reverse (a: Liste): Liste;
    begin
    if a = nil then
        Reverse := nil
    else
        Reverse := Append (Reverse (a^.suivant), Cons (a^.valeur, nil));
    end;
```

Un autre exercice formateur consiste à gérer des listes dans lesquelles les éléments sont rangés en ordre croissant. La procédure d'ajout devient alors 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 68. 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.

```
procedure Insert (v: integer; var a: liste);
    var b: liste;
    begin
    b := a;
    while (b^.suivant <> a) and (v > b^.suivant^.contenu) do
        b := b^.suivant;
```

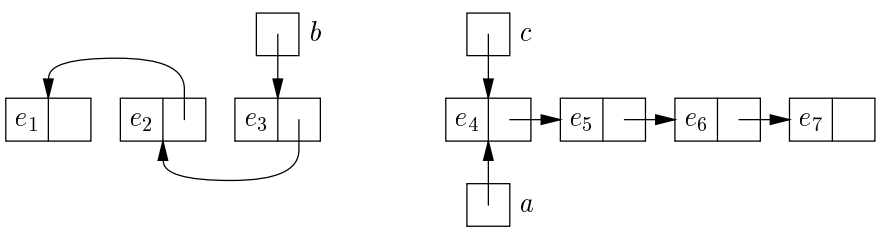


Figure 3.9 : Transformation d'une liste au cours de *Nreverse*

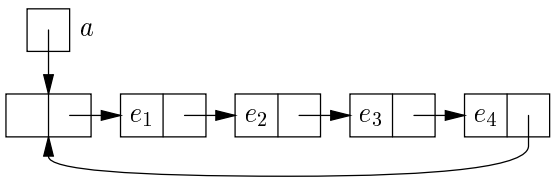


Figure 3.10 : Liste circulaire gardée

```
    b^.suivant := Cons (v, b^.suivant);
    a^.contenu := a^.contenu + 1;
    end;
```

3.6 Programmes en C

```
typedef int Element;

struct Cellule    {
    Element contenu;
    struct Cellule *suivant;
    };

typedef struct Cellule Cellule;
typedef struct Cellule *Liste;
```

Remarque: Nous voulons suivre les signatures des procédures Pascal. On aurait pu ne pas déclarer les types `Element` et `Liste`, et utiliser des déclarations très courantes en C comme :

```
int FaireLvide(Liste *ap)
{
    *ap = NULL;

}

int Lvide (Liste a)    /* voir plus bas */
                       /* Liste vide, voir page 64 */
{
    return a == NULL;
}
```

Remarque: `NULL` est défini dans `<stdio.h>`.

```
void  Lajouter(Element x, Liste *ap)     /* Ajouter, voir page 65 */
{
    Liste b;
    b  = (Liste) malloc(sizeof(Cellule));
    b -> contenu = x;
    b -> suivant = *ap;
    *ap = b;
}
```

```
int Lrecherche(Element x, Liste a)     /* Recherche, voir page 65 */
{
    while (a != NULL) {
        if (a -> cont == x)
            return 1;
        a = a -> suivant;
    }
    return 0;
}
```

```
int Llongueur(Liste a)     /* Longueur d'une liste, voir page 66 */
{
    if (a == NULL)
        return 0;
    else
        return 1 + Llongueur (a -> suivant);
}
```

```
int Llongueur(Liste a)     /* Longueur d'une liste, voir page 66 */
{
    int   r = 0;
    while (a != NULL) {
        ++r;
        a = a -> suivant;
    }
    return r;
}
```

```
void Lsupprimer(Element x, Liste *ap)    /* Supprimer, voir page 66 */
{
    Liste   b,c;

    b = *ap;
    if (b != NULL)
        if (b -> contenu == x){
            c = b;
            b = b -> suivant;
            free(c);
         } else
         Lsupprimer (x, &b -> suivant);
    *ap = b;
}
```

La fonction `LsupprimerIter` qui est non récursive s'écrit bien plus simplement en C:

```
void  LsupprimerIter (Element x, Liste *ap)
{
    Liste   a, b, c;
```

```
    a = *ap;
    if (a != NULL)
        if (a -> contenu == x){
            c = a;
            a = a -> suivant;
            free(c);
        } else {
            b = a ;
            while (b != NULL && b -> suivant -> contenu != x)
                b = b -> suivant;
            if (b != NULL) {
                c = b -> suivant;
                b -> suivant = b -> suivant -> suivant;
                free(c);
            }
        }
    *ap = a;
}
```

```
Liste ListePremier (int n)      /* Liste des nombres premiers, voir page 69 */
{
    Liste  a, b;
    int    i, j, k;

    FaireLvide (a);
    for (i = n; i >= 2; --i) {
         Lajouter (i, &a);
    }
    k = a -> contenu;
    for (b = a; k * k <= n ; b = b -> suivant){
        k = b -> contenu;
        for (j = k; j <= n/k; ++j)
            Lsupprimer (j * k, &a);
        }
    return(a);
}
```

Déclarations et opérations sur les piles voir page 71

```
struct Pile {
    int      hauteur ;
    Element  contenu[MaxP];
};

typedef struct Pile Pile;

int FairePvide(Pile *p)
{
    p -> hauteur = 0;
}

int Pvide(Pile *p)
```

```
{
    return p -> hauteur == 0;
}

void Pajouter(Element x, Pile *p)
{
    p -> contenu[p -> hauteur] = x;
    ++ p -> hauteur;
}

Element Pvaleur(Pile *p)
{
    int   i;

    i = p -> hauteur -1;
    return p -> contenu [i];
 }

void Psupprimer(Pile *p)
{
    -- p -> hauteur;
}
```

Evaluation des expressions préfixées voir page 72:

```
enum Nature {Symbole, Nombre};

struct Element {
     enum Nature   nature;
     int           valeur;
     char          valsymb;
     };
typedef struct Element Element;
typedef Expression Element[MaxP];

int Calculer (char a, int x, int y)
{
    switch (a) {
        case '+': return x + y;
        case '*': return x * y;
    }
}

void Inserer (Element x, Pile *p)
{
    Element    y, z;

    if (Pvide (p) || x.nature == Symbole)
        Pajouter(x, p);
    else {
        y = Pvaleur(p);
        if (y.nature == Symbole)
            Pajouter(y, p);
        else {
```

```
            Psupprimer(p);
            z = Pvaleur(p);
            Psupprimer(p);
            x.valeur = Calculer(z.valsymb, x.valeur, y.valeur);
            Inserer(x,p);
        }
    }
}

int  Evaluer (Expression u, int l)
{
    int i;
    Pile p;

    FairePvide (&p);
    for (i = 1; i <= l ; ++i)
        if (u[i].nature == Symbole ||
            u[i].valsymb == '+' ||
            u[i].valsymb == '*')
            Inserer(u[i] ,&p);
    return (Pvaleur (&p)).valeur;
}
```

```
#define MaxF    100
typedef int     Element;

typedef struct Fil {              /* les files représentées par */
    int         debut;            /* un vecteur voir page 75 */
    int         fin;
    Element     contenu[MaxF];
} Fil;

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

int Fvide(Fil *f)
{
    return  f -> debut == f -> fin;
}

void FaireFil(Fil *f)
{
    f -> debut = 0;
    f -> fin = 0;
}

int Fvaleur (Fil *f)
{
    int i = Successeur(f -> debut);
    return  f -> contenu[i];
}
```

```
void Fajouter (Element x, Fil *f)
{
    f -> fin = Successeur(f -> fin);
    f -> contenu[f -> fin] = x;
}

void Fsupprimer (Fil *f)
{
    f -> debut = Successeur(f -> debut);
}
```

```
typedef int    Element;     /* les files représentées par une liste voir page 76 */

typedef struct Cellule {
        Element         contenu;
        struct Cellule  *suivant;
} Cellule, *Liste;

typedef struct Fil{
        Liste           debut;
        Liste           fin;
} Fil;

void FaireFvide (Fil *f)
{
    f -> debut = (Liste) malloc (sizeof(Cellule));
    f -> fin = f -> debut;
}

Liste Successeur (Liste a)
{
    return a -> suivant;
}

int Fvide (Fil f)
{
    return f.debut == f.fin;
}

Element Fvaleur (Fil f)
{
    Liste b = Successeur(f.debut);
    return b -> contenu;
}

void Fajouter (Element x, Fil *f)
{
    Liste a = (Liste) malloc (sizeof(Cellule));

    a -> contenu = x;
    a -> suivant = NULL;
    f -> fin -> suivant = a;
    f -> fin = a;
}
```

```
void Fsupprimer (Fil *f)
{
    Liste a = f -> debut;

    f -> debut = Successeur (f -> debut);
    free(a);
}
```

```
Liste Tail (Liste a)              /* Tail et Cons voir page 78 */
{
    if (a == NULL) {
        fprintf(stderr, "Tail d'une liste vide.\n");
        exit (1);
    } else
        return a -> suivant;
}

Liste Cons (int v, Liste a)
{
    Liste b = (Liste) malloc (sizeof(Cellule));

    b -> contenu = v;
    b -> suivant = a;
    return b;
}
```

```
Liste Append (Liste a, Liste b)    /*  Append et Nconc voir page 79 */
{
    if (a == NULL)
        return b;
    else
        return Cons (a -> contenu, Append(a -> suivant, b));
}

void Nconc (Liste *ap, Liste b)
{
    Liste     c;

    if (*ap == NULL)
       *ap = b;
    else {
        c = *ap;
        while (c -> suivant != NULL)
             c = c -> suivant;
        c -> suivant = b;
    }
}
```

```
void Nreverse (Liste *ap)          /* Nreverse et Reverse, voir page 80 */
{
```

```
    Liste a, b, c;

    a = *ap;
    b = NULL;
    while (a != NULL) {
        c = a -> suivant;
        a -> suivant = b;
        b = a;
        a = c;
    }
    *ap = b;
}

Liste Reverse (Liste a)
{
    if (a == NULL)
        return  a;
    else
        return  Append (Reverse (a -> suivant),
                        Cons (a -> contenu, NULL));
}
```

```
void Insert (Element v, Liste *ap)            /* Insert, voir page 80 */
{
    Liste a, b;

    a = *ap;
    b = a -> suivant;
    while (b -> suivant != a && v > b -> suivant -> contenu)
         b = b -> suivant;
    b -> suivant = Cons (v, b -> suivant);
    ++ a -> contenu;
    *ap = a;
}
```

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 *nœud* et une suite de sous-arbres. Graphiquement, un arbre est représenté comme suit

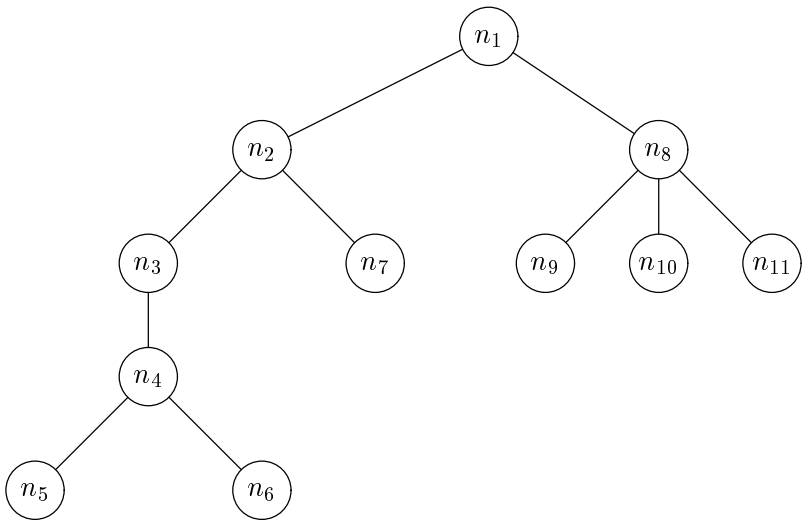


Figure 4.1 : Un exemple d'arbre

Le nœud 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 *nœuds internes*. Plus généralement, l'ensemble des *nœuds* est constitué des nœuds 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 nœud n_i à un nœud n_j plus bas, on dira que n_i est un *ancêtre* de n_j. Une propriété fondamentale d'un arbre est qu'un nœud n'a qu'un seul père. Enfin, un nœud peut contenir une ou plusieurs valeurs, et on parlera alors d'*arbres étiquetés* et de la valeur (ou des valeurs) d'un nœud. Les *arbres binaires* sont des arbres tels que les nœuds ont au plus 2 fils. La *hauteur*, on dit aussi la *profondeur* d'un nœud 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

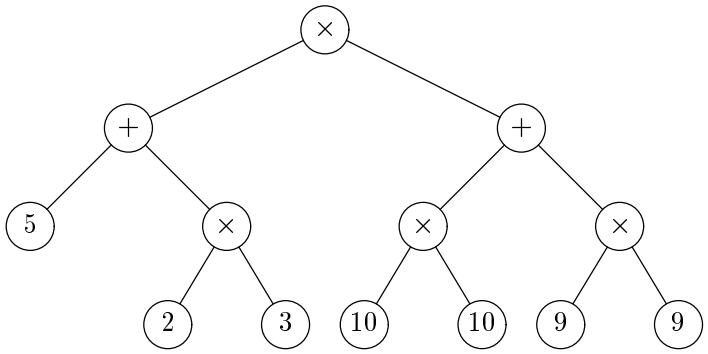


Figure 4.2 : Représentation d'une expression arithmétique par un arbre

nœuds 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 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 3.3 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 3.4, 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

[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.

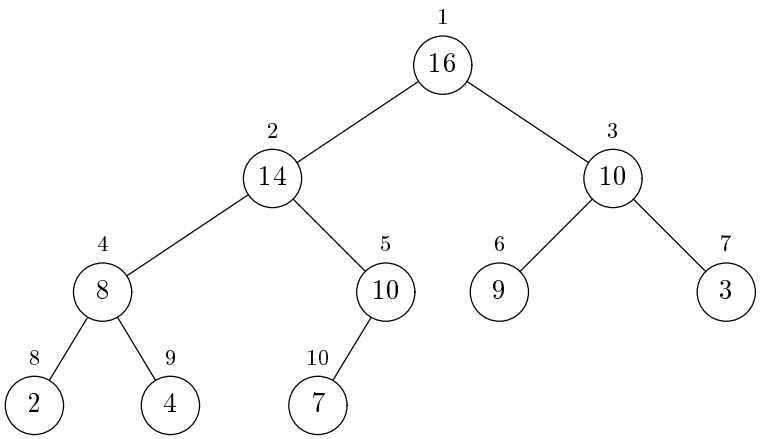


Figure 4.3 : Représentation en arbre d'un tas

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 nœud un élément de la file (comme illustré dans la figure 4.3). L'arbre vérifie deux propriétés importantes: d'une part la valeur de chaque nœud 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 nœuds 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 nœuds, et ainsi de suite (la ligne i contenant au plus 2^i nœuds). Dans un arbre quasi complet les lignes, exceptée peut être la dernière, contiennent toutes un nombre maximal de nœuds (soit 2^i). De plus les feuilles de la dernière ligne se trouvent toutes à gauche, ainsi tous les nœuds 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 nœud i a son père en position $\lfloor i/2 \rfloor$, le fils gauche du nœud 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:

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

Ceci permet d'implémenter cet arbre dans un tableau `a` (voir figure 4.4) où le numéro de chaque nœud 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 nœud n et celle

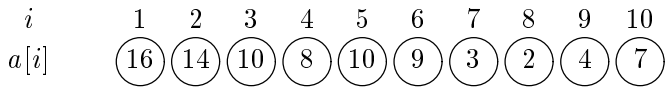


Figure 4.4 : Représentation en tableau d'un tas

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).

```
procedure Ajouter (v: integer);
    var i: integer;
    begin
    nTas := nTas + 1;
    i := nTas;
    a[0] := maxint;             (* sentinelle *)
    while a[i div 2] <= v do
        begin
        a[i] := a[i div 2];
        i := i div 2;
        end;
    a[i] := v;
    end;
```

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 nœuds 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 $O(1)$.

```
function Maximum: Element;
    begin Maximum := a[1] end;
```

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 nœud 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.

```
procedure Supprimer;
    label 0;
    var i, j: integer;
        v: Element;
    begin
    a[1] := a[nTas];
    nTas := nTas - 1;
    i := 1; v := a[1];
    while 2 * i <= nTas do
        begin
        j := 2 * i;
```

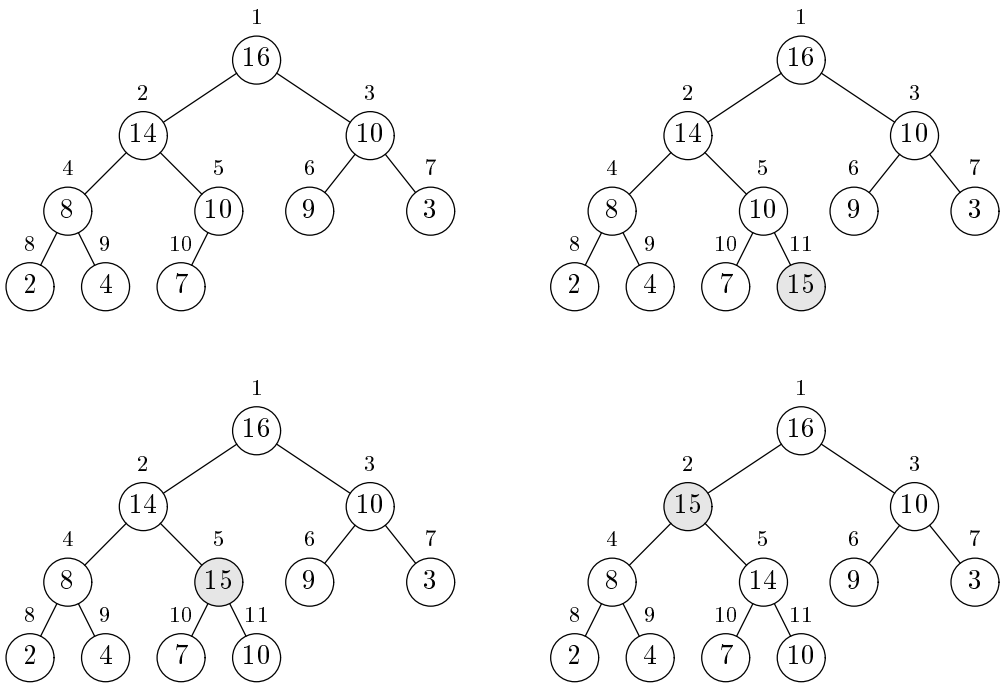


Figure 4.5 : Ajout dans un tas

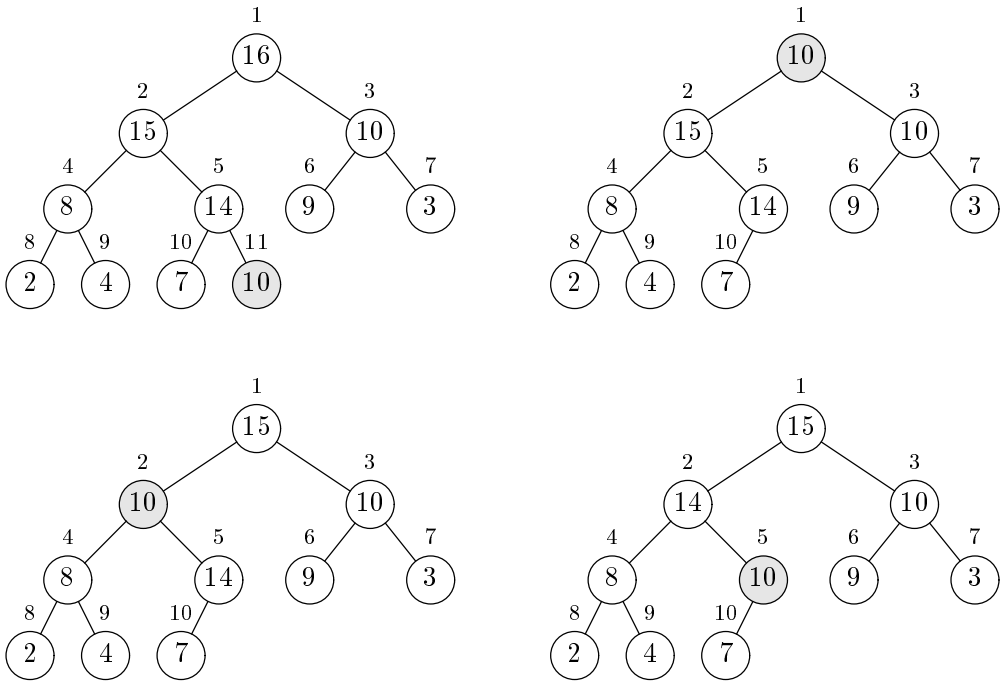


Figure 4.6 : Suppression dans un tas

```
            if j < nTas then if a[j + 1] > a[j] then j := j + 1;
            if v >= a[j] then goto 0;
            a[i] := a[j]; i := j;
            end;
      0:a[i] := v;
        end;
```

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 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

```
procedure HeapSort;
    var i: integer;
        v: Element;
    begin
    nTas := 0;
    for i := 1 to N do Ajouter (a[i]);
    for i := N downto 1 to
        begin
        v := Maximum;
        Supprimer;
        a[i] := v;
        end;
    end;
```

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

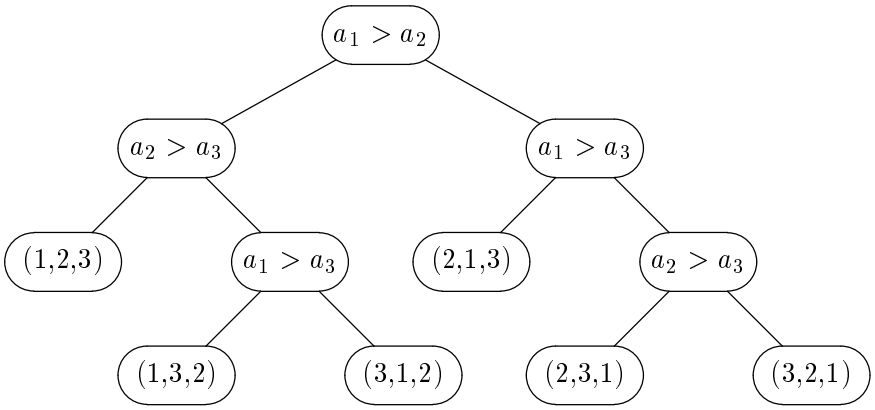


Figure 4.7 : Exemple d'arbre de décision pour le tri

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 nœud 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).*

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 nœud 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 nœud 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 nœud dont les fils gauche et droit ont une valeur vide.

```
type
    Arbre = ^Noeud;
    Noeud = record
        contenu: Element;
        filsG: Arbre;
        filsD: Arbre;
        end;
```

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 nœuds n-aires (ayant n fils). On doit considérer des types différents pour les nœuds 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)

```
type
    Arbre = ^Noeud;
    Noeud = record
        contenu: Element;
        fils: array [1..n] of Arbre;
        end;
```

la deuxième consiste à enchaîner les fils dans une liste

```
type
    Arbre = ^Noeud;
    ListeArbre = ^Cellule;
    Cellule = record
        contenu: Arbre;
        suivant: ListeArbre;
        end;
    Noeud = record
        contenu: Element;
        fils: ListeArbre;
        end;
```

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 nœud 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 nœud, les fils gauche et droit du nouveau nœud. Le résultat sera un pointeur vers ce nœud nouveau. Voici donc comment créer l'arbre de gauche de la figure 4.8.

```
var a5, a7: Arbre;

function NouvelArbre (v: Element; a, b: Arbre): Arbre;
    var c: Arbre;
    begin
    new (c);
    c^.contenu := v;
    c^.filsG := a;
    c^.filsD := b;
    NouvelArbre := c;
    end;

begin
a5 := NouvelArbre (12, NouvelArbre (8, NouvelArbre (6, nil, nil), nil)
                  NouvelArbre (13, nil, nil));
a7 := NouvelArbre (20, NouvelArbre (3, NouvelArbre (3, nil, nil), a5),
                   NouvelArbre (25, NouvelArbre (21, nil, nil),
                                    NouvelArbre (28, nil, nil)));
end.
```

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 nœuds 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.

```
procedure Imprimer (a: Arbre; tab: integer)
    var i: integer;
    begin
    if a <> nil then
        begin
        write (a^.contenu:3, '     ');
        Imprimer (a^.filsD, tab + 8);
        if a^.filsG <> nil then
            begin
            writeln;
```

```
            for i := 1 to tab do
                write (' ');
            end;
        Imprimer (a^.filsG, tab);
        end;
    end;

procedure ImprimerArbre (a: Arbre);
    begin
    Imprimer (a, 0);
    writeln;
    end;
```

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 66), 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 nœuds d'un arbre en suivant la définition récursive du type des arbres:

```
function Taille (a: Arbre): integer;
    begin
    if a = nil then                (* a = Arbre_vide *)
        Taille := 0
    else                           (* a ∈ Arbre × Element × Arbre *)
        Taille := 1 + Taille (a^.filsG) + Taille (a^.filsD);
    end;
```

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 nœuds unaires, la taille t, le nombre de feuilles N_f et le nombre de nœuds 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 nœuds du sous-arbre gauche d'un nœud ont une valeur inférieure (ou égale) à la sienne et tous les nœuds du sous-arbre droit ont une valeur supérieure (ou égale) à la valeur du nœud 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

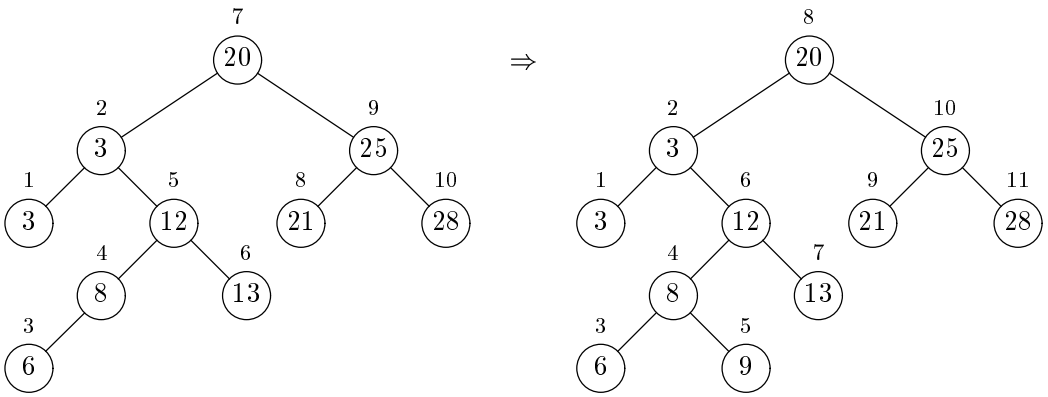


Figure 4.8 : Ajout dans un arbre de recherche

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.

```
function Recherche (v: Element; a: Arbre): Arbre;
    var r: Arbre;
    begin
    if a = nil then
        r := nil
    else if v = a^.contenu then
        r := a
    else if v < a^.contenu then
        r := Recherche (v, a^.filsG)
    else
        r := Recherche (v, a^.filsD)
    Recherche := r;
    end;

procedure Ajouter (v: Element; var a: Arbre);
    begin
    if a = nil then
        a := NouvelArbre (v, nil, nil)
    else if v <= a^.contenu then
        Ajouter (v, a^.filsG)
    else
        Ajouter (v, a^.filsD);
    end;
```

A nouveau, des programmes récursifs correspondent à la structure récursive des arbres. La procédure de recherche renvoie un pointeur vers le nœud contenant la valeur recherchée, `nil` 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 nœud. Dans le cas du bottin de téléphone, le champ `contenu` contiendrait les noms, et l'information serait le numéro de téléphone. Remarquons qu'alors le type `Element` serait une chaîne de caractères et les comparaisons correspondraient à l'ordre lexicographique.

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. On peut remarquer dans la procédure `Ajouter` le passage par référence de l'argument `a`, seule manière de modifier l'arbre.

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 nœuds, 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 nœuds dans un arbre de recherche est appelé *ordre infixe*. Il correspond au petit numéro qui se trouve au dessus de chaque nœud dans la figure 4.8. Nous avons déjà vu dans le cas de l'évaluation des expressions arithmétiques (cf page 71) *l'ordre préfixe*, dans lequel tout nœud reçoit un numéro d'ordre inférieur à celui de tous les nœuds de son sous-arbre de gauche, qui eux-mêmes ont des numéros inférieurs aux nœuds 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 nœud. 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 nœud ne peut excéder 1. Ainsi l'arbre de gauche de la figure 4.8 n'est pas équilibré. Il viole la propriété aux nœuds numérotés 2 et 7, tous les autres nœuds 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 nœuds est de l'ordre de $\log N$, ainsi les temps mis par la procédure `Recherche` vue page 99 seront en $O(\log N)$.

Il faut donc maintenir l'équilibre de tous les nœuds au fur et à mesure des opérations d'insertion ou de suppression d'un nœud dans un arbre AVL. Pour y arriver, on suppose que tout nœud 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 nœud, et on s'arrange donc pour maintenir $-1 \leq$ `a^.bal` ≤ 1 pour tout nœud 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 va-

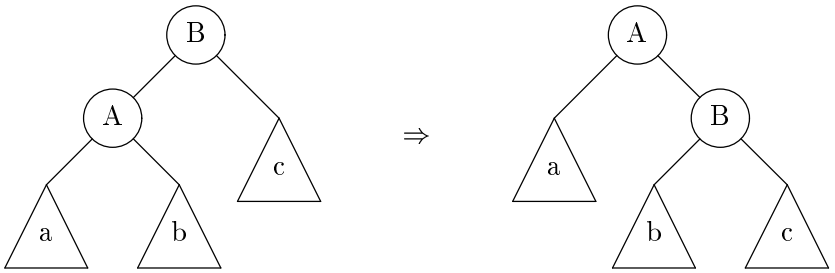


Figure 4.9 : Rotation dans un arbre AVL

leur 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 nœud 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 nœud A, suivie d'une rotation simple vers la droite du nœud 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

```
function Ajouter (v: Element; var a: Arbre): integer;
    var incr: integer;
    begin
    Ajouter := 0;
    if a = nil then begin
        a := NouveauNoeud (v, nil, nil);
        a^.bal := 0;
        Ajouter := 1;
        end
    else
        begin
        if v <= a^.contenu then incr := -Ajouter (v, a^.filsG)
                           else incr :=  Ajouter (v, a^.filsD);
        a^.bal := a^.bal + incr;
        if (incr <> 0) and (a^.bal <> 0) then
            if a^.bal < -1 then
                (* La gauche est trop grande *)
                if a^.filsG^.bal < 0 then RotD(a)
                else begin RotG(a^.filsG); RotD(a) end
            else
            if a^.bal > 1 then
                (* La droite est trop grande *)
                if a^.filsD^.bal > 0 then RotG(a)
                else begin RotD(a^.filsD); RotG(a) end
            else
                Ajouter : = 1;
```

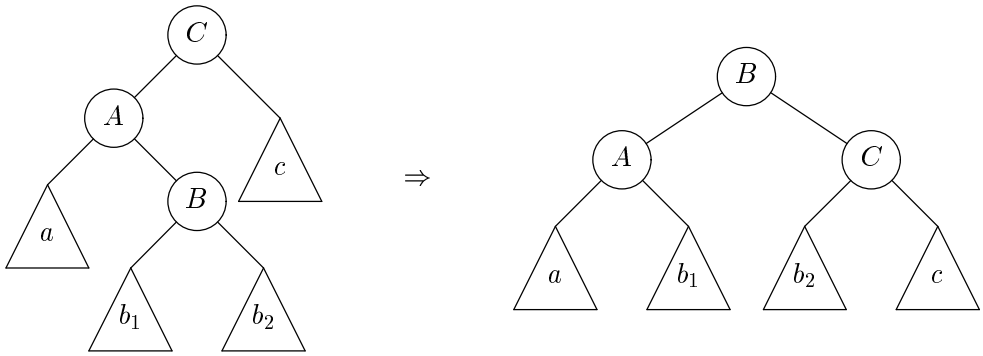


Figure 4.10 : Double rotation dans un arbre AVL

```
        end;
    end;
```

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.

```
procedure RotD (var a: Arbre);
    var b: Arbre;
    begin
    b := a;
    a := a^.filsG;
    b^.filsG := a^.filsD;
    a^.filsD := b;
    Recalculer le champ a^.bal
    end;
```

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 nœud 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 nœuds A et B se calculent en fonction des anciennes valeurs $b(A)$ et $b(B)$ par

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

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

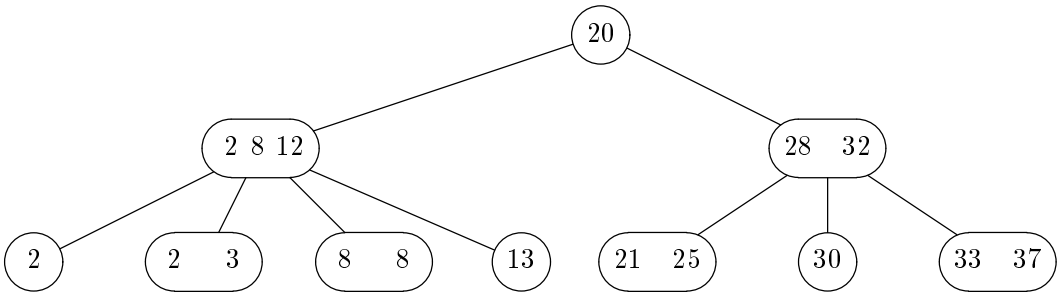


Figure 4.11 : Exemple d'arbre 2-3-4

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 nœuds. 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 [46]. 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 nœuds binaires: tout nœud 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 nœuds 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 nœuds à 4 fils.

L'insertion d'un nouvel élément dans un arbre 2-3-4 se fait en éclatant tout nœud 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 nœuds. Ainsi, on garantit que l'endroit où on insère la nouvelle valeur n'est pas un nœud quaternaire, et il suffit de mettre la valeur dans ce nœud à l'endroit désiré. (Si la racine est quaternaire, on l'éclate en 3 nœuds binaires). Le nombre d'éclatements maximum peut être $\log N$ pour un arbre de N nœuds. 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 nœud pour dire si la branche le reliant à son père est rouge ou noire. Les nœuds quaternaires sont représentés par 3 nœuds reliés en noir. Les nœuds 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 C

```
typedef    int Element;
```

```
int        nTas = 0;

void Ajouter (int v)              /* Ajouter à un tas, voir page 92 */
{
    int    i;

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

```
Element Maximum ()          /* Maximum d'un tas, voir page 92 */
{
    return a[0];
}
```

```
void Supprimer ()          /* Supprimer dans un tas, voir page 92 */
{
    int i, j;
    Element 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;
}
```

```
void HeapSort ()                   /* HeapSort, voir page 94 */
{
    int      i;
    Element  v;

    nTas = 0;
    for (i = 0; i < N; ++i) {
        Ajouter (a[i]);
        Impression();
    }
```

```
    for (i = N - 1; i >= 0; --i) {
        v = Maximum();
        Supprimer();
        a[i] = v;
    }
}
```

```
typedef struct Noeud {          /* Déclaration d'un arbre, voir page 96 */
    Element         contenu;
    struct Noeud    *filsG;
    struct Noeud    *filsD;
}  *Arbre;
```

```
typedef struct Noeud {          /* Déclaration d'un arbre, voir page 96 */
    Element         contenu;
    struct Noeud    *fils[n];
} *Arbre;
```

```
typedef struct  Cellule {           /* Déclaration d'un arbre, voir page 96 */
    struct Noeud         *contenu;
    struct Cellule       *suivant;
} *ListeArbre;

typedef struct Noeud {
    Element     contenu;
    ListeArbre  fils;
} *Arbre;
```

```
Arbre a5, a7;                    /* Ajouter dans un arbre, voir page 96 */

Arbre NouvelArbre (Element v, Arbre a, Arbre b)
{
    Arbre     c;

    c = (Arbre) malloc (sizeof (struct Noeud));
    c -> contenu = v;
    c -> filsG = a;
    c -> filsD = b;
    return c;
}

int main()
{
    a5 = NouvelArbre (12, NouvelArbre (8, NouvelArbre (6, NULL, NULL), NULL),
                          NouvelArbre (13, NULL, NULL));
    a7 = NouvelArbre (20, NouvelArbre (3, NouvelArbre (3, NULL, NULL), a5),
                          NouvelArbre (25, NouvelArbre (21, NULL, NULL),
                                           NouvelArbre (28, NULL, NULL)));
    ...
```

```
}
```

```
void Imprimer (Arbre a, int tab)  /* Impression d'un arbre, voir page 97 */
{
    int i;

    if (a != NULL) {
        printf ("%3d     ", a -> contenu);
        Imprimer (a -> filsD, tab + 8);
        if (a -> filsG != NULL) {
            putchar ('\n');
            for (i = 1; i <= tab; ++i)
                putchar (' ');
        }
        Imprimer (a -> filsG, tab);
    }
}

void ImprimerArbre (Arbre a)
{
    Imprimer (a, 0);
    putchar ('\n');
}
```

```
int Taille (Arbre a)   /* Taille d'un arbre, voir page 98 */
{
    if (a == NULL)
        return 0;
    else
        return 1 + Taille (a -> filsG) + Taille (a -> filsD);
}
```

```
Arbre Recherche (Element v, Arbre a)   /* Arbre de recherche, voir page 99 */
{
    Arbre      r;

    r = NULL;
    if (a == NULL || v == a -> contenu)
         return a;
    else
         if (v < a -> contenu)
             return Recherche (v, a -> filsG);
         else
             return Recherche (v, a -> filsD);
}

void Ajouter (Element v, Arbre *ap)
{
    Arbre  a = *ap;
    if (a == NULL)
```

```
        a = NouvelArbre (v, NULL, NULL);
    else if (v <= a -> contenu)
        Ajouter (v, &a -> filsG);
    else
        Ajouter (v, &a -> filsD);
    *ap = a;
}
```

```
int Ajouter (Element v, Arbre *ap)      /* Ajout dans un AVL, voir page 101 */
{
    int        incr, r;
    Arbre      a = *ap;
    void       RotD(Arbre *), RotG(Arbre *);

    r = 0;
    if (a == NULL) {
        a = NouvelArbre (v, NULL, NULL);
        a -> bal = 0;
        r = 1;
    } else {
        if (v <= a -> contenu)
            incr = -Ajouter (v, &a -> filsG);
        else
            incr =  Ajouter (v, &a -> filsD);
        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)
                    RotD (&a);
                else {
                    RotG (&a -> filsG);
                    RotD (&a);
                }
            else
            if (a -> bal > 1)
                /* La droite est trop grande */
                if (a -> filsD -> bal > 0)
                    RotG (&a);
                else {
                    RotD (&a -> filsD);
                    RotG (&a);
                }
            else
               r = 1;
    }
    *ap = a;
    return r;
}
```

```
#define Min(x, y)      ((x) <= (y)? (x) : (y))
#define Max(x, y)      ((x) >= (y)? (x) : (y))

void RotD (Arbre *ap)              /* Rotation dans un AVL, voir page 102 */
{
    Arbre     a, b;
    int       bA, bB, bAnew, bBnew;

    a = *ap;
    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 - Min(0, bA);
    bAnew = 1 + bA + Max(0, bBnew);
    a -> bal = bAnew;
    b -> bal = bBnew;
    *ap = a;
}

void RotG (Arbre *ap)              /* Rotation dans un AVL, voir page 102 */
{
    Arbre     a, b;
    int       bA, bB, bAnew, bBnew;

    a = *ap;
    b = a -> filsD;
    bA = a -> bal; bB = b -> bal;
    a -> filsD = b -> filsG;
    b -> filsG = a;
    /* Recalculer le champ a -> bal */
    bAnew = bA - 1 - Max(0, bB);
    bBnew = bB - 1 + Min(0, bAnew);
    a -> bal = bAnew;
    b -> bal = bBnew;
    *ap = b;
}
```

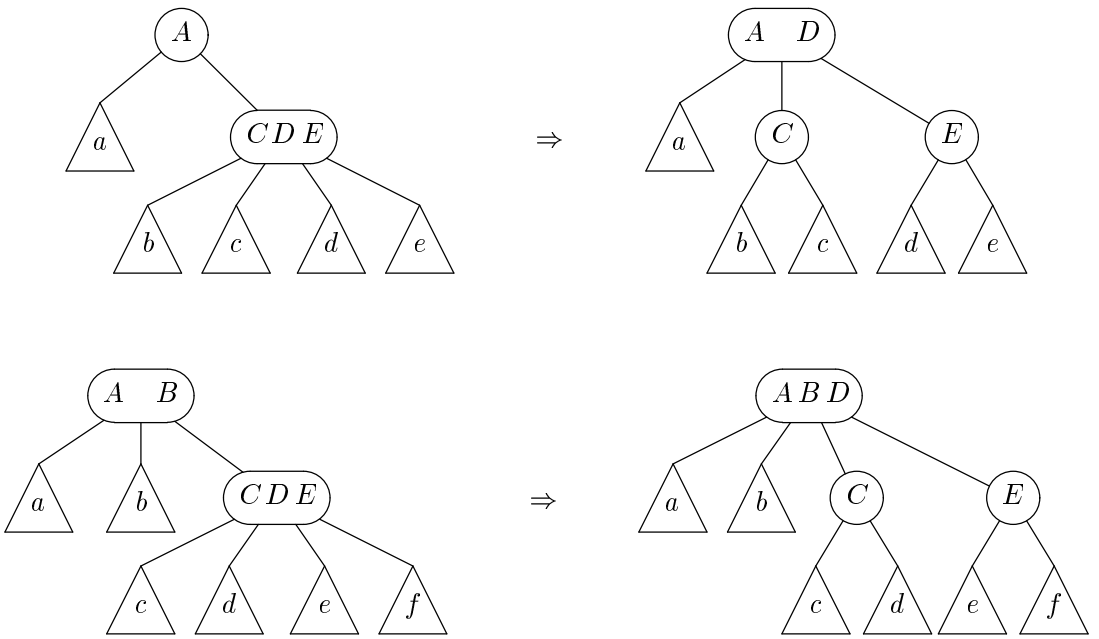


Figure 4.12 : Eclatement d'arbres 2-3-4

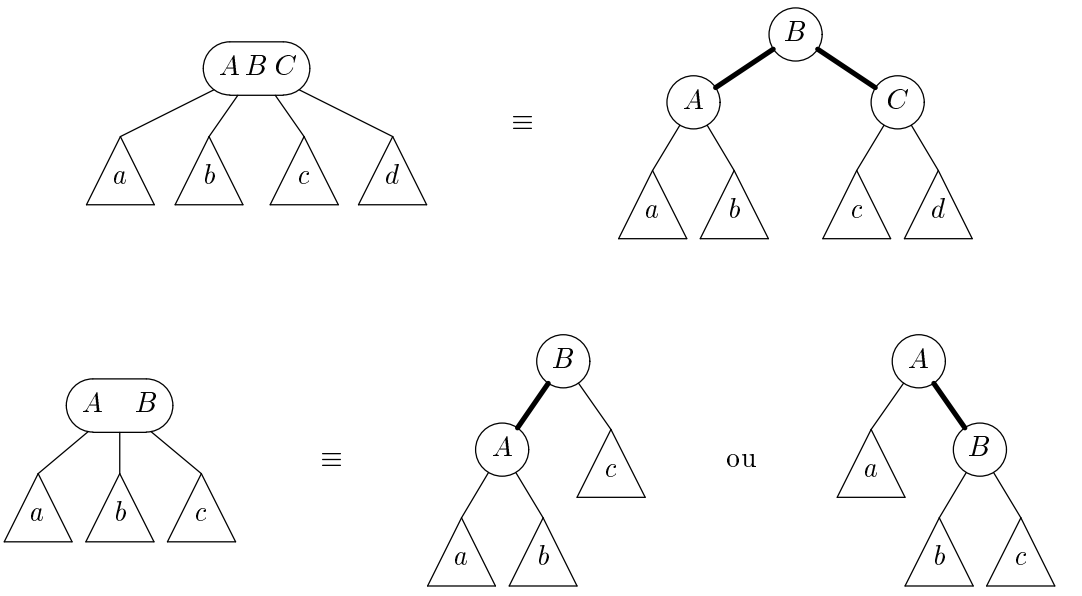


Figure 4.13 : Arbres bicolores

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.*

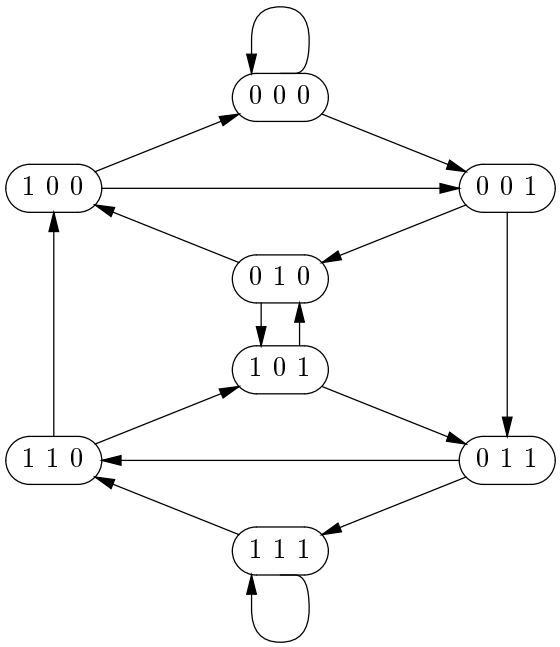


Figure 5.1 : Le graphe de De Bruijn pour $k = 3$

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:*

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

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.

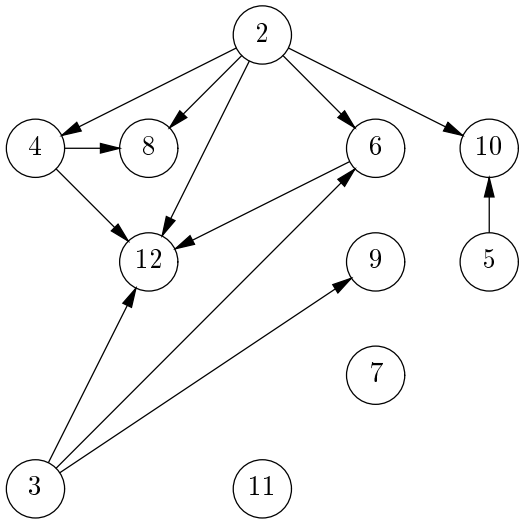


Figure 5.2 : Le graphe des diviseurs, $n = 12$

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 cœfficients sont 0 et 1 telle que:

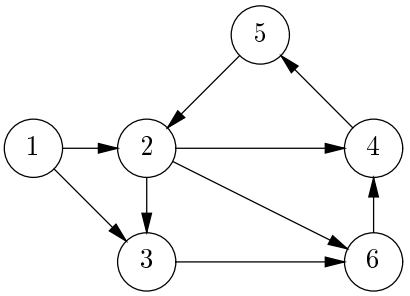
$$M_{i,j} = 1 \quad \text{si} \quad (x_i, x_j) \in A, \quad M_{i,j} = 0 \quad \text{si} \quad (x_i, x_j) \notin A$$

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

```
const
    Nmax = 50;    (* Nombre de sommets maximal pour un graphe *)
type
    IntSom = 1..Nmax;
    GrapheMat = array[IntSom, IntSom] of integer;
var
    m: GrapheMat;
    n: IntSom;   (* n est le nombre effectif de sommets *)
                 (* du graphe dont la matrice est M *)
```

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.*



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

Figure 5.3 : Un exemple de graphe et sa matrice d'adjacence

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:

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

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:

```
function ExisteChemin (i, j: IntSom; n: integer; m: GrapheMat): boolean;
    var
        k: integer;
        u, v, w: GrapheMat;
    begin
    u := m;
    for k := 1 to n do
        begin
        Multiplier(n, u, m, v);
        Ajouter(n, u, v, w);
        u := w;
        end;
    ExisteChemin := (u[i, j] <> 0);
    end;
```

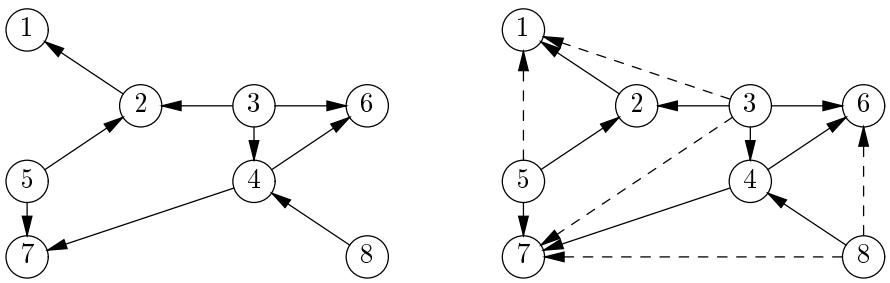


Figure 5.4 : Un graphe et sa fermeture transitive

Dans cet algorithme, les procédures `Multiplier(n, u, v, w)` et `Ajouter(n, u, v, w)` sont respectivement des procédures qui multiplient et ajoutent les deux matrices $n \times n$ `u` et `v` pour obtenir la matrice `w`.

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 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 $G^* = (X, *)$ tel que $(x, y) \in *$ 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

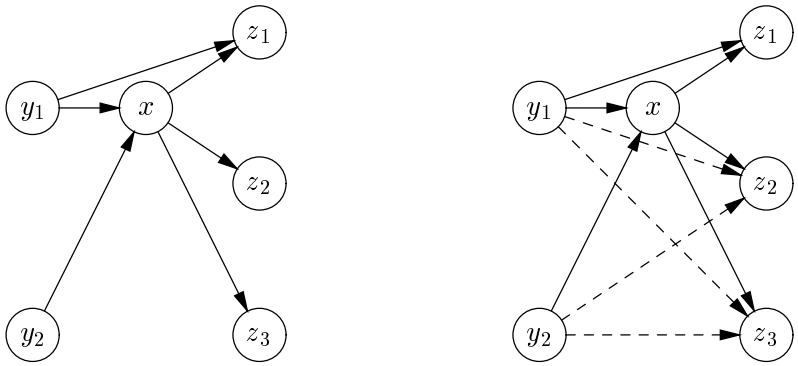


Figure 5.5 : L'effet de l'opération Φ_x: les arcs ajoutés sont en pointillé

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 $(X, *)$, la réponse à chaque question est alors immédiate par simple consultation d'un des cœfficients 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 en machine pour l'exécution du programme.

Le calcul de $(X, *)$ 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 $(u, v) \in \Phi_y(\Phi_x(A))$ □

Proposition 2 *La fermeture transitive $*$ est donnée par :*

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

Preuve On se convainc facilement que $*$ 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 $*$. 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:

```
procedure PHI (var m: GrapheMat; x: IntSom; n: integer);
    var
        u, v: IntSom;
    begin
    for u := 1 to n do
        if (m[u, x] = 1) then
            for v := 1 to n do
                if (m[x, v] = 1) then
                    m[u, v] := 1;
    end;

procedure FermetureTransitive (var m: GrapheMat; n: integer);
    var
        x: IntSom;
    begin
    for x := 1 to n do
        PHI(m, x, n);
    end;
```

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:

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

Les déclarations Pascal correspondantes peuvent être alors les suivantes:

```
const
    Nmax = 50;    (* Nombre maximal de sommets pour un graphe *)
    MaxDeg = 30;  (* Nombre maximal de successeurs pour un sommet *)
    Omega = -1;   (* Quantité différente des valeurs de sommets *)
type
    IntSom = 1..Nmax;
    Sommet = integer;
    GrapheSuc = array[IntSom, 1..MaxDeg] of Sommet;
var
    succ: GrapheSuc;
    n: integer;
```

Le parcours de la liste des successeurs d'un sommet i 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 :

```
k := 1;
j := succ[i,k];
while (j <> Omega) do
    begin
    Traiter(j); (* Traitement du sommet j *)
    k := k + 1;
    j := succ[i,k];
```

```
        end;
```

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

```
procedure TransformMatSuc (m: GrapheMat; n: integer; var succ: GrapheSuc);
    var
        k: integer;
        i, j: IntSom;
    begin
    for i := 1 to n do
        begin
        k := 1;
        for j := 1 to n do
            if (m[i, j] = 1) then
                begin
                succ[i, k] := j;
                k := k + 1
                end;
            succ[i, k] := Omega
        end;
    end;
```

Remarque La structure de liste de successeurs peut être remplacée par une structure de liste chaînée faisant intervenir des pointeurs dans le langage Pascal. 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 :

```
type
    ListeSom = ^SomCell;
    SomCell = record
        val: Sommet;
        suiv: ListeSom;
        end;
    GraphePoint = array[IntSom] of ListeSom;
```

La transformation de la forme matricielle *Succ* en une structure de liste chaînée par des pointeurs se fait par l'algorithme donné ci dessous, on peut noter que celui-ci inverse l'ordre dans lequel ont rangés les successeurs d'un sommet, ceci n'a pas d'importance dans la plupart des cas:

```
procedure TransformSucPoint (succ: GrapheSuc; n: integer;
                             var gpoint: GraphePoint);
    var
        i: integer;
        x: Sommet;
        s: ListeSom;
    begin
```

```
        for x := 1 to n do
            begin
            gpoint[x] := nil;
            i := 1;
            while (succ[x, i] <> Omega) do
                begin
                new(s);
                s^.val := succ[x, i];
                s^.suiv := gpoint[x];
                gpoint[x] := s;
                i := i + 1;
                end;
            end;
    end;
```

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,*

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

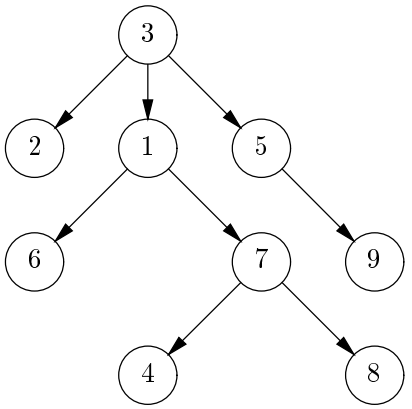
tels que:

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

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

Figure 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 en Pascal on obtient:

```
type
    Arbo = array[IntSom] of Sommet;
    procedure SuccEnPere (succ: GrapheSuc; n: integer; r: Sommet;
                          var pere: Arbo);
        var
            k: integer;
            i, j: Sommet;
    begin
    pere[r] := r;
    for i := 1 to n do
        begin
        k := 1;
        j := succ[i, k];
        while (j <> Omega) do
            begin
            pere[j] := i;
            k := k + 1;
            j := succ[i, k];
            end;
        end;
    end;
```

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.

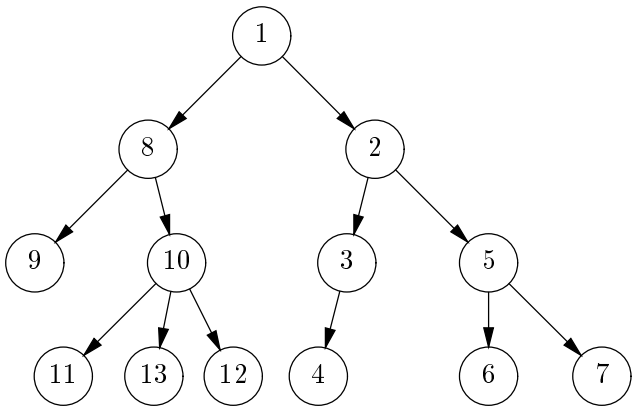


Figure 5.7 : Une arborescence préfixe

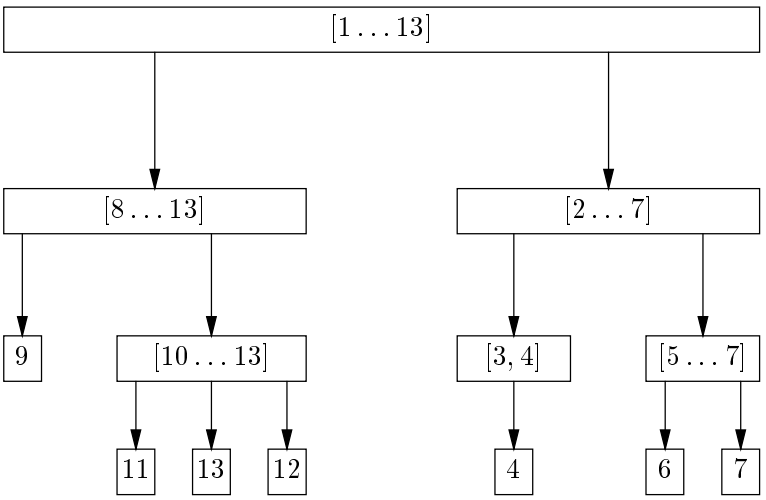


Figure 5.8 : Emboitement des descendants dans une arborescence préfixe

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.
- 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. □

L'algorithme qui est décrit dans la preuve ci-dessus peut s'écrire simplement en Pascal, 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.

```
var
    numero: SomVect;
    succ: GrapheSuc;

procedure Numprefixe (x: Sommet; var num: integer);
    var
       i: integer;
       y: IntSom;
    begin
    numero[x] := num;
    num := num + 1;
    i := 1;
    y := succ[x, i];
    while (y <> Omega) do
        begin
        Numprefixe(y, num);
        i := i + 1;
        y := succ[x, i];
        end;
    end;
num := 1;
Numprefixe(r,num);
```

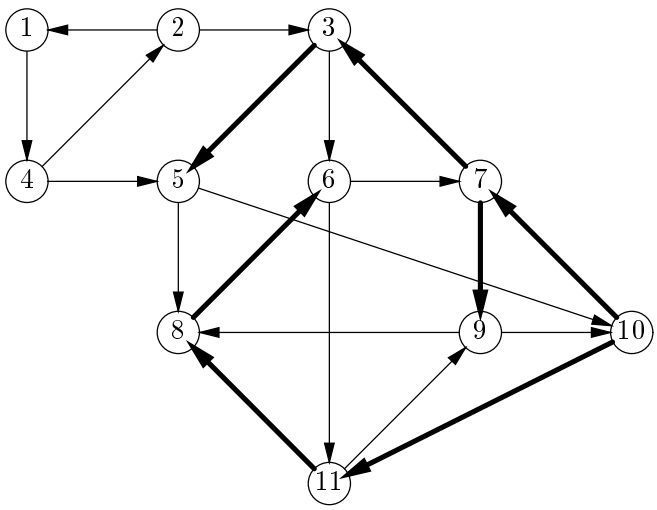


Figure 5.9 : Une arborescence des plus courts chemins de racine 10

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.*

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)`.

```
procedure ArbPlusCourt (succ: GrapheSuc; n: integer; x: Sommet;
                        var pere: Arbo);
    var
        f: Fil;
        u, v: Sommet;
        i: integer;
    begin
    InitialiserFile(f);
    for u := 1 to n do
        pere[u] := Omega;
    Fajouter(x, f);
    pere[x] := x;
    while not (Fvide(f)) do
        begin
        u := Fvaleur(f);
        Fsupprimer(f);
        i := 1;
        v := succ[u, i];
        while (v <> Omega) do
            begin
            if (pere[v] = Omega) then
                begin
                pere[v] := u;
                Fajouter(v, f);
                end;
```

```
            i := i + 1;
            v := succ[u, i];
            end;
        end;
    end;
```

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.

```
procedure TremauxRec (u: Sommet; var pere: Arbo);
   var
       k: integer;
       v: Sommet;
   begin
   k := 1;
   v := succ[u, k];
   while (v <> Omega) do
       begin
       if (pere[v] = Omega) then
           begin
           pere[v] := u;
           TremauxRec(v, pere);
           end;
       k := k + 1;
       v := succ[u, k];
       end;
    end;
```

Le calcul effectif de l'arborescence de Trémaux de racine x s'effectue en initialisant le vecteur `pere` et en effectuant l'appel de `TremauxRec(x,pere)`:

```
for i := 1 to n do
    pere[i] := Omega;
pere[x] := x;
```

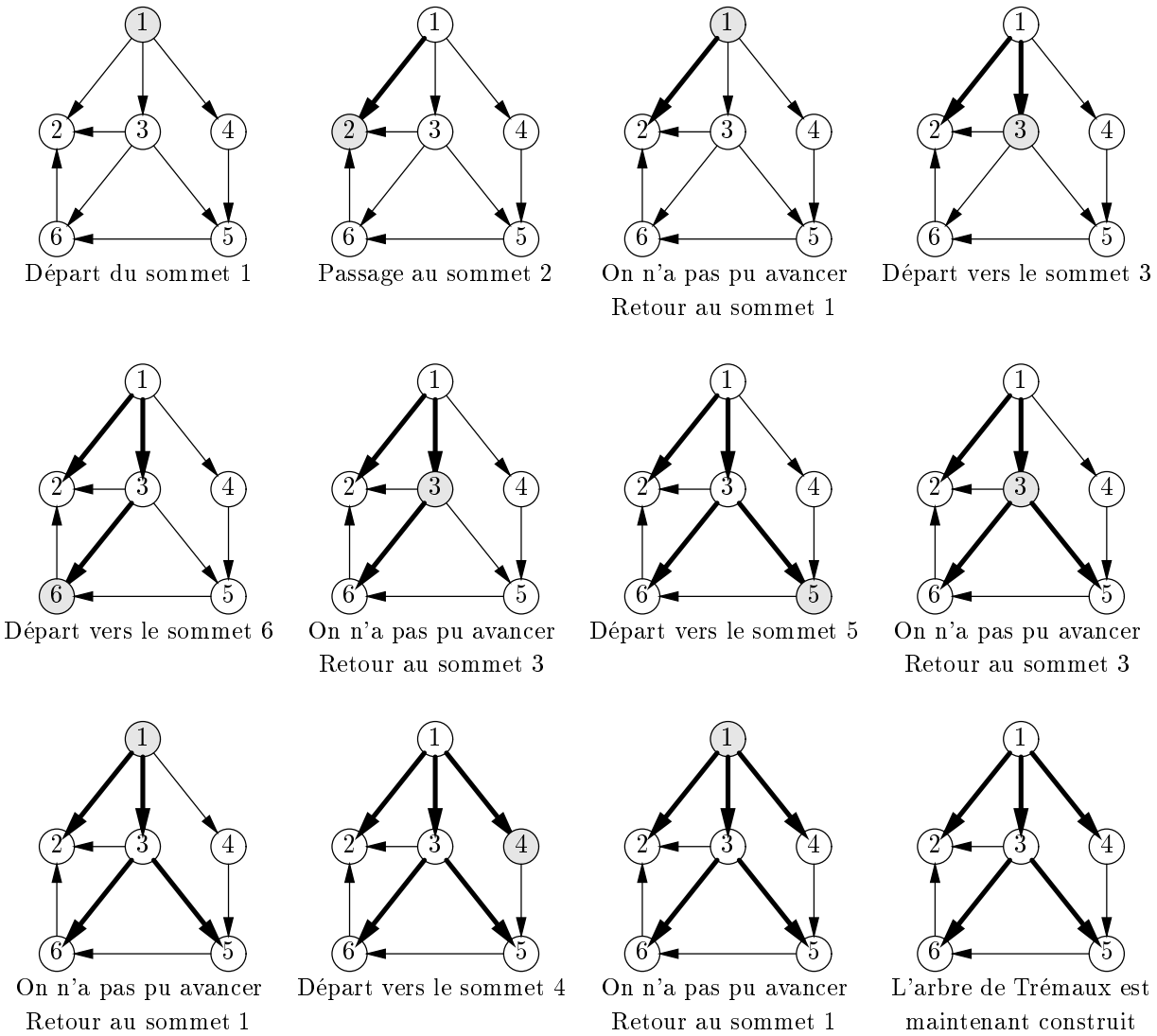


Figure 5.10 : Exécution de l'algorithme de Trémaux

```
TremauxRec(x,pere);
```

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.

```
procedure TremauxPil (succ: GrapheSuc; n: integer; x: Sommet;
                      var pere: Arbo);
    label 999;
    var
        p: Pile;
        i, u, v: Sommet;
        j: integer;
    begin
    for i := 1 to n do
        pere[i] := Omega;
    Pinitialiser(p);
    Pajouter(x, p);
    pere[x] := x;
    while not (Pvide(p)) do
        begin
        u := Pvaleur(p);
        j := 1;
        v := succ[u, j];
        if v <> Omega then
            while  pere[v] <> Omega do
                begin
                j := j + 1;
                v := succ[u, j];
                if v= Omega then goto 999;
                end;
        999:
        if (v <> Omega) then
            begin
            pere[v] := u;
            Pajouter(v, p);
            end
        else
            Psupprimer(p);
        end;
    end;
```

Remarques

1 L'ensemble des sommets atteignables à partir du sommet x est formé des sommets

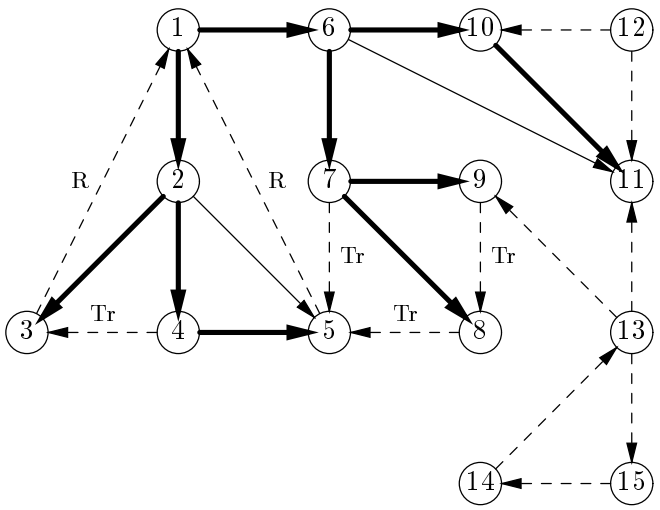


Figure 5.11 : Les arcs obtenus par Trémaux

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.

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

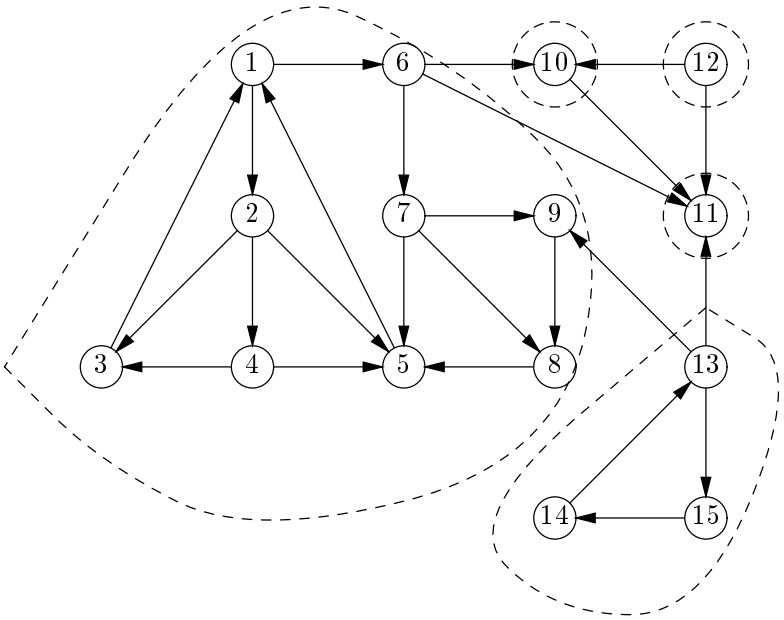


Figure 5.12 : Composantes fortement connexes du graphe de la figure 5.11

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 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.

```
var
    numero: SomVect;
    nu: integer;
procedure TremauxRec (u: Sommet; var pere: Arbo);
    var
        k: integer;
        v: Sommet;
    begin
    nu := nu + 1;
    numero[u] := nu;
    k := 1;
    v := succ[u, k];
    while (v <> Omega) do
        begin
        if (pere[v] = Omega) then
```

```
                begin
                pere[v] := u;
                TremauxRec(v, pere);
                end;
            k := k + 1;
            v := succ[u, k];
            end;
        end;

    for i:=1 to n do
        pere[i] := Omega;
    pere[x] := x;
    nu := 0;
    TremauxRec (x,pere);
```

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).*

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

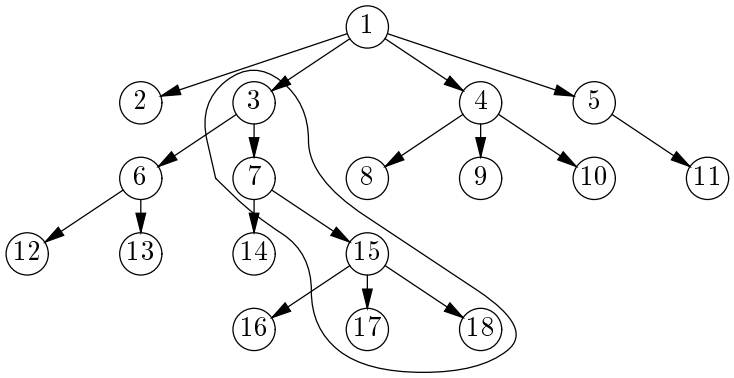


Figure 5.13 : Un exemple de sous-arborescence

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 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.

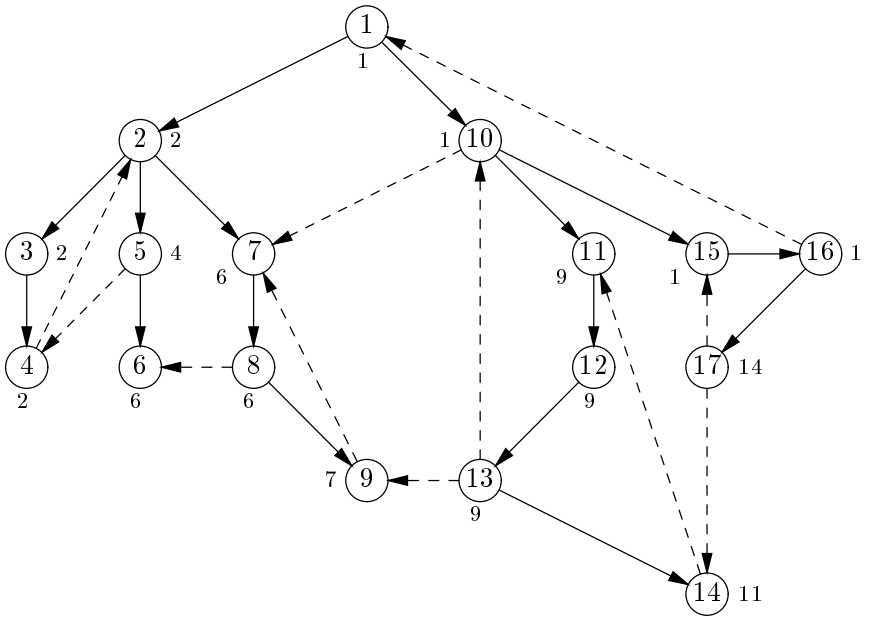


Figure 5.14 : Les points d'attaches des sommets d'un graphe

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`.

```
var
    at: SomVect;
    succ: GrapheSuc;

function PointAttache (u: Sommet): Sommet;
    var
        k: integer;
        v, w, mi: Sommet;
    begin
    k := 1;
```

```
        v := succ[u, k];
        mi := u;
        at[u] := u;
        while (v <> Omega) do
            begin
            if (at[v] = Omega) then
                w := PointAttache(v)
            else
                w := v;
            mi := Min(mi, w);
            k := k + 1;
            v := succ[u, k];
            end;
        at[u] := mi;
        PointAttache := mi
        end;

    for i := 1 to n do
        at[i] := Omega;
    at[x] := PointAttache(x);
```

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 $v_2 \in desc(u)$. □

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, en Pascal, 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:

```
var val:array[sommet] of integer;
    succ: GrapheSuc;
    n:integer;

procedure Supprimer (u: Sommet;  nu: integer;
                     var numComp: SomVect);
     (*La suppression d'un sommet s'effectue *)
     (*en lui donnant un numéro de composante différent de 0 *)
    var
       v: Sommet;
       k: integer;
    begin
    numComp[u] := nu;
    k := 1;
    v := succ[u, k];
    while v <> Omega do
        begin
```

```
        if (v > u) and (numComp[v] = 0) then
            Supprimer(v, nu, numComp);
        k := k + 1;
        v := succ[u, k];
        end;
    end;

procedure PointAttache1 (u: Sommet;  var nu: integer;
                         var at, numComp: Somvect);
    var
        k: integer;
        v, w: Sommet;
    begin
    at[u] := u;       (*Recherche du point d'attache de u*)
    k := 1;           (*Par la même methode que précédemment*)
    v := succ[u, k];
    while (v <> Omega) do
        begin
        if (at[v] = Omega) then
            begin
            PointAttache1(v, nu, at, numComp);
            at[u] := Min(at[u], at[v]) (*C'est une procédure, pas une fonction *)
            end
        else if numComp[u] = 0 then
            at[u] := Min(at[u], v);
        k := k + 1;
        v := succ[u, k];
        end;
      (* Lorsqu'on trouve u = at(u), aucun descendant ne vérifie cette relation. *)
      (* On a donc trouvé une composante qu'il faut supprimer *)
        if u = at[u] then
            begin
            nu := nu + 1;
            Supprimer(u, nu, numComp);
            end;
    end;

procedure CompCon (var numComp: SomVect);
    var
        k, mi, num: integer;
        at: somvect;
        u, v, w: Sommet;
    begin
    for u := 1 to n do
        begin
        numComp[u] := 0;
        at[u] := Omega;
        end;
    u := 1;
    num := 0;
    for u := 1 to n do
```

```
        if ((numComp[u]) = 0 and (at[u] = Omega)) then
            PointAttache1(u, num, at, numComp)
    end;
```

5.9 Programmes en C

```
/* Graphes par matrice et existence de chemins, voir page 113 et 114 */

#define Nmax 100

typedef int Sommet ;

typedef Sommet GrapheMat[Nmax][Nmax];

int ExisteChemin (int i,int j, int n, GrapheMat g)
{
    int       k;
    GrapheMat x,y;

    Copy(g, x, n);
    k = 1;
    while (x[i][j] == 0 && k <= n){
        ++ k;
        Produit(n, x, g, y);
        Copy(y, x, n);
    }
    return (x[i][j] != 0);
}
```

```
void Phi (GrapheMat g, int n, int x)
{                          /* calcul de la Fermeture transitive , voir page 117 */
    int     u,v;

    for(u = 1; u <= n; ++u)
        if (g[u][x] == 1)
            for (v = 1;v <= n; ++v)
               if (g[x][v] == 1)
                   g[u][v] = 1;
}

void FermetureTransitive (GrapheMat g, int n)
{
    int     x;

    for (x = 1; x <= n; ++x)
        Phi(g, n, x);
}
```

```
#define Sucmax   50     /* Représentation par tableau de successeurs voir page 118 */
#define Omega    -1
```

```
typedef Sommet GrapheSuc[Nmax][Sucmax];

void TransformMatSuc (GrapheMat g, int n, GrapheSuc succ)
{
    int     i,j,k;

    for (i = 1; i <= n; ++i){
        k = 1;
        for (j = 1; j <= n; ++j)
            if (g[i][j] == 1){
                succ[i][k] = j;
                ++k;
            }
        succ[i][k] = Omega;
    }
}
```

```
struct Cellule {          /* Représentation par des listes voir page 119 */
    Sommet          contenu;
    struct Cellule  *suivant;
};

typedef struct Cellule   Cellule;
typedef struct Cellule   *Liste;
typedef Liste            GraphePoint[Nmax] ;

void TransformSucPoint (GrapheSuc succ, int n, GraphePoint gpoint)
{
    int     i;
    Liste   a;
    Sommet  x;

    for (x = 1; x <= n; ++x) {
        gpoint[x] = NULL;
        i = 1;
        while (succ[x][i] != Omega) {
            a = (Liste) malloc(sizeof(Cellule));
            a -> contenu = succ[x][i];
            a -> suivant = gpoint[x];
            gpoint[x] = a;
            ++ i;
        }
    }
}
```

```
/* Arborescence et vecteur pere voir page 121 et
 * numérotation préfixe voir page 123
*/

typedef Sommet   Arbo[Nmax];
```

```
void SuccEnPere (GrapheSuc succ, int n, Sommet r, Arbo pere)
{
    int     k;
    Sommet  i,j;

    pere[r] = r;
    for (i = 1; i <= n; ++i){
        k = 1;
        j = succ[i][k];
        while (j != Omega){
            pere[j] = i;
            ++k;
            j = succ[i][k];
        }
    }
}

void Numprefixe (Sommet x, GrapheSuc succ, SomVect numero, int *num)
{
    int      i;
    Sommet   y;

    numero[x] = *num;
    ++ *num;
    for(i = 1; succ[x][i] != Omega; ++i)
        Numprefixe(succ[x][i], succ, numero, num);
}
```

```
void ArbPlusCourt (GrapheSuc succ, int n, Sommet x, Arbo pere)
{                          /* Arborescence des plus courts chemins, voir page 125 */
    Fil      f;
    Sommet   u,v;
    int      i;

    FaireFil(&f);
    for (u = 1; u <= n; ++u)
        pere[u] = Omega;
    Fajouter(x, &f);
    pere[x] = x;
    u = Fvaleur(&f);
    while ( !Fvide (&f) ) {
        u = Fvaleur(&f);
        Fsupprimer (&f);
        for(i = 1; succ[u][i] != Omega; ++i){
            v = succ[u][i];
            if (pere[v] == Omega){
                pere[v] = u;
                Fajouter(v,&f);
            }
        }
    }
}
```

```
void TremauxRec (Sommet u, Arbo pere, GrapheSuc succ, int n)
{                          /* Procédure de Trémaux Récursive voir page 126  */
    int    k;
    Sommet v;

    for (k = 1; succ[u][k] != Omega; ++k) {
        v = succ[u][k];
        if (pere[v] == Omega) {
            pere[v] = u;
            TremauxRec(v, pere, succ, n);
        }
    }
}
```

```
void TremauxPil (GrapheSuc succ, int n, Sommet x, Arbo pere)
{                   /* Procédure de Trémaux Itérative voir page 128  */

    Pile     p;
    Sommet   i,u,v;
    int      j;

    for (i = 1; i <= n; ++i)
         pere[i] = Omega;
    FairePile (&p);
    Pajouter (x, &p);
    pere[x] = x;
    while ( !Pvide(&p) ) {
        u = Pvaleur (&p);
        j = 1;
        v =succ[u][j];
        while (v != Omega && pere[v] != Omega) {
            ++j;
            v = succ[u][j];
        };
        if (v != Omega) {
            pere[v] = u;
            Pajouter (v, &p);
        } else
            Psupprimer (&p);

    }
}
```

```
typedef int    SomVect[Nmax];       /* calcul des points d'attache, voir page 134 */

Sommet PointAttache(Sommet u, GrapheSuc succ, int n, SomVect at)
{
    int      k;
    Sommet   v, w, mi;
```

```
    mi = u;
    at[u] = u;
    for (k = 1; succ[u][k] != Omega; ++k) {
        v = succ[u][k];
        if (at[v] == Omega)
            w = PointAttache(v, succ, n, at);
        else
            w = v;
        if (w < mi)
            mi = w;
    }
    at[u] = mi;
    return mi;
}
```

```
/*Détermination des composantes fortement connexes voir page 136 */
void Supprimer (Sommet u, int nu, SomVect numComp,
                                GrapheSuc succ, int n)
{
    Sommet   v;
    int      k;

    numComp[u] = nu;
    for (k = 1; succ[u][k] != Omega; ++k){
        v = succ[u][k];
        if (v > u && numComp[v] == 0)
            Supprimer(v, nu, numComp, succ, n);
    }
}

void PointAttache1(Sommet u, GrapheSuc succ, int n,
                   SomVect at, SomVect numComp, int *nu)
{
    int    k;
    Sommet v,w,mi;

    at[u] = u;
    for (k = 1; succ[u][k] != Omega; k++){
        v = succ[u][k];
        if (at[v] == Omega){
            PointAttache1 (v, succ, n, at, numComp, nu);
            at[u] = min (at[u], at[v]);
        }else
            if (numComp[v] == 0)
                 at[u] = minimum (at[u], v);
     }
     if (u == at[u]) {
          ++ *nu;
          Supprimer(u, *nu, numComp, succ, n);
      }
}
```

```
void CompCon (GrapheSuc succ, int n,  SomVect numComp)
{
    Sommet   u;
    int      num;
    SomVect  at;

    for (u = 1; u <= n; ++u){
        numComp[u] = 0;
        at[u] = Omega;
    }

    num = 0;
    for (u = 1; u <= n ; ++u)
        if (numComp[u] == 0 && at[u] == Omega)
            PointAttache1(u,  succ, n, at, numComp, &num);
}
```

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 nœud 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 [32] 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 [33].

On utilise un ensemble fini appelé *alphabet* A dont les éléments sont appelés des *lettres*. Un *mot* est une suite finie $f = a_1 a_2 \dots a_n$ 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: $A = \{\texttt{begin}, \texttt{end}, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{while}, \texttt{do}, ;, \texttt{p}, \texttt{q}, \texttt{x}, \texttt{y}, \texttt{z}\}$

   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 $u = u_1 T u_2$. On choisit une règle de la forme $T \to v$. On remplace u par $u' = u_1 v u_2$. On répète l'opération sur u' et ainsi de suite jusqu'à obtenir un mot qui ne contient que des lettres de A.

Dans la mesure où il y a plusieurs choix possibles à chaque étape on voit que le nombre de mots engendrés par une grammaire est souvent infini. Mais certaines grammaires peuvent n'engendrer aucun mot. C'est le cas par exemple des grammaires dans lesquelles tous les membres droits des règles contiennent un lettre de 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 $v = w_1 S w_2$ et $u = w_1 w w_2$. On dit aussi que v se dérive directement en u. On dit que u *dérive de* v, ou que v se dérive en u, si u s'obtient à partir de v par une suite finie de dérivations directes. On note alors:

$$v \xrightarrow{*} u$$

Ce qui signifie l'existence de w_0, w_1, ... w_n, $n \geq 0$ tels que $w_0 = v$, $w_n = u$ et pour tout $i = 1, \ldots n$, 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 $S \rightarrow aTbS$, puis appliquons la règle $T \rightarrow \epsilon$. On obtient:

$$S \rightarrow aTbS \rightarrow abS$$

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

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

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

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

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

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:

$$S \rightarrow aSbS \qquad\qquad S \rightarrow \epsilon$$

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:

$S \to (T\ S\ S) \overset{*}{\to} (*\ a\ a)$
$S \to (T\ S\ S) \overset{*}{\to} (T(T\ S\ S)(T\ S\ S)) \overset{*}{\to}$
$(T(T\ S(T\ S\ S))(T(T\ S\ S)(T\ S\ S))) \overset{*}{\to} (*(+\ a(*\ a\ a))(+(*\ a\ a)(*\ a\ a)))$

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:

$S \to (T_1 S) \qquad S \to (T_2 S S) \qquad S \to a$
$T_1 \to \sqrt{\ } \qquad T_1 \to - \qquad T_2 \to + \qquad T_2 \to * \qquad T_2 \to - \qquad T_2 \to /$

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.

$U \to V_1 U_1 \qquad U \to V \qquad U_1 \to V U_1 \qquad U_1 \to V$
$V_1 \to i \quad$ pour $\quad 1 \leq i \leq 9$
$V \to 0 \qquad V \to V_1$

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.

$$E \to E + T \to E + T + T \ldots\ldots \stackrel{*}{\to} T + T + T \ldots + T$$
$$T \to T * F \to T * F * F \ldots\ldots \stackrel{*}{\to} F * F * F \ldots * F$$

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

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

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.

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

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 en italique par exemple *multiplicative-expression*, *unary-expression*. 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 $\to$ 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 de plusieurs sortes. Il y a ainsi des mots réservés, une trentaine dans chacun des deux exemples, Pascal et C comme `begin`, `end`, `if`, `then`, `else`, `label`, `case`, `record`, ... pour Pascal, `struct`, `int`, `if`, `else`, `break`, ... pour le langage C. Il y a aussi dans A un certain nombre d'opérateurs et de séparateurs souvent communs à Pascal et C comme `+ * / - ; , ( ) [ ] = < >` . D'autres sont spécifiques à C, il s'agit de `{ } # & % ! .`

Dans les grammaires données en annexe, qui ne sont pas tout à fait complètes (manquent les variables et les nombres par exemple), on compte dans la grammaire de Pascal 69 lettres pour l'alphabet auxiliaire et 180 règles. Dans la grammaire du langage C, il y a 62 lettres auxiliaires et 165 règles. Du fait de leur taille importante, il est hors de question de traiter à titre d'exemples ces langages dans un cours d'analyse syntaxique. On se limitera ici aux exemples donnés plus haut comportant un nombre de règles limité mais dans lequel figurent déjà toutes les difficultés que l'on peut trouver par ailleurs.

On peut noter que l'on trouve la grammaire des expressions arithmétiques sous forme BNF dans les exemples concernant les langages Pascal et C des annexes. On remarque en effet à l'intérieur de la forme BNF de Pascal:

additive-expression:
    additive-expression additive-op multiplicative-expression
    multiplicative-expression

additive-op: one of
    `+ - or`

multiplicative-expression:
    multiplicative-expression multiplicative-op unary-expression
    unary-expression

multiplicative-op: one of
    `* / div mod and in`

unary-expression:
    primary-expression

primary-expression:
    variable
    `(` *expression* `)`

Ceci correspond dans notre notation à

$$\begin{array}{lll} E \to EUT & E \to T & \\ U \to + & U \to - & U \to \texttt{or} \\ T \to TVF & T \to F & \\ V \to * & V \to / & V \to \texttt{and} \\ F \to F_1 & F_1 \to (E) & F_1 \to a \end{array}$$

Dans la grammaire du langage C, on retrouve aussi des règles qui rappellent singulièrement les précédentes:

additive-expression:
    multiplicative-expression
    additive-expression `+` *multiplicative-expression*

multiplicative-expression:
    cast-expression
    multiplicative-expression `*` *cast-expression*

cast-expression:
    unary-expression

unary-expression:
    postfix-expression

postfix-expression:
    primary-expression

primary-expression:
    identifier
    `(` *expression* `)`

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 nœuds 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 nœud 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

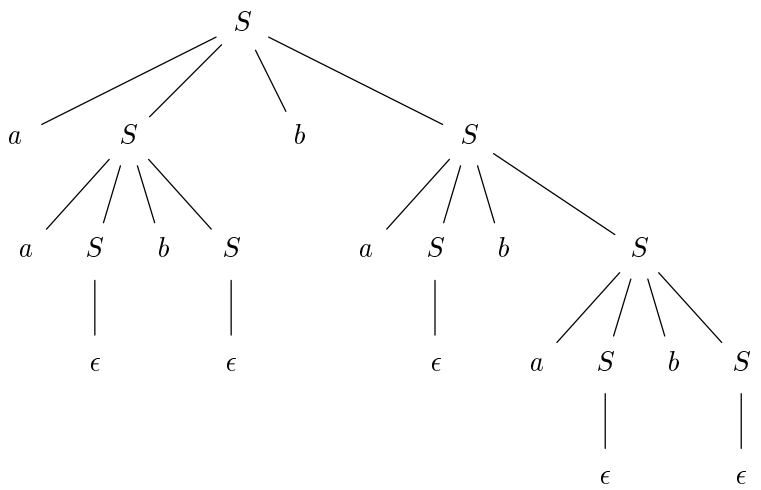


Figure 6.1 : Arbre de dérivation de $aabbabab$

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 nœud 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,

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

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

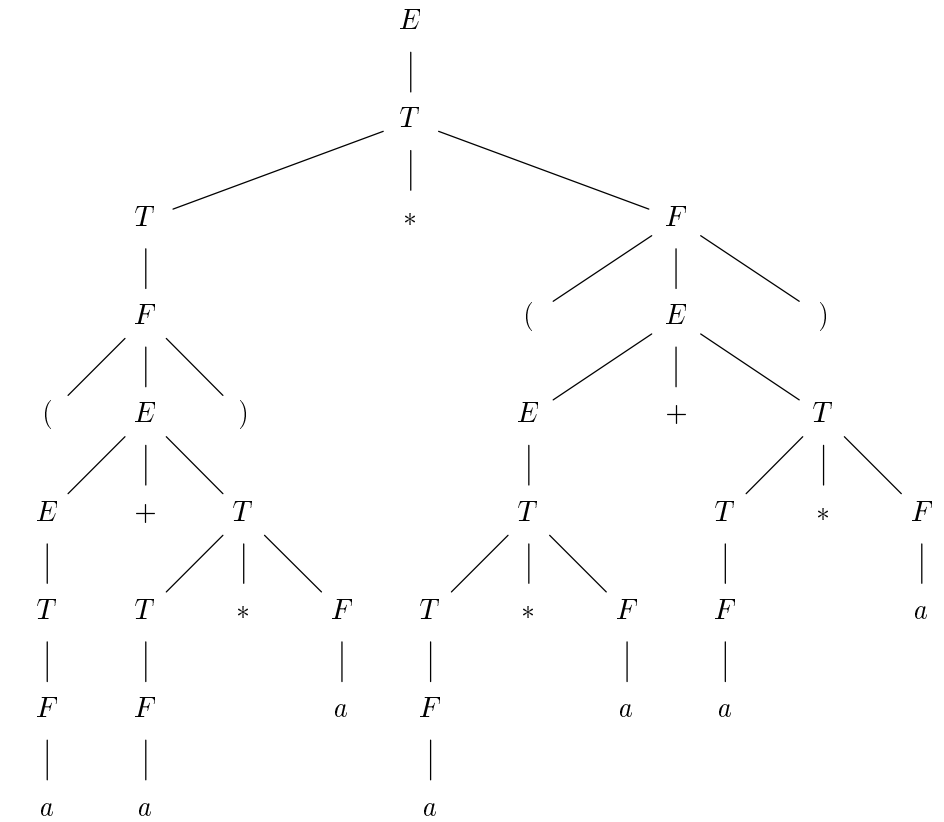


Figure 6.2 : Arbre de dérivation d'une expression arithmétique

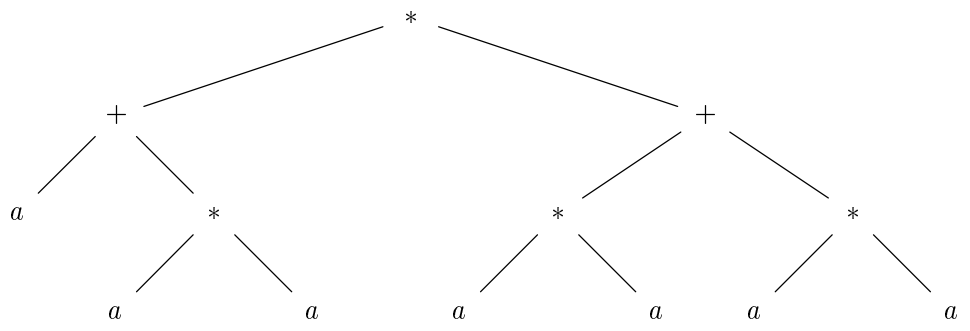


Figure 6.3 : Arbre de syntaxe abstraite de l'expression

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 Pascal. Celles ci construisent l'arbre de syntaxe abstraite en utilisant la fonction `NouvelArbre` donnée dans le chapitre 4.

```
function Terme; forward;

function Facteur; forward;

function Expression: Arbre;
    var a, b: Arbre;
    begin
    a := Terme;
    if f[i] = '+' then
        begin
        i := i + 1;
        b := Expression;
        Expression := NouvelArbre('+', a, b);
        end
    else
        Expression := a;
    end;

function Terme: Arbre;
    var a, b: Arbre;
    begin
    a := Facteur;
    if f[i] = '*' then
        begin
        i := i + 1;
        b := Terme;
        Terme := NouvelArbre('*', a, b);
        end
    else
        Terme  := a;
    end;

function Facteur: Arbre;
    begin
    if f[i]= '(' then
        begin
        i := i + 1;
        Facteur := Expression;
        if f[i] = ')' then
            i := i + 1
        else
            Erreur(i)
        end
    else
```

```
        begin
        if f[i] = 'a' then
            begin
            Facteur := NouvelArbre('a', nil, nil);
            i := i + 1;
            end
        else
            Erreur(i);
    end;
```

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.

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 $\{u_1, u_2 \dots u_p\}$ 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. Plusieurs grammaires de langages de programmation satisfont ces conditions; ainsi N. Wirth concepteur du langage Pascal a construit ce langage en s'arrangeant pour que sa grammaire vérifie une propriété de cette forme.

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

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

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.

$$y \to u_1, y \to u_2, \dots y \to u_k$$

On applique à u successivement chacune de ces règles, on obtient ainsi des mots $v_1, v_2, \dots v_k$. On poursuit l'analyse, chacun des mots $v_1, v_2, \dots v_k$ 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.

```
function Analyse(f, u: Mot): boolean;
    begin
    if f = u then Analyse := true
    else
        begin
        Mettre f et u sous la forme f = gxh, u = gyv où x ≠ y
```

```
            if y ∈ A  then Analyse := false
            else
                begin
                b := false;
                Pour toute règle y → w et tant que not(b) faire
                     b := Analyse(f, gwv);
                Analyse := b;
                end;
            end
        end
```

Cet algorithme se traduit en Pascal simplement. On utilise les procédures et fonctions `SupprimerPremLettre(u)`, `Auxiliaire(y)`, `Concatener(u,v)`, dont les noms ont été choisis pour rappeler la fonction réalisée, `SupprimerPremLettre(u)` supprime la première lettre du mot `u`, `Auxiliaire(y)` est vrai si $y \in \Xi$, `Concatener(u,v)` donne pour résultat le mot `uv`. 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-ème règle dont le membre gauche est S. Le nombre de règles dont le membre gauche est S est fourni par `nbregle[S]`.

```
function AnalyseDescendante(u: Mot; f: Mot): boolean;
    var
        i, pos: integer;
        y: char;
        v: Mot;
        b: boolean;
    begin
    b := false;
    pos := 1;
    while f[pos] = u[pos] do
        pos := pos + 1;
    b :=  (f[pos] = '$') and (u[pos] = '#');
    if not b then
        begin
        y := u[pos];
        if Auxiliaire(y) then
            begin
            i := 1;
            while (not b) and (i <= nbregle[y]) do
                begin
                v := Remplacer(u, pos, regle[y,i]);
                b := Analyse(v, f);
                if b then
                    writeln ('regle : ', y, '->', regle[y,i]) ;
                i := i + 1;
                end;
            end;
        end;
    Analyse := b;
    end;
```

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 que l'on a 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

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

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 $S \rightarrow (+SS)$ qui a été appliquée, si ces deux lettres sont (∗ alors c'est la règle $S \rightarrow (*SS)$. 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.

```
function ArbSyntPref: Arbre;
    var a, b, c: Arbre;
        x: char;

    begin
    if f[pos] = 'a' then
        begin
        a := NouvelArbre('a', nil, nil);
        pos := pos + 1;
        end
    else if (f[pos] = '(') and (f[pos+1] in {'+', '*'}) then
        begin
        x := f[pos + 1];
        pos := pos + 2;
        b := ArbSyntPref;
        c := ArbSyntPref;
        a := NouvelArbre(x, b, c);
```

```
        if f[pos] = ')' then pos := pos + 1
             else Erreur(pos);
        end
    else
        Erreur(pos);
    ArbSyntPref := a
    end;
```

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.

```
function Analyse(f: Mot; pos: integer): boolean;
    var i: integer;

    begin
    pos := 1;
    while Pvaleur(p) = f[pos] do
        begin
        Psupprimer(p);
        pos := pos + 1;
        end;
    if Pvide(p) and f[pos] = '$' then
        Analyse := true
    else
        begin
        y := Pvaleur(p);
        if not Auxiliaire(y) then
           Analyse := false
        else
           begin
           i := Predicteur (y, pos, pos+k-1);
           if i <> Omega then
                begin
                writeln (y, '->', regle[y,i]);
                Pinserer(regle[y,i], p);
                Analyse := Analyse (f, pos);
                end
```

```
            else
                Analyse := false;
            end;
        end;
    end;
```

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

$$S_0 \to u_1 \to u_2 \dots u_{n-1} \to u_n = f$$

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:

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

On voit bien que les règles $S \to aSbS$ et $S \to aSb$ 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 nœuds). 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.

```
function Evaluer(x: Arbre): integer;
    begin
    if x^.valeur = 'a' then
        Evaluer := x^.valeur
```

```
    else if x^.valeur = '+' then
        Evaluer := Evaluer (x^.filsG) + Evaluer (x^.filsD)
    else if x^.valeur = '-' then
        Evaluer := Evaluer (x^.filsG) - Evaluer (x^.filsD)
    else if x^.valeur = '*' then
        Evaluer := Evaluer (x^.filsG) * Evaluer (x^.filsD)
    end
```

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 nœud de l'arbre, par la concaténation des listes d'instructions qui calculent le sous-arbre droit et le sous arbre gauche de ce nœud 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 C

```
/* Analyse descendante simple voir page 155 */

Arbre Terme();

Arbre Facteur();

Arbre Expression()
{
    Arbre a, b;

    a = Terme();
    if (f[i] == '+') {
        i++;
        b = Expression();
        return NouvelArbre('+', a, b);
    }
    else return a;
}

Arbre Terme()
{
    Arbre a, b;

    a = Facteur();
    if (f[i] == '*') {
        i++;
        b = Terme();
        return NouvelArbre('*', a, b);
    }
    else return a;
}

Arbre Facteur()
{
    Arbre a;
```

```
    if (f[i] == '(') {
        i++;
        a = Expression();
        if (f[i] == ')') {
            i++;
            return a;
            }
        else Erreur(i);
    }
    else if (f[i] == 'a') {
            i++;
            a = NouvelArbre ('a', NULL, NULL);
            return a;
         }
         else Erreur(i);
}
```

```
/* Analyse descendante récursive, voir page 157 */
int AnalyseRecursive (Mot f, Mot u)
{
    int i, pos;
    char x, y;
    Mot v;
    int b;

    pos = 1;
    b = 0;
    while (f[pos] == u[pos])
        ++pos;
    if (f[pos] == '$' && u[pos] == '+') {
        printf("analyse re'ussie \n");
        b = 1;
    }
    else if (Auxiliaire(y)) {
         i = 1;
         while ( (!b) && (i <= nbregle[y -'A'])) {
             v = Remplacer (u,  regle[y-'A'][i], pos);
             b = AnalyseRecursive (v, f);
             if (b)
                printf ("regle %d du nonterminal %c \n", i, y);
             else i++;
         }
   }
   return b;
}
```

```
Arbre ArbSyntPref()                    /* Analyse LL(1), voir page 159 */
{
    Arbre a, b, c;
    char x;
```

```
    if (f[pos] == 'a') {
        a = NouvelArbre( 'a', NULL, NULL);
        pos++;
        }
    else if (f[pos] == '(') && ((f[pos + 1] == '+') ||
                                (f[pos + 1] == '*')) {
         x = f[pos + 1];
         pos = pos +2;
         b = ArbSyntPref();
         c = ArbSyntPref();
         a = NouvelArbre(x, b, c);
         if (f[pos] == ')' ) pos++;
             else Erreur(pos);
         }
    else Erreur(pos);
}
```

```
int Evaluer(Arbre x)     /* Evaluation, voir page 162 */
{
    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));
    else Erreur();
```

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, 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 puisqu'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.

Une file de caractères sera alors décrite par une référence vers un enregistrement 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

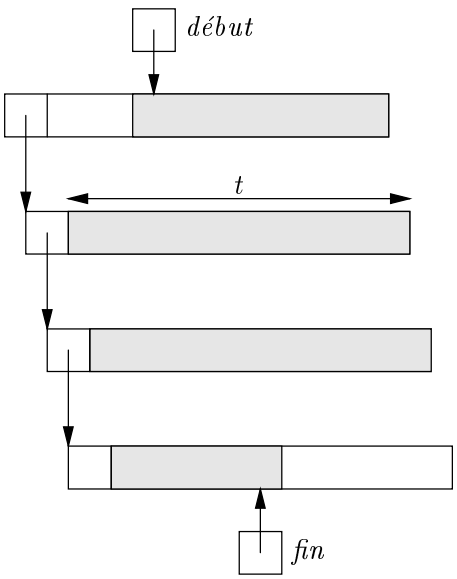


Figure 7.1 : File de caractères

et déplacement d'un caractère, nous entendons une référence vers un bloc de la liste contenant le caractère et son adresse relative dans ce bloc comme indiqué sur la figure 7.2. La déclaration du type `FCtype` d'une file de caractères s'effectue comme suit en Pascal:

```
const
    FCtailleBloc = 12;
type
    FCtype = ^Cellule;
    BlocPtr = ^Bloc;
    BaseDeplacement = record
        b: BlocPtr;
        d: integer;
        end;
    Cellule = record
        cc: integer;
        debut, fin: BaseDeplacement;
        end;
    Bloc = record
        suivant: BlocPtr;
        contenu: array[1..FCtailleBloc] of char;
        end;
```

La file vide est représentée par un compte de caractères nul.

```
procedure FCvide (var x: FCtype);
    begin new(x); x^.cc := 0 end;
```

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

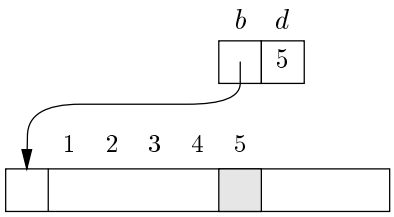


Figure 7.2 : Adresse d'un caractère par base et déplacement

le même bloc ou s'il est nécessaire d'aller chercher le bloc suivant dans la liste des blocs. Lors de l'ajout, il faut allouer un nouveau bloc dans le cas de la file vide ou du franchissement d'un bloc.

```
procedure FCajouter (x: FCtype; c: char);
    var
        bp: BlocPtr;
    begin
    if x^.cc = 0 then
        begin
        FC_nouveauBloc(bp);
        x^.fin.b := bp; x^.fin.d := 0;
        x^.debut.b := bp; x^.debut.d := 1;
        end
    else if x^.fin.d = FCtailleBloc then
        begin
        FC_nouveauBloc(bp);
        x^.fin.b^.suivant := bp;
        x^.fin.b := bp; x^.fin.d := 0;
        end;
    x^.fin.d := x^.fin.d + 1;
    x^.fin.b^.contenu[x^.fin.d] := c;
    x^.cc := x^.cc + 1;
    end;
```

où la procédure d'allocation d'un nouveau bloc est donnée par:

```
procedure FC_nouveauBloc (var bp: BlocPtr);
    begin new(bp); bp^.suivant := nil end;
```

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 erreur, mais c'est relativement compliqué de le faire en Pascal, et ça ne nous arrange pas ici.

```
function FCsupprimer (x: FCtype): char;
    var
        bp: BlocPtr;
    begin
    if x^.cc = 0 then
        FCsupprimer := chr(0)
```

```
else
    begin
    FCsupprimer := x^.debut.b^.contenu[x^.debut.d];
    x^.cc := x^.cc - 1;
    x^.debut.d := x^.debut.d + 1;
    if x^.cc <= 0 then
        dispose(x^.debut.b);
    else if x^.debut.d > FCtailleBloc then
        begin
        bp := x^.debut.b;
        x^.debut.b := bp^.suivant; x^.debut.d := 1;
        dispose(bp);
        end;
    end;
end;
```

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 du type `FCtype` et des 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:

```
type FCtype;
(* Le type d'une liste de caractères *)

procedure FCvide (var x: FCtype);
(* Initialise x par une file vide *)

procedure FCajouter (x: FCtype; c: char);
(* Ajoute c au bout de x *)

function FCsupprimer (x: FCtype): char;
(* Supprime le premier caractère c de x et rend c comme résultat *)
(* Si x est vide, le résultat est chr(0) *)
```

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. Mais dans un module, il y a aussi une partie *cachée* comprenant les types et les corps des procédures ou des fonctions que l'on veut rendre privées. Dans l'interface, il est bon de commenter la fonctionnalité de tous les objets exportés pour comprendre leurs significations, puisque ne figurent pas les programmes qui les réalisent.

Il n'y a pas que les types ou les lignes de programmes à cacher, mais aussi les variables et les fonctions. Reprenons l'exemple des files de caractères, et supposons

que dans le Pascal utilisé la fonction `dispose` soit défaillante et que l'on préfère gérer soi-même l'allocation et la libération des blocs. On construira une liste des blocs libres `listeLibre` dans une nouvelle procédure d'initialisation `FCinit`, et les procédures `FC_NouveauBloc` et `FC_LibererBloc` remplaceront les vieilles procédures `FC_NouveauBloc` et `dispose` comme suit:

```
const
    FCtailleBloc = 12;
    FCnbBlocs = 1000;
type
    FCtype = ^Cellule;
    ...
var listeLibre: BlocPtr;

procedure FCinit;
    var i: integer;
        bp: BlocPtr;
    begin
    listeLibre := nil;
    for i := 1 to FCnbBlocs do
        begin
        new(bp); bp^.suivant := listeLibre;
        listeLibre := bp;
        end;
    end;

procedure FC_NouveauBloc (var bp: BlocPtr);
    begin
    bp := listeLibre; listeLibre := listeLibre^.suivant;
    bp^.suivant := nil;
    end;

procedure FC_LibererBloc (var bp: BlocPtr);
    begin bp^.suivant := listeLibre; listeLibre := bp end;
```

Dans l'interface des files de caractères, on doit rajouter la procédure d'initialisation `FCinit`, mais 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 deux remarques rapides. Premièrement, il est fréquent qu'un module nécessite une procédure d'initialisation. Ensuite, pour ne pas compliquer le programme Pascal, nous ne testons pas le cas où la liste des blocs libres devient vide et donc l'allocation d'un nouveau bloc libre impossible. L'interface devient donc

```
type FCtype;
(* Le type d'une liste de caractères *)

procedure FCinit;
(* Initialise le module des files de caractères. A faire absolument *)
(* avant d'utiliser une autre fonction ou procédure de cette interface *)
...
```

Si les procédures d'allocation et de libération de blocs étaient très compliquées, on créerait un nouveau module d'allocation et on importerait l'interface d'allocation mémoire dans le module des files de caractères.

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 du 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. Toutefois, ici, nous insistons sur la structuration des programmes, car tout le reste n'est que corollaire. Tout le problème de la modularité se résume donc 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 Pascal

Une certaine notion de modularité existe déjà en Pascal avec le découpage en fonctions ou procédures et la notion de bloc. Toutefois, un objet commun à deux procédures disjointes doit se trouver dans un bloc englobant. Ceci aboutit rapidement à repousser beaucoup de variables dans le bloc global. Comme les variables globales sont accessibles dans tout le programme, il est impossible de cacher une entité commune à deux fonctions que l'on veut rendre publiques. Il est donc malheureusement impossible de réaliser en Pascal des modules comme précédemment décrits. Certains langages qui sont des prolongements de Pascal ont des constructions spéciales, comme Mesa, Ada, Modula-2, Oberon ou Modula-3. En C++, les classes permettent une forme de modularité. En ML [36, 52], les modules permettent la vraie abstraction des types et la paramétricité. (On peut faire un module sur les listes de n'importe quel type). En C, on rencontre des problèmes similaires à ceux de Pascal, en moins graves.

On peut néanmoins tourner la difficulté en faisant coïncider les notions de module et de compilation séparée, qui ont en principe si peu à voir.

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. Dans la suite, chaque module d'implémentation sera confondu avec un fichier source, et la notion de module correspondra donc à la notion de fichier. La compilation séparée dépend beaucoup du système d'exploitation utilisé (Unix, Mac/OS, MS-DOS). Dans le cas d'Unix, la commande

```
% pc -c files-de-caracteres.p
```

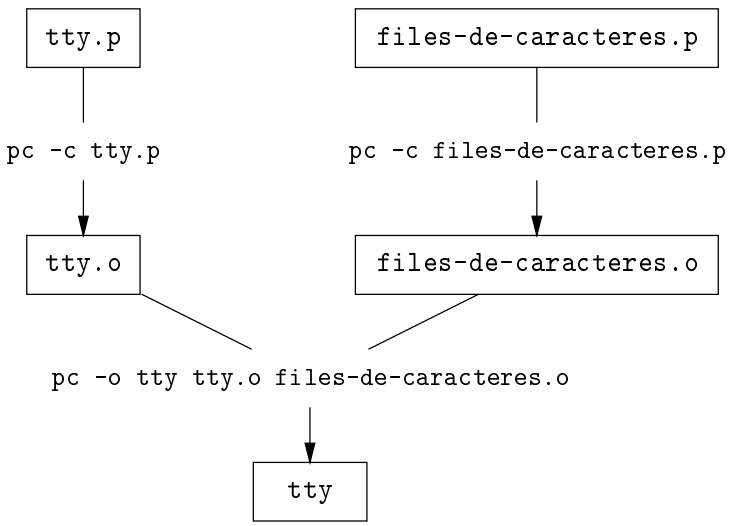


Figure 7.3 : Compilation séparée

permet d'obtenir le fichier *binaire relogeable* `files-de-caracteres.o` qui peut être relié à d'autres modules compilés indépendamment. Supposons qu'un fichier `tty.p` contenant un gestionnaire de terminaux utilise les fonctions sur les files de caractères. En Unix, on devra compiler séparément `tty.p` et relier les deux binaires obtenus, ce qui s'obtient en écrivant

```
% pc -c tty.p
% pc -o tty tty.o files-de-caracteres.o
```

Le binaire exécutable résultat est dans le fichier `tty`. Graphiquement, les phases de compilation sont représentées par la figure 7.3.

Quand il y a un grand nombre de fichiers binaires relogeables à relier entre eux, par exemple tous ceux qui permettent de faire fonctionner un gros système comme X-window, on peut pré-relier tous les binaires relogeables entre eux dans une *librairie*, par exemple `libX11.a` pour X-window. Et la commande

```
% pc -o tty tty.o files-de-caracteres.o libX11.a
```

permet de relier les deux fichiers `.o` avec les fichiers nécessaires dans `libX11.a`.

En ThinkPascal, la notion de *projet* permet de combiner un certain nombre de fichiers Pascal comme `files-de-caracteres.p` et `tty.p`, 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.

Essayons à présent de faire coïncider les notions de modules et de compilation séparée. En Pascal BSD (sur Vax), on peut se servir des fichiers *include* pour les interfaces, et de la directive `external`. Une fonction externe est une fonction qui se trouvera dans un autre module compilé séparément. Elle a le même format que la directive `forward` (cf Appendice A). En Pascal BSD, on peut inclure un fichier avant la partie implémentation ou utilisatrice de l'interface. Dans le cas des files de caractères, on pourra avoir le fichier interface `files-de-caracteres.h` suivant

```
const
```

```
    FCtailleBloc = 12;
type
    FCtype = ^Cellule;      (* Le type d'une liste de caractères *)
    BlocPtr = ^Bloc;
    BaseDeplacement = record
        b: BlocPtr;
        d: integer;
        end;
    Cellule = record
        cc: integer;
        debut, fin: BaseDeplacement;
        end;
    Bloc = record
        suivant: BlocPtr;
        contenu: array[1..FCtailleBloc] of char;
        end;

procedure FCinit;
(* Initialise le module des files de caractères. A faire absolument *)
(* avant d'utiliser une autre fonction ou procédure de cette interface *)
    external;

procedure FCvide (var x: FCtype);
(* Initialise x par une file vide *)
    external;

procedure FCajouter (x: FCtype; c: char);
(* Ajoute c au bout de x *)
    external;

function FCsupprimer (x: FCtype): char;
(* Supprime le premier caractère c de x et rend c comme résultat *)
(* Si x est vide, le résultat est chr(0) *)
    external;
```

Remarquons que la syntaxe acceptée par Pascal, nous interdit de cacher le corps de la structure de donnée `FCtype`. Pour l'utilisation de l'interface, le fichier `tty.p` sera

```
program TTY;
#include "files-de-caracteres.h"
... un programme ou un module normal
end.
```

et la partie implémentation `files-de-caracteres.p` sera

```
#include "files-de-caracteres.h"
var listeLibre: BlocPtr;

procedure FCinit;
    var i: integer;
        bp: BlocPtr;
    begin
    listeLibre := nil;
    for i := 1 to FCnbBlocs do
        begin
        new(bp); bp^.suivant := listeLibre;
```

```
            listeLibre := bp;
            end;
        end;

    procedure FCajouter {x: FCtype; c: char};
        var bp: BlocPtr;
        begin
        if x^.cc = 0 then
        ...
```

On a inclut le fichier d'interface pour au moins garantir que les types des objets manipulés par le corps des programmes sont du type indiqué par l'interface. A ce propos, Pascal interdit de redéclarer la signature des fonctions `external`, comme pour les directives `forward`. Donc nous avons mis des accolades de commentaires plutôt que des parenthèses dans la définition de FCajouter. En ThinkPascal, il faut utiliser des directives très spécifiques `unit`, `uses` et `implementation`. Le fichier `files-de-caracteres.p` devient:

```
unit FilesDeCaracteres;
interface
    on recopie le contenu du fichier files-de-caracteres.h
    sans les directives external
implementation
    le contenu du fichier files-de-caracteres.p sans la ligne include
end.
```

et le fichier utilisateur `tty.p`

```
program TTY;
    uses FilesDeCaracteres;
le reste du programme
```

On peut remarquer que Think autorise à redéclarer la signature des fonctions dans la partie implémentation. De même, il n'est pas nécessaire de mettre `external` après les définitions de l'interface. En Think, il faut bien faire attention à mettre les modules dans le bon ordre dans le projet pour que la compilation se passe bien.

Mais nous allons garder uniquement la technique de Pascal BSD, car elle est très similaire à ce que l'on fait dans le langage C avec le préprocesseur (cf la section 7.7).

Enfin, on doit encore remarquer qu'il est très difficile de cacher les noms des objets privés d'un module en Pascal. Pour les types, nous avons vu que le langage nous impose de rendre publics les composants du type. Pour les variables et fonctions, les noms sont toujours déclarés comme *externes* par l'éditeur de liens d'Unix, contrairement à C (voir la directive `static` de la section 7.7). Si on veut éviter les collisions entre noms identiques de variables ou de fonctions différentes de modules différents, il est préférable d'utiliser le nom du module comme préfixe des noms de variables ou de fonctions privées du module (comme nous l'avons fait pour `FC_NouveauBloc`). Le principe de coupler les notions de compilation séparée et de modules est donc de se servir de l'éditeur de liens pour rendre inaccessibles les variables cachées d'un module.

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. Par exemple, dans le cas précédent de notre gestionnaire de terminaux, nous

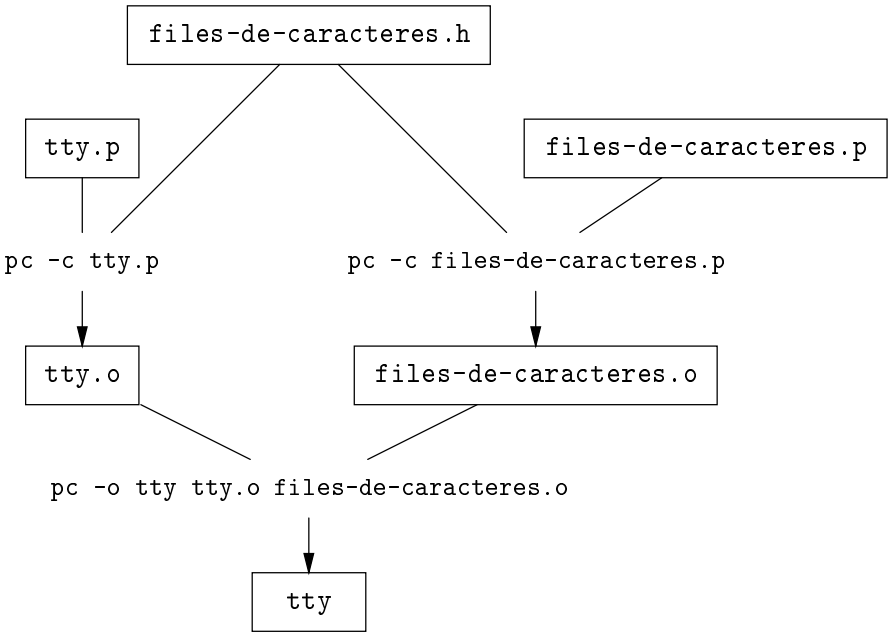


Figure 7.4 : Dépendances dans un *Makefile*

voulons indiquer que les dépendances induites par la figure 7.4 pour construire l'exécutable `tty`. Il faut donc signaler que `files-de-caracteres.h` et `tty.p` sont nécessaires pour fabriquer `tty.o`, comme `files-de-caracteres.h` et `files-de-caracteres.c` le sont pour `files-de-caracteres.o`. Enfin, les deux fichiers `.o` sont nécessaires pour faire `tty`.

La description des dépendances varie selon le système. En Unix, le principe est de décrire le graphe des dépendances précédents dans un fichier *Makefile*. La commande `make` calcule les dernières dates de modification des fichiers en cause dans ce *Makefile*, et effectue les opérations strictement nécessaires pour reconstruire le fichier exécutable `tty`. Le *Makefile* a la syntaxe suivante:

```
tty: tty.o files-de-caracteres.o
        pc -o tty tty.o files-de-caracteres.o

tty.o: tty.p files-de-caracteres.h
        pc -c tty.p

files-de-caracteres.o: files-de-caracteres.p files-de-caracteres.h
        pc -c files-de-caracteres.p
```

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 `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* et nous allons la considérer dans un contexte beaucoup plus large. Remarquons auparavant qu'en Think Pascal, on doit faire cette opération manuellement en déclarant dans le

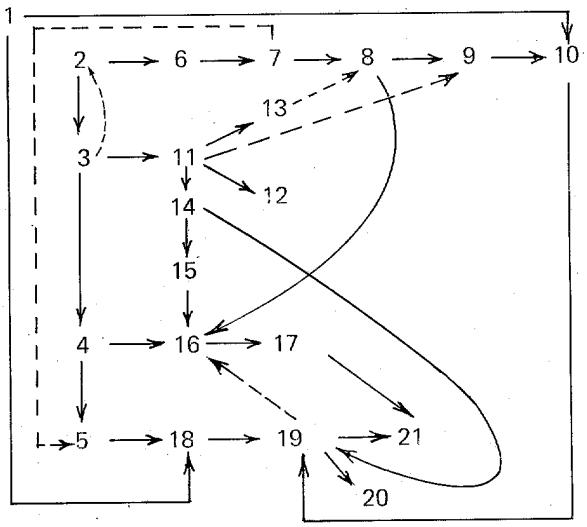


Figure 7.5 : Un exemple de graphe acyclique

bon ordre les composants d'un projet.

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 nœuds 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 nœud déjà visité que celui-ci figure dans la liste résultat. Pour ce faire, il faut utiliser un

tableau annexe `etat` sur les nœuds qui indique si le nœud est visité, en cours de visite, ou non visité.

```
procedure TriTopologique (u: Sommet; var resultat: Liste);
    ...
    procedure DFS (u: Sommet);
        var k: integer;
            v: Sommet;
        begin
        k := 1; v := pred[u, k];
        while v <> Omega do
            begin
            if etat[v] = NonVu then
                begin etat[v] := EnCours; DFS (v) end
            else if etat[v] = EnCours then
                Erreur ('Le graphe a un cycle');
            k := k + 1; v := pred[u, k];
            end;
        etat[u] := Vu;
        res := Cons (u, res);
        end;
    begin
    for i := 1 to Nsommets do etat[i] := NonVu;
    resultat := nil; resultat := Reverse (DFS (u));
    end;
```

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 nœud.

7.7 Programmes en C

Ici, il est intéressant d'examiner la différence entre Pascal et C. Les noms de fonctions sont différents de ceux donnés dans le programme Pascal pour les files de caractères, car nous avons copié textuellement ce code directement du noyau du système Unix. C'est 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 de Pascal. 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 `files-de-caracteres.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 FCget and FCput
 * 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 *FCtype;

int  FCput(char c, FCtype p);
int  FCget(FCtype p);
void FCinit(void);
```

Dans la partie implémentation qui suit, on remarque l'emploi de la directive `static` 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 `FCtype` 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 `files-de-caracteres.c` est

```
#include <stdlib.h>
#include <files-de-caracteres.h>

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

int FCput(char c, FCtype 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 FCget(FCtype 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 FCinit()
{
    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;
    }
}
```

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 œuvre 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 :

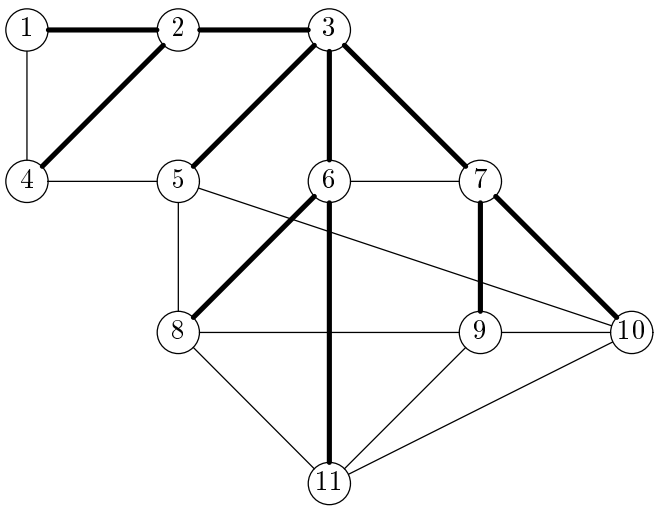


Figure 8.1 : Un graphe symétrique et l'un de ses arbres recouvrants

- *Etape 1:* Classer les arêtes par ordre de poids croissants. Elles constituent alors une suite

  $$e_1, e_2, \ldots e_n$$

  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.

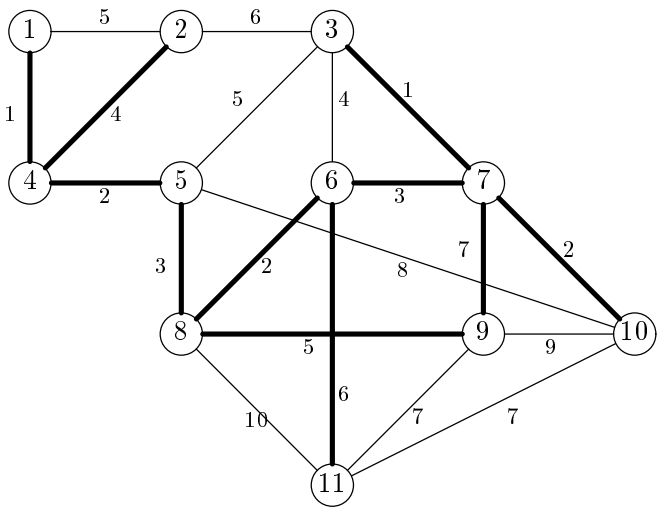


Figure 8.2 : Un arbre recouvrant de poids minimum

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.

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

[1] Dans le cas où M est en entier, on peut trouver un algorithme très efficace fondé sur la programmation dynamique.

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[1..n]` permettant de coder toutes les possibilités, un objet i est mis dans le sac si `sac[i] = 1`, il n'est pas mis si `sac[i] = 0`. Il faut donc parcourir tous les vecteurs possibles de 0 et de 1, pour cela on considère successivement toutes les positions $i = 1, \ldots n$ et on effectue les deux choix possibles `sac[i] = 0` ou `sac[i] = 1` 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 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`.

```
procedure Calcul (i: integer; u: real);
    begin
    if i > n then
        if u < meilleur then
            begin
            for j:= 1 to n do msac[i] := sac[i];
            meilleur := u;
            end;
    else
        begin
        if  p[i] <= u then
            begin
            sac[i] := 1;
            Calcul(i + 1, u - p[i]);
            end;
        sac[i] := 0;
        Calcul(i + 1, u);
        end;
    end;
```

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

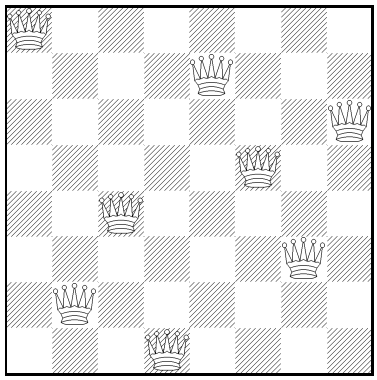


Figure 8.3 : Huit reines sur un échiquier

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).

```
function Conflit (i1, j1, i2, j2: integer): boolean;
    begin
    Conflit := (i1 = i2) or (j1 = j2)
            or (abs (i1 - i2) = abs (j1 - j2));
    end;
```

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 $1, \ldots, i-1$:

```
function Compatible (i, j: integer): boolean;
    var k: integer;
        c: boolean;
    begin
        c := true;
        k := 1;
        while c and (k < i) do
            begin
            c := not Conflit (i, j, k, pos[k]);
            k := k + 1
            end;
        Compatible := c;
    end;
```

La fonction récursive qui trouve une solution au problème des reines est alors la suivante:

```
procedure Reines (i: integer)
    begin
    if i > Nreines then
        Imprimer_Solution
    else
        begin
        for j:= 1 to Nreines do
        if Compatible (i,j) then
            begin
```

```
            pos[i] := j;
            Reines(i+1)
            end;
        end;
    end;
```

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 les procédure en s'arrêtant dès que l'on a trouvé une solution ou pour simplement compter le nombre de solutions différentes. 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 := 1 to n
    for i := 1 to n
        for j := 1 to n
           delta[i,j] := 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:

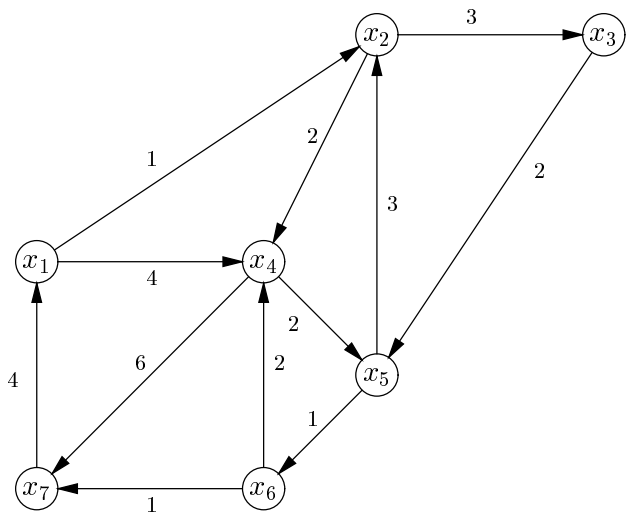


Figure 8.4 : Un graphe aux arcs valués

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

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 k := 1 to n
    for i := 1 to n
        for j := 1 to n
          if delta[i, j] > (delta[i, k] + delta[k, j]) then
              begin
              delta[i, j] :=  delta[i, k] + delta[k, j];
              suivant[i, j] := suivant[i, k];
              end;
```

Une fois le calcul des deux matrices effectué on peut retrouver le chemin le plus court qui joint i à j par la procédure:

```
procedure PlusCourtChemin(i, j: integer);
    var k: integer;
    begin
```

```
k := i;
while k <> j do
    begin
    write (k, '  ');
    k := suivant[k, j];
    end;
writeln(j);
end;
```

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 $b_1 \cdots b_m$

```
var
    long: array[0...Nmax, 0...Nmax] of integer;

function LongSSC(n, m:integer; var u, v: array [1..Nmax] of char);
    var i, j: integer;
    begin
        for i := 0 to n do long[i, 0] := 0;
```

```
        for j := 1 to m do long[0, j] := 0;
        for i := 1 to n do
            for j := 1 to m do
                if v[j] = u[i] then
                     long[i,j] := 1 + long[i-1, j-1]
                else if long[i,j-1] > long[i-1, j] then
                     long[i,j] := long[i, j-1]
                else
                     long[i,j] := long[i-1,j];
    end;
```

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.

```
type
    Sequence =  array[1..Nmax] of char;

var
    long: array[0...Nmax, 0...Nmax] of integer;
    provient: array[1...Nmax, 1...Nmax] of integer;

function LongSSC(n, m: integer; var u, v: Sequence);
    var i, j: integer;
    begin
        for i := 0 to n do long[i, 0] := 0;
        for j := 1 to m do long[0, j] := 0;
        for i := 1 to n do
            for j := 1 to m do
                if v[j] = u[i] then
                     begin
                     long[i,j] := 1 + long[i-1, j-1];
                     provient[i,j] := 1;
                     end
                else if long[i,j-1] > long[i-1, j] then
                     begin
                     long[i,j] := long[i, j-1];
                     provient[i,j] := 2;
                     end
                else
                     begin
                     long[i,j] := long[i-1,j];
                     provient[i,j] := 3
                     end
    end;
```

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]`.

```
procedure SSC (n, m: integer; var p: integer; var u, v, w: Sequence);
    var
        i, j, k: integer;
    begin
```

```
        LongSSC(n, m, u, v);
        p := long[n,m];
        i := n; j := m; k := p;
        while k <> 0 do
            if provient[i,j] = 1 then
                begin
                w[k] := u[i];
                i := i - 1;
                j := j - 1;
                k := k - 1;
                end
            else if provient[i,j] = 2 then
                j := j - 1;
            else
                i := i - 1;
    end;
```

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.

Annexe A

Pascal

Pascal est un langage typé, conçu par Wirth [19] en 1972, comme une simplification du langage AlgolW [23], langage précurseur dû aussi à Wirth. Son principe repose sur le typage et une implémentation facile.

A.1 Un exemple simple

Considérons l'exemple des carrés magiques. Un carré magique est une matrice a carrée de dimension $n \times n$ telle que la somme des lignes, des colonnes, et des deux diagonales soient les mêmes. Si n est impair, on met 1 au milieu de la dernière ligne en $a_{n,\lfloor n/2 \rfloor + 1}$. On suit la première diagonale 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?

Ecrivons le programme correspondant:

```
program CarreMagique (input, output);

label 999;                                  (* Déclaration de l'étiquette 999 *)
const NMax = 100;
type  T = array [1..NMax, 1..NMax] of integer;

var a: T;
    n: integer;
```

```
procedure Init (n: integer);
    var i, j: integer;

    begin
    for i := 1 to n do
        for j := 1 to n do
            a[i, j] := 0;
    end;

function Pred (i: integer): integer;
    begin
    if i > 1 then
        Pred := i - 1;
    else
        Pred := n;
    end;

procedure Magique (n: integer);

    var i, j, k: integer;

    begin
    i := n; j := n div 2 + 1
    for k := 1 to n * n do
        begin
        while a[i, j] <> 0 do
            begin
            i := Pred (Pred (i));
            j := Pred (j);
            end;
        a[i, j] := k;
        i := 1 + i mod n;
        j := 1 + j mod n;
        end;
    end;

procedure Erreur;
    begin
    writeln ('Taille impossible.');
    goto 999;
    end;

procedure Lire (var n: integer);
    begin
    write('Taille du carre'' magique, svp?::  ');
    readln(n);
    if (n <= 0) or (n > NMax) or not odd(n) then Erreur;
    end;

procedure Imprimer (n: integer);
    var i, j: integer;
    begin
    for i := 1 to n do
        begin
        for j := 1 to n do
```

```
            write (a[i, j] : 4);
        writeln;
        end;
    end;

begin
Lire(n);
Init(n);
Magique(n);
Imprimer(n);
999:                                    (* Définition de l'étiquette 999 *)
end.
```

Plusieurs remarques peuvent être faites. D'abord, on constate que les déclarations des objets doivent précéder leur utilisation. Les déclarations suivent un ordre très précis: les étiquettes, les constantes, les types, les variables globales du programme, les procédures ou fonctions, et le corps du programme principal.

Le programme a curieusement un en-tête avec un nom pour désigner le programme, et éventuellement des noms de fichiers, qui étaient obligatoires autrefois et le sont moins dans les versions récentes. Une bonne manière d'assurer la compatibilité entre les différentes versions de Pascal est de toujours mettre `(input, output)` comme noms de fichiers.

Les étiquettes sont des constantes entières et non des identificateurs. Le principe est que leur utilisation doit être découragée pour n'écrire que des programmes structurés. On utilisera principalement les étiquettes dans le cas des arrêts exceptionnels, on déclarera alors une étiquette `999` ou toute autre valeur frappante, et cette étiquette peut être définie à la fin du programme.

Les variables sont typées, c'est à dire ne prennent leur valeur que dans l'ensemble défini par leur type. Par exemple, `var x: integer` définit la variable `x` de type entier, c'est-à-dire `-maxint` $\leq$ `x` $\leq$ `maxint`. Typiquement `maxint` $= 2^{n-1}$ avec $n = 16$ ou $n = 32$.

Pascal permet de définir des types plus structurés. Considérons d'abord les tableaux comme dans l'exemple du carré magique. La ligne

```
var a: array [1..NMax, 1..NMax] of integer;
```

permet de définir une matrice `NMax` $\times$ `NMax`, dont les éléments sont des entiers. En fait, nous avons déclaré un type `T`, ce qui permettrait de réutiliser ce type, si on avait une autre matrice de même type. Ici, cette déclaration est inutile. Chaque élément du tableau `a` peut être accédé par la notation `a[i, j]`, comme $a_{i,j}$ est un élément de la matrice a.

Plus important, la constante `NMax` a été déclarée pour permettre de paramétrer le programme par cette constante. Une des règles d'or de la programmation consiste à localiser les déclarations de constantes, plutôt que d'avoir leurs valeurs éparpillées sur tout le programme. On peut alors changer facilement la valeur de `NMax`. La variable `n` est la taille effective de la matrice `a`. On veillera donc à avoir toujours la relation 1 $\leq$ `n` $\leq$ `NMax`, ce que l'on vérifie dans la procédure `Lire`. En Pascal,

les tableaux sont de taille fixe

Ceci signifie que l'on doit déclarer une matrice `a` de taille maximale `NMax` $\times$ `NMax`, et travailler dans le sous-bloc en haut et à gauche de taille `n` $\times$ `n`. Tout programme utilisant des tableaux rencontre ce problème.

Après les déclarations de variables, un programme peut contenir une suite de déclarations de procédures ou de fonctions. Une procédure permet de regrouper un certain nombre d'instructions, et peut prendre des arguments. Une fonction fait la même chose, mais rend un résultat. Par exemple, la procédure `Init` remet à zéro toute la matrice `a`. Elle prend comme argument la taille `n` de la sous-matrice effectivement utilisée. La procédure contient les variables locales `i` et `j`, indices des deux boucles `for`. Remarque: il faut toujours déclarer les indices de boucles, plutôt comme variables locales. Dans certaines versions de Pascal, on est obligé de les déclarer dans le plus petit bloc contenant les instructions `for`.

La fonction `Pred` donne le prédécesseur d'un indice variant sur l'intervalle $[1, n]$. Il faut faire attention que le prédécesseur de 1 est n, ce qui n'est pas trop facile à écrire naturellement avec la fonction `mod`, donnant le reste de la division entière. Cette fonction a souvent des résultats inattendus pour les nombres négatifs. Syntaxiquement, on remarque qu'une fonction a un type résultat, ici entier, et que le résultat est donné en affectant au nom de la fonction la valeur retournée.

La procédure `Magique` construit un carré magique d'ordre n impair. 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, 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. On vérifie aisément qu'alors toutes les sommes des valeurs sur toutes les lignes, colonnes et diagonales sont identiques. La logique même de cet algorithme fait apparaître d'autres solutions pour la procédure `Magic`, comme:

```
procedure Magique (n: integer);

    var i, j, k: integer;

    begin
    i := n; j := n div 2 + 1;
    for k := 1 to n * n do
        begin
        a[i, j] := k;
        if (k mod n) = 0 then
                 i := i - 1
        else begin
            i := 1 + i mod n;
            j := 1 + j mod n;
            end;
        end;
    end;
```

Cette version de la procédure est meilleure, car elle n'implique pas l'initialisation préalable du tableau `a`. Elle est toutefois moins naturelle que la première version. Ce raisonnement est typique de la programmation, plusieurs solutions sont possibles pour un même problème que l'on atteint par raffinements successifs.

La procédure `Lire` lit la valeur de la variable globale `n`, et fait quelques vérifications sur sa valeur. Pour lire la valeur, on utilise un appel de la procédure de lecture d'une ligne `readln`. Auparavant, un message est imprimé sur le terminal pour demander la valeur de `n`. La procédure `write` d'impression sur terminal peut prendre en paramètre une chaîne de caractères, délimitée par des apostrophes. Remarque: comme notre message contient une apostrophe, on doit la doubler pour la différencier de la fin de chaîne. Remarquons que la procédure `Lire` donne une valeur à son paramètre `n`. On ne peut se contenter de la valeur du paramètre, et on doit passer son paramètre par référence, pour

identifier son paramètre formel et son paramètre réel. La notion d'appel par référence sera vue en détail page 204.

La procédure `Imprimer` imprime le tableau `a` en utilisant les procédures d'impression sur terminal `write` et `writeln`. Ces procédures peuvent prendre non seulement des chaînes de caractères en argument comme dans `Lire`, mais aussi des valeurs entières. La différence entre `write` et `writeln` est que cette dernière fait un retour à la ligne après l'impression. Remarque: on peut mettre des informations de format derrière la valeur entière pour signaler que l'impression se fera sur 4 caractères, cadrée à droite.

Enfin, le programme principal fait un appel successif aux différentes procédures. Un bon principe consiste à réduire d'autant plus la taille du programme principal que le programme est long. Il est préférable de structurer les différentes parties d'un programme, pour améliorer la lisibilité et rendre plus faciles les modifications. Par exemple, on aurait très bien pu ne pas déclarer de procédures dans le programme précédent, et tout faire dans le programme principal. Cela n'aurait fait que mélanger le cœur du programme (la procédure `Magic`) et les entités annexes comme l'impression et la lecture.

A.2 Quelques éléments de Pascal

A.2.1 Symboles, séparateurs, identificateurs

Les identificateurs sont des séquences de lettres et de chiffres commençant par une lettre. Les identificateurs sont séparés par des espaces, des caractères de tabulation, des retours à la ligne ou par des caractères spéciaux comme `+`, `-`, `*`. Certains identificateurs ne peuvent être utilisés pour des noms de variables ou procédures, et sont réservés pour des *mots clés* de la syntaxe, comme `and`, `array`, `begin`, `end`, `while`, ...

Par convention, il sera commode de commencer les noms de constantes par une majuscule, et le nom d'une variable par une minuscule. Cela peut être fort utile dans un gros programme. Aussi on pourra se servir des majuscules pour un identificateur formé de plusieurs mots, comme `unJoliIdentificateur`. Enfin, il sera permis de déroger à cette règle pour les fonctions d'une lettre.

Certains Pascal ne font pas de distinction entre majuscules et minuscules. Il est fortement conseillé de ne jamais utiliser cette particularité. En effet, il peut être commode de ne pas avoir à taper des lettres majuscules, mais on s'expose très facilement à rendre ainsi les programmes non portables, puisque d'autres Pascal font eux la différence entre majuscules et minuscules (en Pascal Berkeley sur Vax par exemple). On ne saura trop répéter la phrase suivante:

Majuscules et minuscules doivent être considérées comme différentes

A.2.2 Types de base

Les *booléens* ont un type `boolean` prédéfini, qui contient deux constantes `true` et `false`. Les *entiers* ont un type `integer` prédéfini. Les constantes entières sont une suite de chiffres décimaux, éventuellement précédée d'un signe, comme 234, -128, ... Les valeurs extrémales sont `-maxint` et `maxint`. Remarque: Pascal ne suit pas notre convention de faire commencer les constantes par une majuscule. Les *réels* ont un type `real` prédéfini. Les constantes réelles ont deux formats possibles, en notation décimale comme `3.1416`, ou en notation avec exposant comme `3141.6E-3` pour désigner $3141,6 \times 10^{-3}$. Attention: en anglais, on écrit 1.5 plutôt que 1, 5. Le dernier type de base est le type prédéfini

`char` des *caractères*. Une constante caractère est `'a'`, `'b'`, ..., `'+'`, `':'`. Remarque: le caractère apostrophe se note `''''` en doublant l'apostrophe.

A.2.3 Types scalaires

Au lieu de donner des valeurs conventionnelles à des objets symboliques, on peut utiliser les types *énumérés*. Par exemple

```
type Couleur = (Bleu, Blanc, Rouge);
     Sens = (Gauche, Haut, Droite, Bas);
var c,d: Couleur;
    s: Sens;
begin
    c := Bleu;
    s := Droite;
...
end;
```

où `Couleur` est l'énumération des trois valeurs `Bleu`, `Blanc`, `Rouge`. Le type `boolean` est un type énuméré prédéfini tel que:

```
type boolean = (false, true);
```

Les types de base sont tous scalaires. Les types entiers, caractères, énumérés sont aussi dits *types ordinaux*. Une fonction `ord` donne le numéro d'ordre de chaque élément. Ainsi,

$0 =$ `ord(0)` $=$ `ord(Bleu)` $=$ `ord(false)`
$1 =$ `ord(1)` $=$ `ord(Blanc)` $=$ `ord(true)`
$2 =$ `ord(2)` $=$ `ord(Rouge)`
$97 =$ `ord('a')`
$98 =$ `ord('b')`
$99 =$ `ord('c')`
$48 =$ `ord('0')`
$49 =$ `ord('1')`
$50 =$ `ord('2')`

Pour le type ordinal caractère, la valeur de la fonction `ord` donne la valeur entre 0 et 127 dans le code ASCII (American Standard Codes for Information Interchange) du caractère. Attention: `ord('0')` est le code du caractère `'0'`, et est donc différent de `ord(0)`, numéro d'ordre de l'entier 0, qui vaut 0. Il existe aussi une fonction inverse `chr` qui donne le caractère par son code ASCII. Enfin tous les types ordinaux sont munis de deux fonctions `succ` et `pred` qui donnent l'élément suivant ou précédent dans l'énumération. Ces valeurs ne sont pas définies aux bornes.

Sur tout type ordinal, on peut définir un type *intervalle*, par exemple `1..99` pour restreindre le champ des valeurs possibles prises par une variable. En général, Pascal vérifie que la valeur d'une variable sur un tel intervalle reste à l'intérieur. Les intervalles rendent donc la programmation plus sûre. Ils sont nécessaires pour définir les valeurs sur lesquelles varient les indices d'un tableau. Voici quelques exemples d'intervalles:

```
type Minuscule = 'a'..'z';
     Majuscule = 'A'..'Z';
     Chiffre = '0'..'9';
```

```
% more /usr/pub/ascii
| 00 nul| 01 soh| 02 stx| 03 etx| 04 eot| 05 enq| 06 ack| 07 bel|
| 08 bs | 09 ht | 0a nl | 0b vt | 0c np | 0d cr | 0e so | 0f si |
| 10 dle| 11 dc1| 12 dc2| 13 dc3| 14 dc4| 15 nak| 16 syn| 17 etb|
| 18 can| 19 em | 1a sub| 1b esc| 1c fs | 1d gs | 1e rs | 1f us |
| 20 sp | 21  ! | 22  " | 23  # | 24  $ | 25  % | 26  & | 27  ' |
| 28  ( | 29  ) | 2a  * | 2b  + | 2c  , | 2d  - | 2e  . | 2f  / |
| 30  0 | 31  1 | 32  2 | 33  3 | 34  4 | 35  5 | 36  6 | 37  7 |
| 38  8 | 39  9 | 3a  : | 3b  ; | 3c  < | 3d  = | 3e  > | 3f  ? |
| 40  @ | 41  A | 42  B | 43  C | 44  D | 45  E | 46  F | 47  G |
| 48  H | 49  I | 4a  J | 4b  K | 4c  L | 4d  M | 4e  N | 4f  O |
| 50  P | 51  Q | 52  R | 53  S | 54  T | 55  U | 56  V | 57  W |
| 58  X | 59  Y | 5a  Z | 5b  [ | 5c  \ | 5d  ] | 5e  ^ | 5f  _ |
| 60  ` | 61  a | 62  b | 63  c | 64  d | 65  e | 66  f | 67  g |
| 68  h | 69  i | 6a  j | 6b  k | 6c  l | 6d  m | 6e  n | 6f  o |
| 70  p | 71  q | 72  r | 73  s | 74  t | 75  u | 76  v | 77  w |
| 78  x | 79  y | 7a  z | 7b  { | 7c  | | 7d  } | 7e  ~ | 7f del|
```

Figure A.1 : Le code ASCII en hexadécimal

```
     Naturel = 0 .. maxint;
var x: 1..99;
    c: Minuscule;
    d: Chiffre;
    n: Naturel;
```

Attention: le type intervalle peut être trompeur dans une boucle. Si on déclare une variable `i` de type `1..10`, il se peut qu'une erreur de type arrive à l'exécution si `i` est utilisé dans une boucle `while` itérant pour `i` variant de 1 à 10. En effet, en fin de boucle, la variable `i` peut prendre la valeur interdite 11, déclenchant ainsi une erreur de débordement d'intervalle autorisé.

A.2.4 Expressions

Les expressions sont elles aussi de plusieurs types. Les expressions arithmétiques font intervenir les opérateurs classiques sur les entiers `+` (addition), `-` (soustraction), `*` (multiplication), `div` (division entière), `mod` (modulo). On utilise les parenthèses comme en mathématiques standard. Ainsi, si `x` et `y` sont deux variables entières, on peut écrire `3 * (x + 2 * y) + 2 * x * x` pour $3(x+2y)+2x^2$.

De même, les mêmes opérateurs peuvent servir pour les expressions réelles, à l'exception de la division qui se note `/`. Donc si `z` et `t` sont deux variables réelles, on peut écrire `3 * (z + 1) / 2` pour $3(z+1)/2$. Il y a les fonctions `trunc` et `round` de conversion des réels dans les entiers: la première donne la partie entière, la seconde l'entier le plus proche. Réciproquement, les entiers sont considérés comme un sous-ensemble des réels, et on peut écrire librement `3.5 + (x div 2)`.

On peut aussi faire des expressions booléennes, à partir des opérateurs `or`, `and`, `not`. Ainsi si `b` etc sont deux variables booléennes, l'expression

```
(b and not c) or (not b and c)
```

représente le ou-exclusif de `b` et `c`. (On peut aussi l'écrire simplement `b <> c`).

Il existe aussi des opérateurs plus hétérogènes. Ainsi, les opérateurs de comparaison `=`, `<>`, `<=`, `<`, `>`, `>=` rendent des valeurs booléennes. On peut comparer des entiers, des réels, des booléens, des caractères (dans ce dernier cas, l'ordre est celui du code ASCII).

La précédence des opérateurs est relativement naturelle. Ainsi `*` est plus prioritaire que `+`, lui-même plus prioritaire que `=`. Si un doute existe, il ne faut pas hésiter à mettre des parenthèses. Il faut faire attention dans les expressions booléennes, et bien mettre des parenthèses autour des expressions atomiques booléennes, comme dans:

```
if (x > 1) and (y = 3) then ...
```

L'ordre d'évaluation des opérateurs dans les expressions est malheureusement très simple. Ainsi, dans $e_1 \otimes e_2$, on évalue d'abord e_1 et e_2 donnant les valeurs v_1 et v_2, puis on évalue $v_1 \otimes v_2$. Ceci veut donc dire que si l'évaluation de e_1 ou e_2 se passe mal (non terminaison, erreur de type à l'exécution), l'évaluation de $e_1 \otimes e_2$ se passera mal également. Cette remarque sera particulièrement gênante quand on fera de la recherche dans une table `a`, où on voudra écrire typiquement une boucle du genre

```
while (i <= TailleMax) and (a[i] <> v) do ...
```

Cette écriture sera interdite, puisqu'on devra évaluer toujours les deux arguments de l'opérateur `and`, même dans le cas où on finit avec `i` $>$ `TailleMax` et `a[i]` alors indéfini. Pascal, contrairement au langage C, devra tourner autour de cette difficulté avec des booléens, sentinelles ou autres instructions `goto`. Si l'on veut rester portable, il ne faut pas utiliser les particularités de certains Pascal (Think) qui évaluent leurs expressions booléennes de la gauche vers la droite contrairement à la définition standard.

A.2.5 Types tableaux

Les tableaux servent à représenter des vecteurs, matrices, ou autres tenseurs à plusieurs dimensions. Ce sont des structures homogènes, tous les éléments devant avoir un même type. Les indices sont pris dans un type intervalle ou énuméré. Ainsi

```
var v: array [0..99] of integer;
    a: array [1..2, 1..2] of real;
```

définissent `v` comme un vecteur de 100 entiers, et `a` comme une matrice 2×2. Les éléments sont désignés par `v[i]` et `a[j,k]` où $0 \leq$ `i` ≤ 99 et `j`, `k` $\in \{1, 2\}$. Les tableaux sont de taille fixe et n'ont aucune restriction sur le type de leur élément. Donc

```
var b: array [1..2] of array [1..2] of real;
```

désigne un vecteur de 2 éléments dont chaque élément est aussi un même vecteur. On écrira `b[i][j]` pour accéder à l'élément $b_{i,j}$. Donc `a` et `b` sont clairement isomorphes.

A.2.6 Procédures et fonctions

Des exemples de procédures ont déjà été donnés. Voici un exemple de fonction des entiers dans les entiers:

```
function Suivant (x: integer): integer;
begin
    if odd(x) then
        Suivant := 3 * x + 1
    else
        Suivant := x div 2;
end;
```

On peut remarquer que les types des arguments et du résultat de la fonction, sa *signature*, sont définis assez naturellement. Le type du résultat est ce qui distingue une fonction d'une procédure. Curieusement, il n'y a pas d'instruction pour retourner le résultat d'une fonction. La convention est de prendre le nom de la fonction et de lui affecter la valeur du résultat. La fonction précédente renvoie donc $3x+1$ si x est impair, et $\lfloor x/2 \rfloor$ si x est pair. (On peut s'amuser à itérer la fonction et regarder le résultat; on ne sait toujours pas démontrer qu'on finira avec 1 pour tout entier de départ. Par exemple: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1).

Une autre possibilité dans l'exemple du carré magique aurait été de définir la fonction `Even` des entiers dans les booléens

```
function Even (x: integer): boolean;
begin
    Even := not odd(x);
end;
```

qui répond `true` si `x` est pair. Pascal fournit curieusement uniquement la fonction "impair" `odd`.

Certaines fonctions ou procédures (par exemple `Lire` dans l'exemple du carré magique) ne se contentent pas d'utiliser la valeur de leur argument, mais veulent lui fixer une valeur. Techniquement, on dit qu'elles ont *un effet de bord* sur leur argument. Pendant le calcul de la procédure, on veut que le paramètre et la variable qui est effectivement passée en argument soient un même objet. Ainsi, chaque fois qu'on modifiera le paramètre, on modifiera la variable argument. Pour arriver à créer de tels *alias* entre paramètres formels et paramètres réels, on doit signaler que l'argument est appelé par référence en mettant le mot-clé `var`. Dans l'exemple suivant

```
var a, b: integer;

procedure RAZ (var x: integer);
    begin
    x := 0;
    end;

begin
a := 1;
b := 2;
RAZ(a);  (* Instruction1 *)
RAZ(b);  (* Instruction2 *)
end;
```

Avant l'instruction$_1$, la valeur de `a` est 1. Après l'instruction$_1$, la valeur de `a` est 0. Avant l'instruction$_2$, la valeur de `b` est 2. Après l'instruction$_2$, la valeur de `b` est 0. Donc la procédure `RAZ` a bien un effet sur le paramètre effectif de la procédure. Un autre exemple plus utile est

```
var a, b: integer;

procedure Echange (var x, y: integer);
    var z: integer;
    begin
    z := x; x := y; y := z;
    end;

begin
a := 1;
```

```
b := 2;
Echange (a, b);  (* Instruction d'échange *)
end;
```

Après l'instruction d'échange, les valeurs de `a` et `b` sont inversées, l'échange se faisant par permutation circulaire sur les trois variables `x`, `y`, `z` dans la procédure. Il faut donc bien comprendre la distinction entre les deux formes d'arguments des procédures ou fonctions, en appel par valeur ou par référence. Dans le premier cas, seule la valeur de l'argument compte, et on ne change pas l'argument lorsque celui-ci est une variable ou un objet modifiable. Dans le deuxième cas, c'est la référence qui compte et chaque fois que le paramètre de la procédure est modifié, l'argument est aussi modifié. Dans le cas de l'appel par référence, l'argument doit être un objet modifiable, c'est-à-dire une variable, un élément de tableau, ... Il est interdit de passer par référence une constante entière ou tout autre objet qui ne correspond pas à un emplacement dans le programme.

Il y a donc deux types d'appel de procédure ou de fonctions: appel par valeur et appel par référence. Dans le premier cas, à l'entrée dans la fonction ou procédure, la valeur de l'argument est stocké provisoirement et on n'a accès qu'à elle. Dans le deuxième cas, le paramètre de la procédure devient un alias à l'argument de la procédure pendant la durée de l'exécution de la procédure, et, chaque fois qu'on modifiera l'un, on modifiera l'autre.

L'appel par valeur est plus sûr

Il y a d'autre manières pour modifier un objet que de le passer en paramètre par référence à une fonction. On peut tout simplement le modifier directement depuis la procédure, à condition qu'il soit dans un bloc englobant la définition de la procédure. Cette modification par effet de bord n'est alors plus paramétrable.

A.2.7 Blocs et portée des variables

Dans des fonctions ou procédures, on peut définir de nouvelles variables *locales*. En fait, le programme principal est une procédure particulière, quoique la syntaxe soit légèrement différente. Les procédures sont des procédures locales du programme principal. Il en est de même pour toutes les autres déclarations d'étiquettes, de constantes, de variables. Donc après l'en-tête de toute procédure `F`, on peut recommencer un nouveau *bloc*, c'est-à-dire une nouvelle suite de déclarations d'étiquettes, constantes, variables, procédures ou fonctions qui seront locales à `F`.

Le bloc principal, celui qui correspond à tout le programme, peut donc contenir des blocs locaux, qui eux-mêmes peuvent contenir des blocs plus locaux, puisqu'on peut recommencer un bloc dans toute procédure ou fonction. Les blocs peuvent contenir des variables ou procédures de même nom. Si on utilise une variable `x`, il s'agira toujours de la variable déclarée dans le plus petit bloc contenant l'utilisation de cette variable dans le programme.

En général, il est plus élégant de garder les variables aussi locales que possible et de minimiser le nombre de variables globales.

A.2.8 Types déclarés

On peut donner un nom à un type, et le déclarer comme le type `T` du carré magique. Cela peut être utile pour la lisibilité du programme, en localisant les définitions des

espaces sur lesquels varient les objets d'un programme. Mais c'est souvent nécessaire pour une autre raison, car en Pascal

> l'égalité de type est l'égalité de leur nom

Cela signifie que, si on veut par exemple donner en argument à une procédure un tableau, comme il faut s'assurer de la concordance des types entre paramètres de procédure et arguments, on doit déclarer un type correspondant au tableau. Supposons que dans le cas du carré magique, la procédure `Init` prenne la matrice `a` en argument. Alors

```
procedure Init (var x: T; n: integer);
    var i, j: integer;

    begin
    for i := 1 to n do
        for j := 1 to n do
            x[i, j] := 0;
    end;
...
Init(a, n);
```

Déclarer des types pour améliorer la lisibilité du programme doit se faire avec modération, car trop de types déclarés peut au contraire diminuer la lisibilité.

En fait, la concordance de types est légèrement plus compliquée. Certains types sont dits *génératifs*. Deux types sont égaux si la déclaration de leurs parties géneratives est faite au même endroit dans le programme. Tous les constructeurs de types complexes sont génératifs (tableaux, enregistrements, déréférencement). Donc, l'égalité des noms de types déclarés est une bonne approximation de cette règle.

A.2.9 Instructions

Après les déclarations, un bloc contient une suite d'instructions entourée par les mots-clé `begin` et `end`. L'instruction de base est l'*affectation*, comme `x := 1`, ou plus généralement `x :=` e où e est une expression quelconque de même type que `x`. En général, la concordance de type entre partie gauche et partie droite de l'affectation implique que les deux correspondent à la déclaration d'un même type. Pourtant, il y a quelques commodités. La partie droite peut être une expression entière et la gauche réelle, puisque nous avons vu que les entiers sont considérés comme un sous-ensemble des réels. De même, pour un type intervalle, seul le type du domaine support compte, sans oublier les vérifications d'appartenance à l'intervalle.

Une instruction peut être conditionnelle, instruction `if`, et n'être exécutée que si une expression booléenne est vraie. Il y a deux formes d'instructions `if`. L'instruction conditionnelle partielle

`if` e `then` S_1

qui permet de n'exécuter S_1 que si e est vraie. Dans l'instruction conditionnelle complète

`if` e `then` S_1 `else` S_2

on peut exécuter S_2 si e est faux. On peut remarquer que, si S_1 est une instruction conditionnelle partielle, il y a une ambiguïté car on ne sait plus si S_2 est le défaut de cette nouvelle instruction conditionnelle ou de l'instruction plus globale. En Pascal, la convention est d'associer la partie `else` à l'instruction `if` la plus interne. Ainsi

```
if e then if e' then S1 else S2
```

et

```
if e then begin if e' then S1 end else S2
```

ont deux sens différents. L'instruction `case` permet d'éviter une cascade d'instructions `if` et permet de faire un aiguillage selon différentes valeurs d'une expression e de type ordinal. Selon la version de Pascal, il y a un cas `otherwise`, qui est le cas par défaut. Ainsi

```
case e of
v1: S1;
v2: S2;
...
vn: Sn;
otherwise S';
end
```

permet de faire l'instruction S_i si $e = v_i$, ou S' si $e \neq v_i$ pour tout i.

Les autres instructions sont principalement des instructions d'itération, comme les instructions `for`, `while` ou `repeat`. L'instruction `for` permet d'itérer sans risque de non terminaison sur une variable de contrôle qui est incrémentée de 1 ou de -1 à chaque itération. Ainsi

```
for i:= e1 to e2 do S
```

itère $\lceil 0, 1 + \texttt{ord}(e_2) - \texttt{ord}(e_1) \rceil$ fois l'instruction S. De même

```
for i:= e1 downto e2 do S
```

fait pour itérer de e_1 à e_2 en décroissant.

On ne doit pas changer la valeur de la variable de contrôle dans une instruction `for`

En effet, si on modifie cette variable, on s'expose à la non terminaison de l'itération. Remarque: il n'y a que 1 et -1 comme pas d'itération. Si on veut un autre pas d'itération, il faudra multiplier la valeur de la variable de contrôle. Par ailleurs, la valeur de la variable de contrôle à la fin de la boucle `for` est indéfinie, et il est très déconseillé de l'utiliser.

Dans l'instruction `while`,

```
while e do S
```

on itère l'instruction S tant que l'expression booléenne e est vraie. De même dans l'instruction `repeat`,

```
repeat S1;S2; ...Sn until e
```

on effectue la suite des instructions $S_1; S_2; \ldots S_n$ tant que l'expression booléenne e est fausse. Remarque: `repeat` fait au moins une fois l'itération. Donc l'instruction `while` est plus sûre, puisqu'elle permet de traiter le cas où le nombre d'itérations est nul. En outre, il faut bien faire attention que les instructions `while` et `repeat` peuvent ne pas terminer.

Il y a encore quatre types d'instructions: l'instruction composée, l'instruction vide, l'instruction `goto`, et l'appel de procédure. L'appel de procédure permet d'appeler une procédure comme déjà vu dans l'exemple du carré magique. L'instruction composée

`begin` S_1 ; S_2 ; ... S_n `end`

permet de faire les n instructions S_1, S_1, ... S_n en séquence. Ceci peut être particulièrement utile quand on veut par exemple itérer plusieurs instructions dans une instruction `for`, `while`, ou faire plusieurs instructions dans une alternative d'un `if` ou d'un `case`. L'instruction vide est à la fois une commodité syntaxique et une nécessité. Ainsi dans

`begin` S_1 ; S_2 ; ... S_n ; `end`

il y a une $n+1^{\text{ème}}$-instruction vide avant `end`. Ceci est particulièrement commode quand les instructions sont mises sur plusieurs lignes, car on peut facilement insérer une instruction entre S_n et `end` si nécessaire, sans changer le programme. Par ailleurs, l'instruction vide permet de faire des attentes actives d'événements asynchrones. Ainsi l'instruction

```
while not Button do ;
```

permet d'attendre que le bouton de la souris soit enfoncé.

L'instruction `goto` permet de faire des branchements vers des étiquettes. En règle générale, son utilisation est fortement déconseillée, il existe suffisamment d'instructions structurées pour permettre d'en limiter l'utilisation. On peut se brancher avec `goto` vers une étiquette déclarée dans un bloc contenant l'instruction `goto`. Toutefois, on ne peut se brancher n'importe où. En règle générale, on ne peut rentrer dans toute instruction que par le début. Il est donc par exemple impossible de définir une étiquette devant l'alternative d'un `if` ou d'un `case`.

> Un programme est d'autant plus mauvais qu'il contient un grand nombre d'instructions `goto`

L'instruction `goto` a toutefois son utilité. Pascal n'ayant pas d'autre moyen de traiter les cas exceptionnels, il faut utiliser des `goto` dans les cas d'erreurs, comme dans l'exemple du carré magique, ou à la rigueur d'une boucle à deux sorties.

A.2.10 Chaînes de caractères

Les chaînes de caractères sont des tableaux de caractères. Plus exactement ce sont des tableaux compressés (*packed*). Les constantes chaînes de caractères sont de la forme `'Ceci est une belle chaine.'`, comme nous en avons déjà rencontrées dans l'exemple du carré magique. Ainsi une chaîne

`s` de longueur `NMax` se déclare

```
var s: packed array [1..NMax] of char;
```

Le `i`$^{\text{ème}}$ caractère de `s` se trouve en `s[i]`. Les problèmes commencent à se poser quand on veut donner une valeur à `s`. On peut toujours affecter tous les caractères successivements, mais c'est peu commode. On veut pouvoir écrire

```
s := 'Une jolie chai^ne.';
```

En Pascal standard, ceci est impossible. Il faut que la chaîne en partie droite de l'affectation soit de même taille que la chaîne en partie gauche. Il faudrait donc compléter avec des espaces `' '` en supposant `NMax` > 18. De même, si `t` est une autre chaîne, on ne peut écrire `s := t` que si les longueurs de `s` et `t` sont identiques. Tout cela est bien contraignant. En Pascal Berkeley sur Vax, on autorise la longueur de la chaîne en partie gauche à être de longueur supérieure ou égale à celle en partie droite, la chaîne

étant complétée par des espaces. Remarque: cette convention marche aussi si on veut passer une chaîne en argument à une procédure ou fonction. Le paramètre (par valeur) peut être une chaîne de longueur supérieure ou égale à l'argument effectif. Ceci est bien utile si on veut écrire une procédure `Erreur` prenant en argument un message d'erreur. En copiant l'exemple initial du carré magique, on aurait

```
type Message = packed array [1..256] of char;
...
procedure Erreur (s: Message);
    begin
    writeln ('Erreur fatale: ', s);
    goto 999;
    end;

procedure Lire (var n: integer);
    begin
    write('Taille du carre'' magique, svp?::  ');
    readln(n);
    if (n <= 0) or (n > NMax) or Even(n) then
        Erreur('Taille impossible.');
    end;
```

Certains Pascal, notamment sur Macintosh, autorisent un type spécial prédéfini `string` pour les chaînes de caractères, avec des opérations de concaténation, de suppression de sous-chaîne. Si on utilise de telles particularités, on se retrouve avec des programmes non portables. Tout marchera bien sur Macintosh, et non sur Vax (ce qui peut se révéler très gênant pour un gros programme, qui lui ne tournera vite que sur Vax).

> En Pascal, on ne doit utiliser que la partie portable des chaînes de caractères.

Toutefois, les Pascal sur Macintosh n'autorisent la manipulation des chaînes de caractères que si on définit un objet du type `string`. Sinon, on se retrouve avec les contraintes classiques de l'affectation des tableaux. Il faudra donc faire une petite gymnastique entre un programme tournant sur Macinstosh ou sur Vax. Sur Mac, on écrira

```
type Message = string [n];
```

et on changera cette déclaration sur Vax en

```
type Message = packed array [1..n] of char;
```

Le reste du programme reste inchangé, grâce aux déclarations de types.

Enfin, il faudra bien faire attention que Pascal ne distingue pas la constante caractère `'a'` et la chaîne de caractère `'a'` de longueur 1.

A.2.11 Ensembles

Ce sont une des particularités de Pascal. Une variable `e` peut être de type ensemble par la déclaration

```
var e: set of type_ordinal;
```

Les constantes du type ensemble sont `[ ]` pour l'ensemble vide, `[`C_1`,` C_2`,` C_3`..`C_4`,` C_5`]` pour représenter l'ensemble $\{C_1, C_2, C_5\} \cup \{c \mid C_3 \leq c \leq C_4\}$. Les opérations sont les opérations ensemblistes usuelles union (`+` ou `or`), intersection (`*` ou `and`), soustraction (`-` ou `/`), complément (`not`). Il existe une opération hétérogène d'appartenance notée `x in e` qui a un résultat booléen et qui correspond à `x` $\in$ `e`.

Les ensembles doivent être toutefois utilisés avec modération, car la cardinalité maximale des ensembles dépend de la version de Pascal. La règle générale est que l'on peut utiliser des petits ensembles de l'ordre de 32, 64 ou 128 éléments. Il y a une raison bien simple: les ensembles sont souvent représentés en machines par des tableaux de bits donnant leur fonction caractéristique, et les différentes versions de Pascal n'autorisent qu'un certain nombre de mots-machine pour cette représentation.

Les ensembles peuvent se révéler utiles lorsqu'on ne dispose pas du cas défaut dans une instruction `case`. On peut alors écrire

```
if e in [v1, v2, ..., vn] then
    case e of
    v1: S1;
    v2: S2;
    ...
    vn: Sn;
    end
else
    S'
```

A.2.12 Arguments fonctionnels

Les procédures et fonctions peuvent prendre un argument fonctionnel, mais ne peuvent rendre une valeur fonctionnelle. Ceci peut être utile dans certains cas bien spécifiques. Supposons que l'on veuille écrire une procédure `Zero` de recherche d'une racine sur un intervalle donné. On veut pouvoir utiliser cette fonction de la manière suivante

```
function Log10 (x: real): real;
begin
    Log10 := ln(x) / ln (10.0);
end;

begin
    writeln ('le zero de log10 = ', Zero (Log10, 0.5, 1.5));
end;
```

La déclaration de la fonction `Zero` se fait comme suit. Il faut bien remarquer le premier paramètre `f` qui est une fonction des réels dans les réels, les deuxième et troisième arguments étant les bornes de l'intervalle de recherche du zéro de `f`.

```
function Zero (function f(x: real): real;
               a,b: real): real;
const Epsilon = 1.0E-7;
      Nmax  = 100;
var   n : integer;
      m : real;
begin
     n := 1;
     while (abs (b - a) < Epsilon) and (n < Nmax) do
         begin
```

```
            m := (a + b) / 2;
            if (f (m) > 0.0) = (f (a) > 0.0) then
                a := m
            else
                b := m;
            n := n + 1;
            end;
        Zero := a;
end;
```

Les arguments fonctionnels rendent donc l'écriture particulièrement élégante. Il faut bien noter que l'efficacité du programme n'en est pas du tout affectée. (Ce fut une des découvertes de Randell et Russel en 1960 [40]).

A.2.13 Entrées – Sorties

Pascal standard a des entrées très rudimentaires. D'abord on peut lire sur le terminal (ou la fenêtre texte) par la fonction prédéfinie `read`. Cette fonction peut prendre un nombre quelconque d'arguments, qui seront les objets lus successivement. Ainsi

`read(`x_1`, `x_2`, ..., `x_n`)`

équivaut à

`begin read(`x_1`); read(`x_2`); ... read(`x_n`) end`

De même, `readln` permet de sauter jusqu'à la fin de ligne après le dernier argument. Donc

`readln(`x_1`, `x_2`, ..., `x_n`)`

est une abréviation pour

`begin read(`x_1`); read(`x_2`); ... read(`x_n`); readln; end`

Il en est de même pour les procédures `write` et `writeln` d'écriture. Il y a une petite particularité dans `write`: on peut mettre un format après tout argument. Ainsi `write(x:4)` permet d'écrire la valeur de `x` sur 4 chiffres. Dans le cas d'une variable `y` réelle, on peut préciser le nombre de chiffres avant et après la virgule en écrivant `write(y:4:2)`. Il faut faire attention aux formats par défaut, car il se peut qu'un programme soit correct et que le format par défaut ne permette pas d'imprimer les valeurs correctes.

On peut aussi de lire et d'écrire dans des fichiers. Pour leur plus grand malheur, les fichiers font partie du langage. Ainsi, on peut déclarer une variable `f` de type fichier

```
var f: file of type;
    x: type;
...
begin
...
reset (f, 'MonFichier');
read (f, x);
...
close (f);
end;
```

Un fichier en Pascal standard est ouvert par l'instruction `reset`. On associe alors à la variable `f` un véritable fichier du système de fichiers sous-jacent (du Macintosh ou du Vax). Les fonctions `read` ou `write` peuvent prendre un argument de type fichier comme premier argument pour signifier que la lecture ou l'écriture se fera à partir ou sur le fichier correspondant. Par défaut, le terminal (ou la fenêtre texte) sont les fichiers prédéfinis `input` et `output` que l'on peut mettre dans la première ligne du programme. Donc `read(x)` est une abréviation pour `read(input, x)`. De même, `write(x)` est un raccourci pour `write(output, x)`. En Pascal standard, les fichiers sur lesquels on écrit sont ouverts par une procédure différente de `reset`. On écrit alors

```
var f: file of type;
    x: type;
...
begin
...
rewrite (f, 'MonFichier');
write (f, x);
...
close (f);
end;
```

Il faut faire attention que `rewrite` initialise le fichier `f` à vide. Si on veut donc modifier un fichier ou rajouter quelque chose au bout, on doit commencer par le recopier, en insérant les valeurs nouvelles. Cette vision des fichiers date quelque peu, et a des réminiscences des fichiers à accès séquentiels, comme les bandes magnétiques. Aujourd'hui, pratiquement tous les fichiers sont en accès direct, et ce fut une des grandes contributions du système Unix d'uniformiser les accès aux fichiers. Pascal standard autorise aussi à manipuler des fichiers de tout type: on peut avoir des fichiers de caractères, d'entiers, de réels, ... Mais tout le système Unix démontre que l'on peut bien survivre avec seulement des fichiers de caractères. Ce seront donc les plus fréquents, quoique Pascal sur Unix autorise bien sûr à manipuler des fichiers de type plus exotique (qui ne font qu'être codés par des fichiers de caractères).

L'accès des fichiers, sur Macintosh, se fait aussi par `open(f, nomExterne)` qui ouvre un fichier indifféremment en lecture ou écriture. La fonction `seek(f, n)` permet d'aller au $n^{\text{ème}}$ élément du fichier `f`. La fonction `filepos(f)` donne la position courante dans le fichier `f`.

De manière générale, le prédicat `eof(f)` dit si on se trouve à la fin du fichier `f`. Ainsi le programme suivant copie un fichier dans un autre

```
program CopierFichier;

  var f, g: file of char;
      c: char;

begin
    reset(f, 'MonFichier');
    rewrite(g, 'MaCopie');
    while not eof(f) do
    begin
      read (f, c);
      write (g, c);
    end;
end.
```

Figure A.2 : Les boîtes de dialogue

Il existe un type de fichiers caractères spéciaux: les fichiers `text`. Ces fichiers sont des fichiers caractères avec la notion de ligne. Un prédicat `eoln(f)` permet de dire si on se trouve à la fin d'une ligne. Cette convention est peu pratique, mais il faut arriver à cœxister avec elle. Par exemple, `input` et `output` sont des fichiers `text`. Recommençons le programme précédent avec deux fichiers `text`.

```
program CopierFichierText;

  var f,g: text;
      c: char;

begin
    reset(f, 'MonFichier');
    rewrite(g, 'MaCopie');
    while not eof(f) do
    begin
      while not eoln(f) do
        begin
          read (f, c);
          write (g, c);
        end;
      writeln(g);
    end;
end.
```

Remarquons que ce programme suppose que la fin de fichier se rencontre juste après une fin de ligne. Problème: comment programmer la copie de tels fichiers quand on suppose la fin de fichier possible à tout endroit? On peut alors constater que Pascal rend cette programmation particulièrement difficile.

Pascal sur Macintosh autorise quelques appels à des boîtes de dialogues pour rentrer les noms de fichier avec des menus déroulants. Ainsi

```
open(f, OldFileName('Ancien  Fichier ??'));
open(f, NewFileName('Nouveau  Fichier ??'));
```

affiche le message dans une boîte de dialogue avec un menu déroulant dans le deuxième cas. Cela permet de rentrer plus simplement le nom (externe) du fichier que l'on veut associer à `f`.

En Pascal, Wirth a défini aussi la notion de tampon pour un fichier, et les opérations `get` et `put`. Nous ne les utiliserons jamais, puisque `read` et `write` nous suffiront. Wirth

voulait une notion de position courante dans un fichier `f` contenant la valeur `f^`, et il donne la possibilité de mettre la valeur courante du fichier dans le tampon `f^` par l'instruction `get(f)`, ou d'écrire la valeur du tampon dans le fichier par `put(f)`. Ainsi `write(f, 10)` peut s'écrire `f^ := 10; put(f)`. Nous oublierons ces opérations trop atomiques et fortement inspirées par la notion obsolète de fichier séquentiel.

A.2.14 Enregistrements

Les enregistrements (*records* en anglais) permettent de regrouper des informations hétérogènes. Ainsi, on peut déclarer un type `Date` comme suit:

```
type
    Jour = 1..31;
    Mois = (Jan, Fev, Mar, Avr, Mai, Juin, Juil,
            Aou, Sep, Oct, Nov, Dec);
    Annee = 1..2100;
    Date = record
        j: Jour;
        m: Mois;
        a: Annee;
        end;
var
    berlin, bastille: Date;
begin
    berlin.j := 10; berlin.m := Nov; berlin.a := 1989;
    bastille.j := 14; bastille.m := Juil; bastille.a := 1789;
end.
```

Un enregistrement peut contenir des champs de tout type, et notamment d'autres enregistrements. Supposons qu'une personne soit représentée par son nom, et sa date de naissance; le type correspondant sera

```
type
    Personne = record
        nom: Chaine;
        naissance: Date;
        end;
var
     poincare: Personne;
begin
    poincare.nom := 'Poincare';
    poincare.naissance.j := 29;
    poincare.naissance.m := Avr;
    poincare.naissance.a := 1854;
end.
```

Un enregistrement peut avoir une partie fixe et une partie variable. La partie fixe est toujours au début, le variant à la fin. Ainsi si on suppose que les nombres complexes sont représentés en coordonnées cartésiennes ou en coordonnées polaires, on pourra écrire

```
type
    Coordonnees = (Cartesiennes, Polaires);
    Complexe = record
```

```
        case c: Coordonnees of
          Cartesiennes: (re, im: real);
          Polaires: (rho, theta: real);
        end;

var x,y: Complexe;

begin
    x.c := Cartesiennes; x.re := 0; x.im := 1;
    x.c := Polaires; x.rho := 1; x.theta := PI/2;
end.
```

Dans cet exemple, la partie fixe est vide, ou plus exactement réduite au champ `c` représentant l'indicateur du variant. Ce champ peut prendre la valeur `Cartesiennes` ou `Polaires` qui permet de décider quelle variante on veut de la partie variable de l'enregistrement. Ainsi, une rotation de $\pi/2$ s'écrira

```
function RotationPiSurDeux (x: Complexe): Complexe;
    var r: Complexe;
    begin
    if x.c = Cartesiennes then
        begin
        r.re := -x.im;
        r.im := x.re;
        end
    else
        begin
        r.rho := x.rho;
        r.theta := x.theta + PI/2;
        end;
    RotationPiSurDeux := r;
    end;
```

Très peu d'implémentations de Pascal vérifient que le champ d'un variant n'est accédé que si l'indicateur correspondant est positionné de manière cohérente. C'est un des trous bien connus du typage de Pascal, puisqu'on peut voir un même objet comme un réel ou un entier sans qu'il n'y ait une quelconque correspondance entre les deux. On peut même ne pas mettre d'indicateur, et n'avoir aucune trace dans l'enregistrement sur le cas choisi dans la partie variant de l'enregistrement. Il suffit de mettre un type ordinal à la place de l'indicateur. Ainsi les complexes sans indicateur de variant s'écrivent

```
type
    Coordonnees = (Cartesiennes, Polaires);
    Complexe = record
       case Coordonnees of
         Cartesiennes: (re, im: real);
         Polaires: (rho, theta: real);
       end;
```

Alors il n'y a plus l'indicateur `c` et savoir si un nombre complexe est pris en coordonnées cartésiennes ou polaires est laissé complètement à la responsabilité du programmeur et à la logique du programme.

Enfin, il faut signaler l'instruction `with` qui marche souvent avec les enregistrements. Ainsi, si `x` est un nombre complexe,

```
with x do
    begin
    re := 1;
    im := 0;
    end;
```

est une abréviation pour

```
x.re := 1;
x.im := 0;
```

Ici il n'est pas bien clair si l'écriture est plus compacte. Mais dans le cas où le calcul de l'enregistrement `x` est très compliqué, l'écriture peut être commode. Toutefois, cette instruction peut prêter très rapidement à confusion, notamment si plusieurs `with` sont imbriqués. D'ailleurs, il y a une commodité pour écrire

```
with r1, r2, ... do
    S
```

à la place de

```
with r1 do
    with r2 do
        ...
        S
```

Une bonne règle est peut être de ne pas utiliser l'instruction `with`.

A.2.15 Pointeurs

Toutes les données ne sont pas nommées. Il peut y avoir des données créées dynamiquement pour des structures de données complexes: listes, arbres, graphes, ... Nous n'allons pas nous appesantir sur ces structures qui sont expliquées tout au long des chapitres de ce cours. Nous nous contentons de donner la syntaxe des opérations sur les pointeurs. Un pointeur est une référence sur une donnée (souvent un enregistrement, seul type de donnée qui permet de construire des structures de données complexes). Sa déclaration est de la forme

```
type Liste = ^Cellule;
     Cellule = record
         valeur: integer;
         suivant: Liste;
     end;
```

où `Liste` est le type des pointeurs vers des cellules qui sont des enregistrements dont un champ contient une valeur entière, et l'autre un pointeur vers la liste qui suit. Pour déclarer le type `Liste`, une entorse à la règle générale qui consiste à n'utiliser que des objets déjà définis, a dû être faite, puisque `Liste` est un pointeur vers un type `Cellule` qui n'est pas encore défini. Mais comme il faut bien définir un de ces deux types en premier, la règle est de définir toujours le type pointeur d'abord, que l'on doit nommer si on veut pouvoir affecter un objet à un champ du type pointeur, puisque l'égalité des types est l'égalité de leur nom.

Les opérations sur un pointeur sont très simples. On peut tester l'égalité de deux pointeurs, on peut déréférencer un pointeur `p` en désignant la valeur de l'objet pointé en écrivant `p^`. Ensuite il y a la constante `nil` pour tout type pointeur, qui représente le pointeur vide. Puis on peut affecter une valeur à un pointeur en lui donnant la valeur

`nil` ou celle d'un autre pointeur ou en utilisant la procédure prédéfinie `new`. Supposons que `p` soit un pointeur de type `Liste`. L'instruction `new(p)` donne une valeur à `p` qui est un pointeur vers une nouvelle cellule fraîche. Ainsi

```
var l: Liste;

function NewListe (v: integer; s: Liste): Liste;
    var x: Liste;
    begin              (* On crée d'abord un espace mémoire pour *)
    new (x);           (* la nouvelle cellule pointée par x *)
    x^.valeur := v;    (* Puis, on met à jour son champ valeur *)
    x^.suivant := s;   (* et son champ suivant *)
    NewListe := x;
    end;

begin
l := nil;
for i := 100 downto 1 do
     l := Newliste (i*i, l);
end.
```

permet de construire dynamiquement une liste de 100 éléments contenant les 100 premiers carrés. La construction des nouvelles cellules s'est faite dans la procédure `NewListe`, qui appelle la procédure `new` pour créer l'espace nécessaire pour la nouvelle cellule pointée par `x`. La procédure `new` alloue les objets dans une partie de l'espace mémoire de tout programme appelé le *tas* (*heap* en anglais). Typiquement, le tas est tout le reste de l'espace des données d'un programme, une fois que l'on a retiré les variables globales et locales, dites dans la *pile* (*stack* en anglais). Il y a donc deux types de données complètement différents: les données dites statiques qui sont les variables locales et globales, les données dites dynamiques qui sont les variables allouées dans le tas par la procédure `new`.

A titre d'érudition, on peut signaler la procédure `dispose` qui est l'inverse de `new` et permet de rendre l'espace de donnée pointé par l'argument de `dispose` au tas. Malheureusement, dans beaucoup de Pascal, cette procédure ne marche pas (elle marche sur Vax). Aussi, il est possible de donner un deuxième argument à `new`, quand l'objet pointé est un enregistrement avec variant, dont la taille peut dépendre de la valeur prise par l'indicateur de variant. Le deuxième argument est donc la valeur de l'indicateur pour l'enregistrement nouveau sur lequel on veut pointer. Et ainsi de suite pour un troisième argument, si les variants contiennent aussi des variants ....

A.2.16 Fonctions graphiques

Pascal sur Macintosh (et sur Vax Berkeley par l'intermédiaire de l'émulateur terminal TGiX de Philippe Chassignet, cf *Le manuel TGiX, version 2.2*) donne la possibilité de faire très simplement des fonctions graphiques, grâce à une interface très simple avec la bibliothèque QuickDraw du Macintosh. Sur Macintosh, une fenêtre *Drawing* permet de gérer un écran de 512×340 points. L'origine du système de coordonnées est en haut et à gauche. L'axe des x va classiquement de la gauche vers la droite, l'axe des y va plus curieusement du haut vers le bas (c'est une vieille tradition de l'informatique, dure à remettre en cause). En QuickDraw, x et y sont souvent appelés `h` (horizontal) et `v` (vertical). Il y a une notion de point courant et de crayon avec une taille et une couleur courantes. On peut déplacer le crayon, en le levant ou en dessinant des vecteurs par les fonctions suivantes

`MoveTo (x, y)` Déplace le crayon aux coordonnées absolues `x`, `y`.

`Move (dx, dy)` Déplace le crayon en relatif de `dx`, `dy`.

`LineTo (x, y)` Trace une ligne depuis le point courant jusqu'au point de coordonnées `x`, `y`.

`Line (dx, dy)` Trace le vecteur (`dx`, `dy`) depuis le point courant.

`PenPat(pattern)` Change la couleur du crayon: `white`, `black`, `gray`, `dkGray` (*dark gray*), `ltGray` (*light gray*).

`PenSize(dx, dy)` Change la taille du crayon. La taille par défaut est (1, 1). Toutes les opérations de tracé peuvent se faire avec une certaine épaisseur du crayon.

`PenMode(mode)` Change le mode d'écriture: `patCopy` (mode par défaut qui efface ce sur quoi on trace), `patOr` (mode Union, i.e. sans effacer ce sur quoi on trace), `patXor` (mode Xor, i.e. en inversant ce sur quoi on trace).

Certaines opérations sont possibles sur les rectangles. Un rectangle `r` a un type prédéfini `Rect`. Ce type est en fait un *record* qui a le format suivant

```
type
    VHSelect = (V, H);
    Point = record case indicateur of
            0: (v: integer;
                h: integer);
            1: (vh: array [VHSelect] of integer)
            end;

    Rect = record case indicateur of
           0: (top: integer;
               left: integer;
               bottom: integer;
               right: integer);
           1: (topLeft: Point;
               botRight: Point);
           end;
```

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.

Dans le fichier `<MacLib.h>` du *directory* `/usr/local/pascal/vax` sur Vax, on trouve toutes les signatures des fonctiosn de QuickDraw, qui sont par ailleurs définies dans le document *Inside The Macinstosh*. Voici quelques exemples. La procédure suivante permet de lire les coordonnées d'un point, quand on appuie sur le bouton de la souris.

```
procedure GetXY (var x, y: integer);
    const N = 2;
    var r: Rect; p: Point;
    begin
    while not Button do    (* On attend que le bouton de la souris soit enfoncé *)
        ;
    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 *)
    while Button do        (* On attend que le bouton de la souris soit relâché *)
        ;
    end;
```

Mais la lecture est souvent plus commode sur le front montant.

```
procedure GetXY (var x, y: integer);
    const N = 2;
    var r: Rect; p: Point;
    begin
    while not Button do    (* On attend que le bouton de la souris soit enfoncé *)
        ;
    while Button do        (* On attend que le bouton de la souris soit 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 *)
    end;
```

Un exemple plus amusant est le programme qui fait rebondir une balle dans un rectangle première étape vers un *pong*.

```
program Pong;
    const
        C = 5;                (* Le rayon de la balle *)
        X0 = 5; X1 = 250;
        Y0 = 5; Y1 = 180;
    var
        x, y, dx, dy: integer;
        r, s: Rect;
        i: integer;
    procedure GetXY (var x, y: integer);
        begin
        ...
        end;
begin
SetRect(s, 50, 50, X1 + 100, Y1 + 100);
SetDrawingRect(s);            (* Pour ne pas avoir à positionner *)
ShowDrawing;                  (* manuellement la fenêtre Drawing *)
SetRect(s, X0, Y0, X1, Y1);
FrameRect(s);                 (* Le rectangle de jeu *)
GetXY(x, y);                  (* On note les coordonnées du pointeur *)
dx := 1;                      (* La vitesse initiale *)
dy := 1;                      (* de la balle *)
while true do
    begin
    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) or (x + C >= X1 - 1) then
        dx := -dx;
    y := y + dy;
    if (y - C <= Y0 + 1) or (y + C >= Y1 - 1) then
        dy := -dy;
    for i := 1 to 500 do
        ;                     (* On temporise *)
    InvertOval(r);            (* On efface la balle *)
```

```
        end;
    end.
```

A.3 Syntaxe BNF de Pascal

Ce qui suite est une syntaxe sous forme BNF (*Backus Naur Form*). Chaque petit paragraphe est la définition souvent récursive d'un fragment de syntaxe dénommé par le nom (malheureusement en anglais). Chaque ligne correspond à différentes définitions possibles. L'indice *optional* sera mis pour signaler l'aspect facultatif de l'objet indicé. Certains objets (*token*) seront supposé prédéfinis: *empty* pour l'objet vide, *identifier* pour tout identificateur, *integer* pour toute constante entière, ... La syntaxe du langage ne garantit pas la concordance des types, certaines phrases pouvant être syntaxiquement correctes, mais fausses pour les types.

pascal-program:
`program` *identifier program-heading*$_{opt}$ `;` *block* `.`

program-heading:
`(` *identifier-list* `)`

identifier-list:
identifier
identifier-list `,` *identifier*

block:
block1
label-declaration `;` *block1*

block1:
block2
constant-declaration `;` *block2*

block2:
block3
type-declaration `;` *block3*

block3:
block4
variable-declaration `;` *block4*

block4:
block5
proc-and-func-declaration `;` *block5*

block5:
`begin` *statement-list* `end`

label-declaration:
`label` *unsigned-integer*
label-declaration `,` *unsigned-integer*

constant-declaration:
`const` *identifier* `=` *constant*
constant-declaration `;` *identifier* `=` *constant*

type-declaration:
`type` *identifier* `=` *type*
type-declaration `;` *identifier* `=` *type*

variable-declaration:
`var` *variableid-list* `:` *type*
variable-declaration `;` *variableid-list* `:` *type*

variableid-list:
identifier
variableid-list `,` *identifier*

constant:
integer
real
string
constid
`+` *constid*
`-` *constid*

type:
simple-type
structured-type
`^` *typeid*

simple-type:
`(` *identifier-list* `)`
constant `..` *constant*
typeid

structured-type:
`array [` *index-list* `] of` *type*
`record` *field-list* `end`
`set of` *simple-type*
`file of` *type*
`packed` *structured-type*

index-list:
simple-type
index-list `,` *simple-type*

field-list:
fixed-part
fixed-part `;` *variant-part*
variant-part

fixed-part:
record-field
fixed-part `;` *record-field*

record-field:
empty
fieldid-list `:` *type*

fieldid-list:
identifier
fieldid-list `,` *identifier*

variant-part:
`case` *tag-field* `of` *variant-list*

tag-field:
typeid

identifier `:` *typeid*

variant-list:
variant
variant-list `;` *variant*

variant:
empty
case-label-list `:` `(` *field-list* `)`

case-label-list:
constant
case-label-list `,` *constant*

proc-and-func-declaration:
proc-or-func
proc-and-func-declaration `;` *proc-or-func*

proc-or-func:
`procedure` *identifier* $parameters_{opt}$ `;` *block-or-forward*
`function` *identifier* $parameters_{opt}$ `:` *typeid* `;` *block-or-forward*

block-or-forward:
block
`forward`

parameters:
`(` *formal-parameter-list* `)`

formal-parameter-list:
formal-parameter-section
formal-parameter-list `;` *formal-parameter-section*

formal-parameter-section:
parameterid-list `:` *typeid*
`var` *parameterid-list* `:` *typeid*
`procedure` *identifier* $parameters_{opt}$
`function` *identifier* $parameters_{opt}$ `:` *typeid*

parameterid-list:
identifier
parameterid-list `,` *identifier*

statement-list:
statement
statement-list `;` *statement*

statement:
empty
variable `:=` *expression*
`begin` *statement-list* `end`
`if` *expression* `then` *statement*
`if` *expression* `then` *statement* `else` *statement*
`case` *expression* `of` *case-list* `end`
`while` *expression* `do` *statement*
`repeat` *statement-list* `until` *expression*
`for` *varid* `:=` *for-list* `do` *statement*
procid
procid `(` *expression-list* `)`
`goto` *label*

`with` *record-variable-list* `do` *statement*
label `:` *statement*

variable:
identifier
variable `[` *subscript-list* `]`
variable `.` *fieldid*
variable `^`

subscript-list:
expression
subscript-list `,` *expression*

case-list:
case-label-list `:` *statement*
case-list `;` *case-label-list* `:` *statement*

for-list:
expression `to` *expression*
expression `downto` *expression*

expression-list:
expression
expression-list `,` *expression*

label:
unsigned-integer

record-variable-list:
variable
record-variable-list `,` *variable*

expression:
expression relational-op additive-expression
additive-expression

relational-op: one of
`<` `<=` `=` `<>` `=>` `>`

additive-expression:
additive-expression additive-op multiplicative-expression
multiplicative-expression

additive-op: one of
`+` `-` `or`

multiplicative-expression:
multiplicative-expression multiplicative-op unary-expression
unary-expression

multiplicative-op: one of
`*` `/` `div` `mod` `and` `in`

unary-expression:
unary-op unary-expression
primary-expression

unary-op: one of
`+` `-` `not`

primary-expression:

variable
unsigned-integer
unsigned-real
string
`nil`
funcid `(` *expression-list* `)`
`[` *element-list* `]`
`(` *expression* `)`

element-list:
empty
element
element-list `,` *element*

element:
expression
expression `..` *expression*

constid:
identifier

typeid:
identifier

funcid:
identifier

procid:
identifier

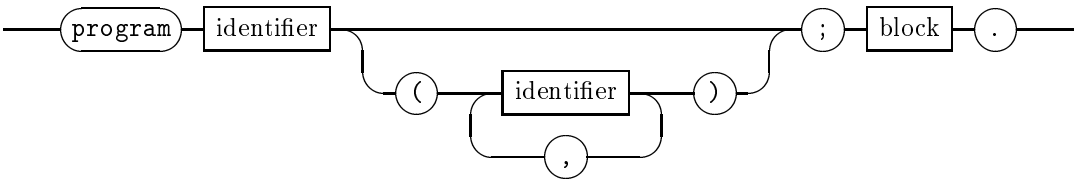
fieldid:
identifier

varid:
identifier

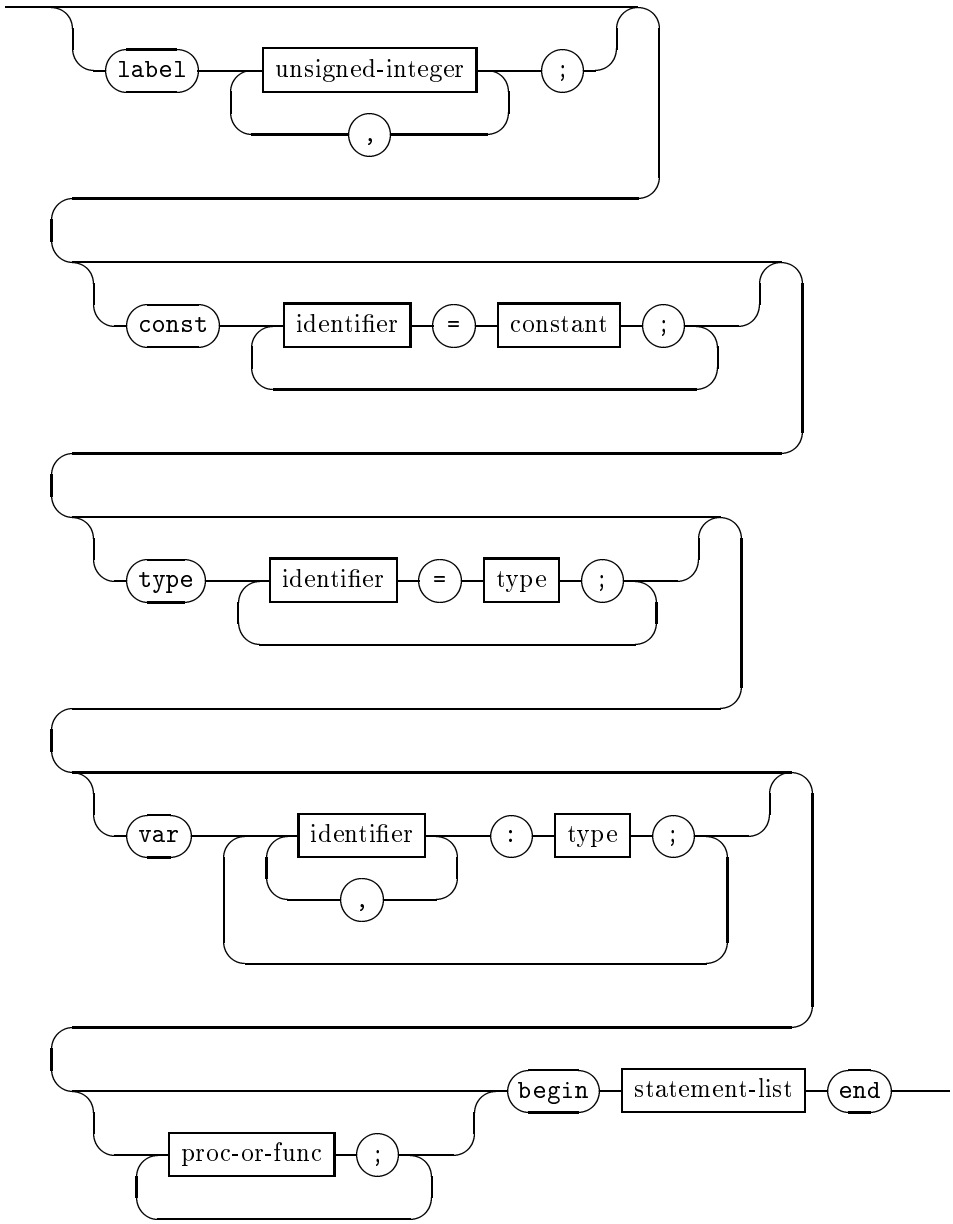
empty:

A.4 Diagrammes de la syntaxe de Pascal

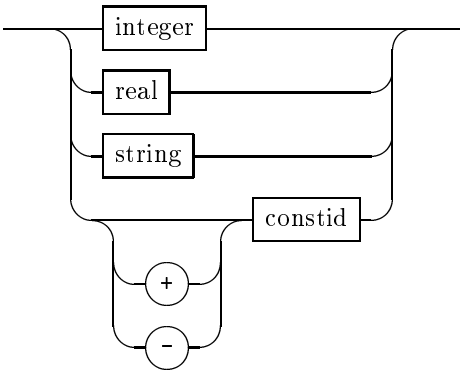
pascal-program



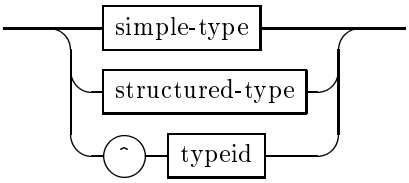
block


label
unsigned-integer
;
,
const
identifier
=
constant
;
type
identifier
=
type
;
var
identifier
:
type
;
,
begin
statement-list
end
proc-or-func
;

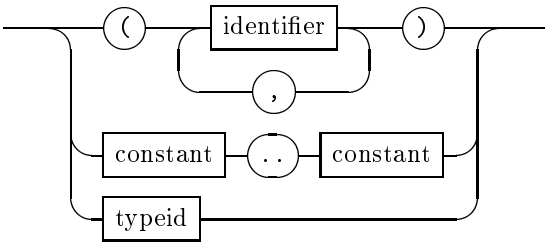
constant



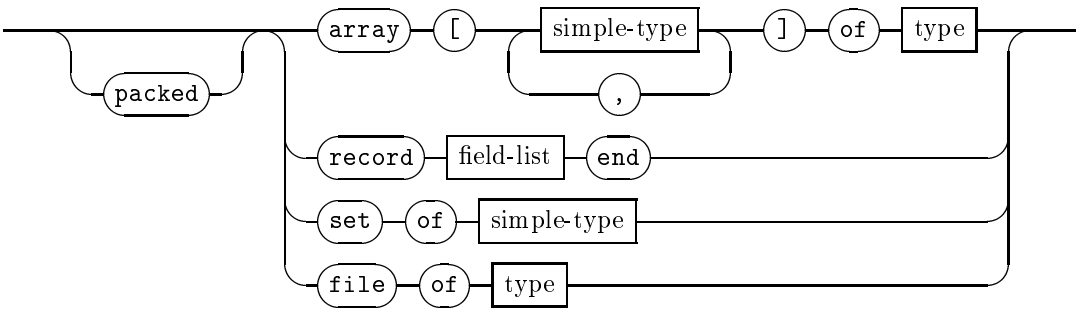
type



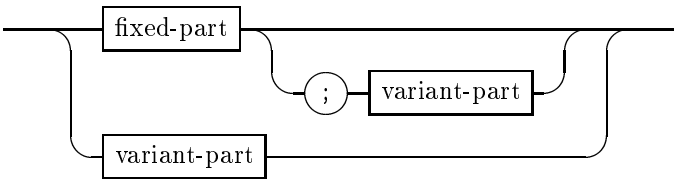
simple-type



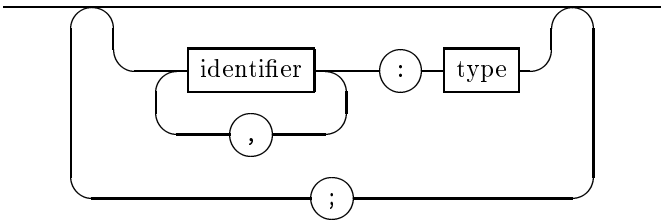
structured-type



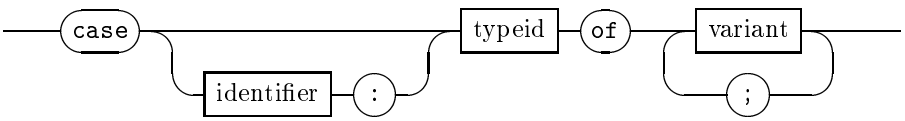
field-list



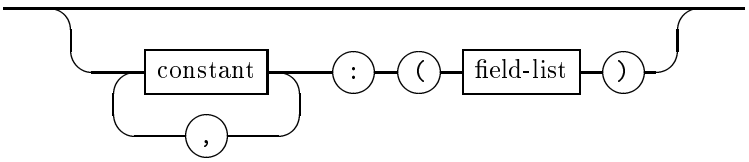
fixed-part



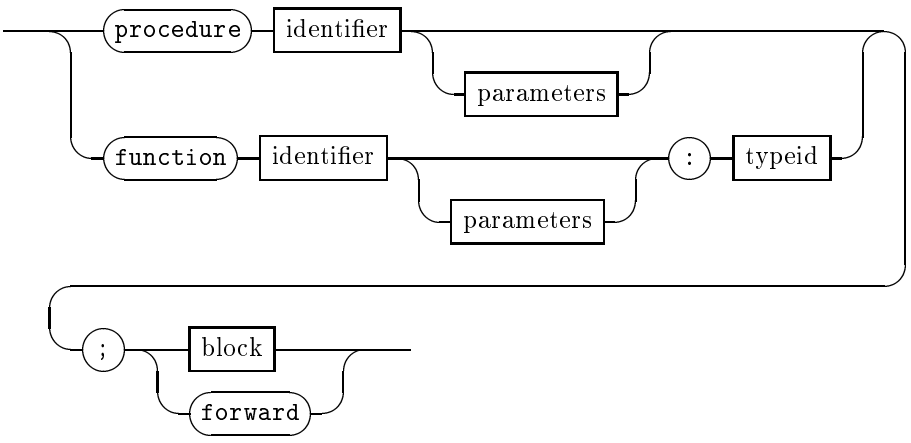
variant-part



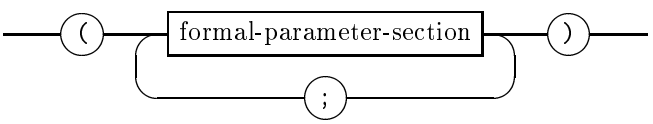
variant



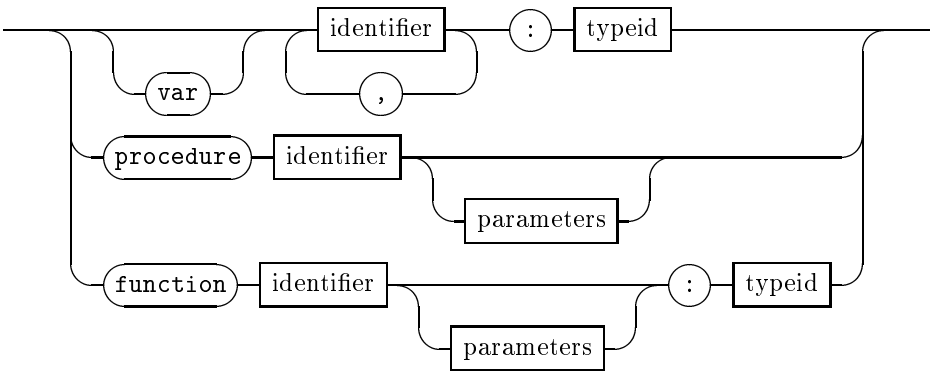
proc-or-func



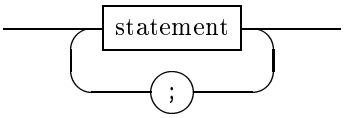
parameters



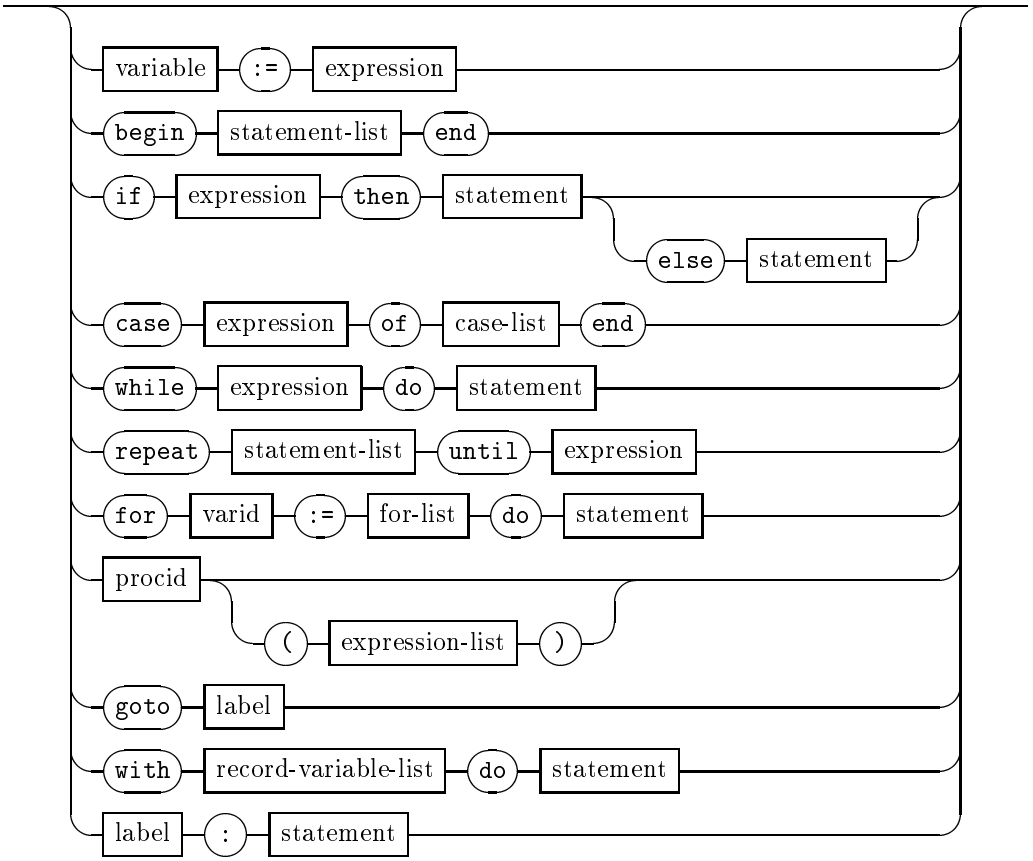
formal-parameter-section



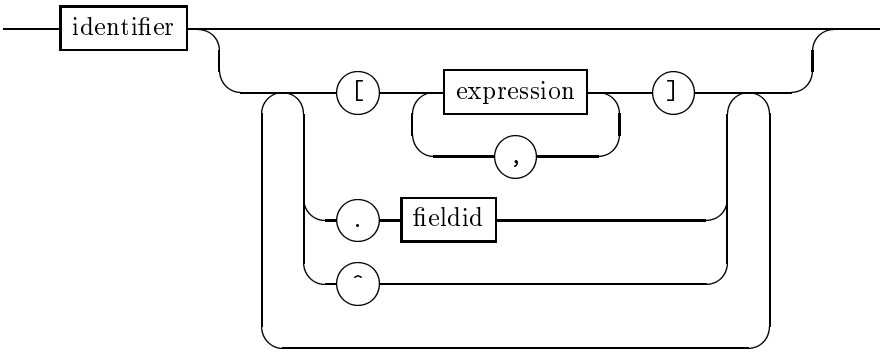
statement-list



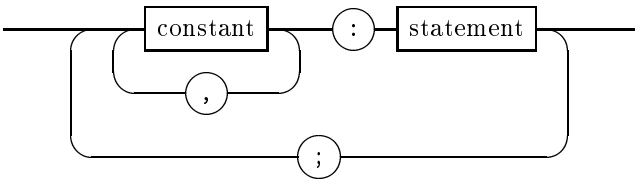
statement



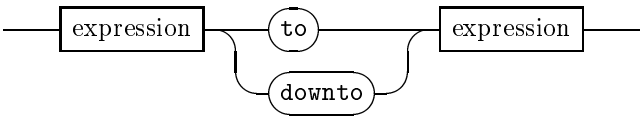
variable



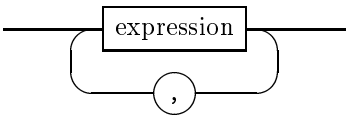
case-list



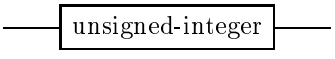
for-list



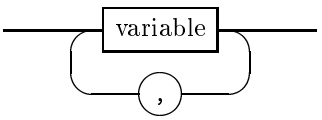
expression-list



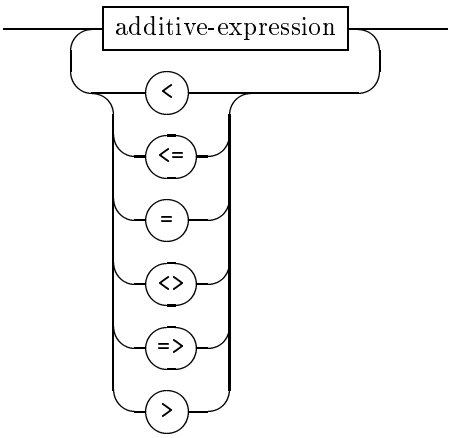
label



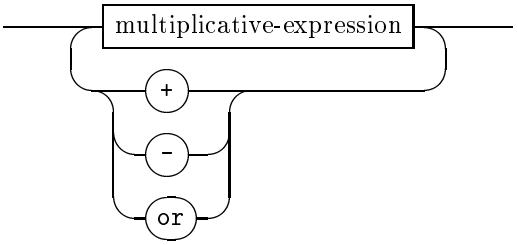
record-variable-list



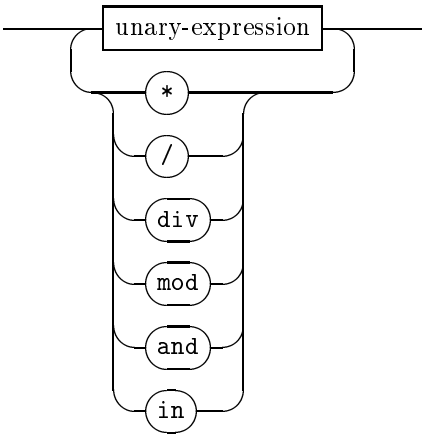
expression



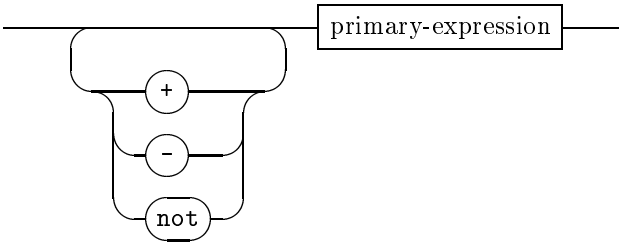
additive-expression



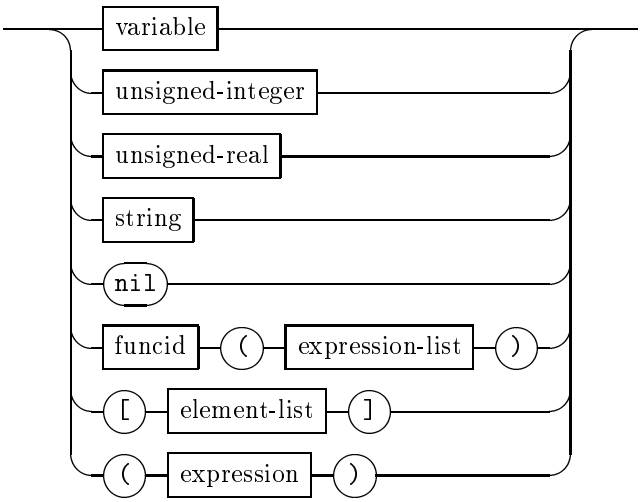
multiplicative-expression



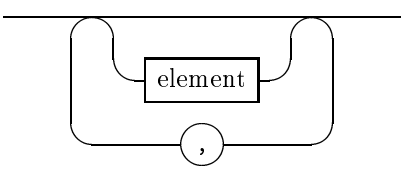
unary-expression



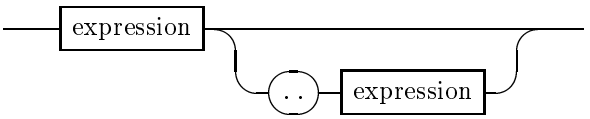
primary-expression



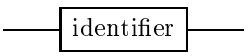
element-list



element

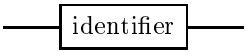


constid

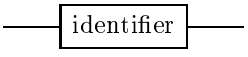


typeid

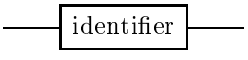
funcid



procid



fieldid



varid

Annexe B

Le langage C

Le langage C a été conçu par Kernighan et Ritchie pour écrire le système Unix dans un langage portable. Il provient de BCPL [41, 42] et en est une version typée. Son typage est clairement défini, surtout dans la version ANSI. Il est toutefois orienté vers la programmation système, et permet donc de faire facilement des conversions de type. Le langage est très populaire à présent, quoiqu'ancien; il est très bien décrit dans le livre de Kernighan et Ritchie [21].

B.1 Un exemple simple

Reprenons l'exemple du carré magique, et transcrivons le en C.

```
#include <stdio.h>

#define N  100

int     a[N][N];
int     n;

void Init (int n)
{
    int i, j;

    for (i = 0 ; i < n; ++i)
        for (j = 0; j < n; ++j)
            a[i][j] = 0;
}

void Magique (int n)
{
    int i, j, k;

    i = n - 1; j = n / 2;
    for (k = 1; k <=  n * n; ++k) {
        a[i][j] = k;
        if ((k % n) == 0)
            i = i - 1;
        else {
            i = (i + 1) % n;
            j = (j + 1) % n;
```

```
        }
    }
}

void Erreur (char s[])
{
    printf ("Erreur fatale: %s\n", s);
    exit (1);
}

void Lire (int *n)
{
    printf ("Taille du carre' magique, svp?::  ");
    scanf ("%d", n);
    if ((*n <= 0) || (*n > N) || (*n % 2 == 0))
        Erreur ("Taille impossible.");
}

void Imprimer (int n)
{
    int i, j;

    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j)
            printf ("%4d ", a[i][j]);
        printf ("\n");
    }
}

int main ()
{
    Lire(&n);
    Init(n);        /* Cette procédure est inutile */
    Magique(n);
    Imprimer(n);
    return 0;
}
```

D'abord, on remarque qu'un programme est une suite linéaire de procédures. Par convention, celle dont le nom est `main` est le point de départ du programme. Une autre toute première remarque est qu'un programme commence souvent par des lignes bizarres démarrant par le caractère `#`. Ce sont des instructions au préprocesseur C. En effet, toute compilation est précédée d'une passe où ces lignes sont traitées. La première ligne

```
#include <stdio.h>
```

dit d'inclure en tête du programme la définition des entrées-sorties standard (par exemple `printf`, ou `scanf` que l'on verra plus tard). La deuxième ligne donne une définition de `N`. C'est la manière traditionnelle en C de définir les constantes. Il est fréquent d'écrire les constantes avec des majuscules uniquement. Mais, nous adopterons la même convention qu'en Pascal, et essaierons de simplement commencer les constantes par une majuscule, et toujours les variables par une minuscule. C fait heureusement la distinction entre majuscules et minuscules.

Une deuxième remarque est de constater que les variables sont déclarées avec une syntaxe différente de Pascal. Le principe est d'écrire un type de base d'une expression utilisant la variable. Ainsi pour le tableau `a`, on écrit le type `int` (entier) de ses éléments `a[i][j]`. De même pour la variable entière `n`. On remarquera qu'un tableau ne peut avoir qu'une seule dimension, et notre matrice a doit être déclarée comme un tableau de tableaux. On ne peut mettre que le nombre d'éléments, et les indices `i`, `j` de `a` varient sur l'intervalle $[0, \texttt{N} - 1]$. Il n'y a donc pas la possibilité de faire démarrer les indices à 1 ou toute autre valeur arbitraire.

Les procédures et fonctions ne diffèrent que par le type de leur résultat, `void` pour les procédures. Maintenant, considérons avec attention la signature de la procédure `Init`. Elle a un argument entier `n`. La procédure `Init` définit un bloc de déclarations et d'instructions, dans cet ordre, compris entre deux accolades. Elle utilise deux variables locales entières `i` et `j` qui n'existent que pendant la durée d'activation de la procédure. Les instructions sont deux boucles `for` imbriquées qui initialisent tous les éléments de `a` à 0. On peut remarquer que, contrairement à Pascal, le symbole d'affectation est malheureusement `=`, sous principe qu'il est plus court à taper que `:=`, mais c'est la source de nombreux problèmes pour les débutants. Il faut donc bien comprendre que le symbole égal (test d'égalité) est écrit `==`, et que le symbole `=` est l'opérateur d'affectation (comme en Fortran!). Les boucles `for` ont trois champs séparés par point-virgule: l'expression d'initialisation, le test de fin, l'expression d'itération. Par initialisation, on entend ce qui sera fait inconditionnellement avant la première itération. Par test de fin, il faut comprendre que l'itération sera faite tant que ce test sera vrai. Par expression d'itération, on signifie l'expression qui est évaluée à la fin de chaque itération.

La procédure `Magique` est fondamentalement identique à celle que nous avions pour Pascal (voir page 198). On tient simplement compte du fait que les indices ont une valeur comprise entre 0 et `n`-1. Remarque: `%` représente l'opérateur modulo, `++` l'opération d'incrémentation par 1. Une remarque plus fine est le point-virgule avant le `else` qui ferait hurler tout compilateur Pascal. 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`. Il faut donc simplement comprendre que le point-virgule fait partie de l'instruction, alors qu'en Pascal c'est un délimiteur d'instructions.

La procédure `Erreur` prend comme argument une chaîne de caractères `s`, c'est-à-dire un tableau de caractères, dont on ne connaît pas la longueur. Nous verrons plus tard les petites subtilités cachées derrière cette déclaration assez naturelle. `Erreur` imprime la chaîne `s` avec le format donné en premier argument de la procédure d'impression formattée `printf`. Un format est une chaîne de caractères où se trouvent quelques trous, indiqués par le symbole `%`, et remplacés par les arguments suivants de `printf`. Ici, `%s` dit que l'argument suivant est une chaîne de caractères. L'impression sera donc constituée des caractères `Erreur fatale: `, suivis par la chaîne `s`, et du caractère `\n` (*new-line*, aussi appelé *linefeed*, de code ASCII 10). Enfin, la procédure fait un appel à la fonction standard `exit` de la librairie C qui arrête l'exécution du programme avec un code d'erreur (0 voulant dire arrêt normal, tout autre valeur un arrêt anormal).

La procédure `Lire` donne une valeur à `n`. Il faut bien faire attention à sa signature. L'argument n'est pas un entier `n`, mais un pointeur `n` sur un entier. Il n'y a pas d'appel par référence comme en Pascal, tout n'est qu'appel par valeur. Si on veut modifier un paramètre comme dans la procédure `Lire`, on devra passer la valeur de la référence à la variable passée en argument. C'est ce pointeur que l'on retrouve dans l'argument de `Lire`. La valeur pointée par un objet est obtenue avec l'opérateur préfixe `*`. Pour définir un pointeur sur un entier, on utilise donc simplement la notation `int *n` qui dit que

la valeur pointée par `n` est entière. `Lire` imprime donc une question pour demander la taille voulue du carré magique. La procédure `scanf` de lecture formattée lit la valeur des arguments suivants. Ici on a `%d` pour indiquer que l'argument suivant est un entier décimal. Et on teste si la valeur lue est correcte. A nouveau, on remarque qu'il faut déréférencer le pointeur `n` pour avoir sa valeur, et que le symbole d'égalité est `==` et non `=`. Le prédicat est entre parenthèses, on n'a donc pas besoin du mot-clé `then`. Si une erreur se produit, on appelle `Erreur` avec le message d'erreur correspondant. Les chaînes de caractères sont encadrées par des guillemets et non par des apostrophes.

La procédure `Imprimer` est une imbrication déjà vue de deux itérations. On peut remarquer le format `"%4d "` pour signifier que le nombre entier `a[i][j]` sera imprimé sur 4 caractères cadré à droite.

Enfin, le programme principal `main` contient un appel à `Init` inutile, mais laissé pour des raisons pédagogiques. Remarquons que les commentaires sont compris entre les séparateurs `/*` et `*/`.

Pour résumer, C apparaît pour le moment comme peu différent de Pascal. Il est un peu plus plat dans la structure de ses procédures ou programmes qui ne peuvent contenir d'autres procédures. Il n'a malheureusement pas d'appel par référence (c'est réparé en C++). Il a la même contrainte que Pascal de définir les objets avant leur utilisation. Nous essaierons dans ce cours d'utiliser C comme Pascal. Il est toutefois possible d'écrire des programmes incompréhensibles pour un pascalien en C. Avec l'avènement des machines modernes (en particulier des machines RISC qui réduisent le nombre d'instructions), ces programmes sont inutiles et ne sont en fait que de vieilles réminiscences de la période du Vax et du pdp 11.

B.2 Quelques éléments de C

B.2.1 Symboles, séparateurs, identificateurs

Les identificateurs sont des séquences de lettres et de chiffres commençant par une lettre. Les identificateurs sont séparés par des espaces, des caractères de tabulation, des retours à la ligne ou par des caractères spéciaux comme `+`, `-`, `*`. Certains identificateurs ne peuvent être utilisés pour des noms de variables ou procédures, et sont réservés pour des *mots clés* de la syntaxe, comme `int`, `char`, `for`, `while`, ...

B.2.2 Types de base

Les entiers ont le type `int`, `short` ou `long`. On utilise principalement le premier, le second est encore un beau reste du pdp11. Autrefois, les entiers pouvaient être représentés sous 32 bits ou 16 bits, les seconds étant plus efficaces que les premiers. Aujourd'hui toutes les machines ont des processeurs 32 bits, et donc tous les entiers ont le type `int`. Toutefois, en Think Pascal, les entiers sont encore sur 16 bits. Les constantes sont des nombres décimaux avec signe. Attention: il y a 2 conventions bien spécifiques sur les nombres entiers: les nombres commençant par `0` sont des nombres octaux, les nombres précédés par `0x` sont des nombres hexadécimaux. Ainsi `0377` et `0xff` valent 255, `015` et `0x0d` valent 13. De même, sur une machine 32 bits, `0xffffffff` vaut -1. Les constantes entières longues sont de la forme `1L`, `-2L`; les constantes *non-signées* de la forme ont le suffixe `U`. Ainsi `0xffUL` est 255 long non-signé.

Les réels ont le type `float` ou `double`. Ce sont des nombres flottants en simple ou double précision. Les constantes sont en notation décimale `3.1416` ou en notation avec exposant `31.416e-1`.

Les caractères sont de type `char`. Les constantes sont écrites entre apostrophes, comme `'A'`, `'B'`, `'a'`, `'b'`, `'0'`, `'1'`, `' '`. Le caractère apostrophe se note `'\''`, et plus généralement il y a des conventions pour des caractères fréquents, `'\n'` pour *newline*, `'\r'` pour retour-charriot, `'\t'` pour tabulation, `'\\'` pour \. On peut aussi écrire un caractère par son code ASCII `'\0'` pour le caractère nul (code 0), ou `'\012'` pour *newline*.

B.2.3 Types scalaires

En C ANSI, on peut déclarer des objets de type énuméré. Ainsi

```
enum Boolean {False, True};
enum Couleur {Bleu, Blanc, Rouge};
enum Sens    {Gauche, Haut, Droite, Bas};

enum Boolean b;
enum Couleur c,d;
enum Sens s;
...
    b = True;
    s = Haut;
    if (c == Rouge) ...
```

Les trois premières lignes donnent un sens aux énumérations `Boolean`, `Couleur` et `Sens`, et donnent les valeurs entières 0, 1 pour `False` et `True`, 0, 1, 2 pour `Bleu`, `Blanc` et `Rouge`, et les valeurs 0, 1, 2, 3 pour `Gauche`, `Haut`, `Droite` et `Bas`. Ceci est donc équivalent à écrire

```
#define False  0
#define True   1
#define Bleu   0
#define Blanc  1
#define Rouge  2
#define Gauche 0
...
```

On peut fixer la valeur des objets dans une énumération. Ainsi

```
enum Escapes {TAB = '\t', NEWLINE ='\n', RETURN = '\r'};
enum Mois {Jan = 1, Fev, Mar, Avr, Mai, Juin,
           Juil, Aou, Sep, Oct, Nov, Dec};
```

Les noms des constantes doivent être distincts dans toutes les énumérations. Enfin, on peut regrouper si l'on veut la déclaration de l'énumération et de quelques variables de ce type en écrivant

```
enum Couleur {Bleu, Blanc, Rouge} c,d;
```

Il n'y a pas de type intervalle en C. Cependant, il y a un type qui n'existe pas en Pascal, les entiers non-signés. Ainsi `unsigned int x` veut dire que les entiers sont pris entre 0 et $2^{32}-1$ sur une machine 32 bits, au lieu de $2^{31}-1$ avec bit de signe. Le préfixe `unsigned` peut se mettre devant les caractères aussi, ce qui interdira l'extension de leur bit de signe (sur 8 bits). Mais ce point sera vu plus tard (voir section B.2.4).

B.2.4 Expressions

Expressions élémentaires

Les expressions arithmétiques s'écrivent comme en Pascal. Seuls quelques opérateurs diffèrent surtout par leur syntaxe. Les opérateurs arithmétiques sont `+`, `-`, `*`, `/`, et `%` pour modulo. Les opérateurs logiques sont `>`, `>=`, `<`, `<=`, `==` et `!=` pour faire des comparaisons (le dernier signifiant $\neq$). Plus intéressant, les opérateurs `&&` et `||` permettent d'évaluer de la gauche vers la droite un certain nombre de conditions (à la différence de Pascal qui évalue les deux côtés des connecteurs logiques *et*, *ou*). Par définition, la valeur vraie d'une expression logique est 1, et faux vaut 0. En fait, plus généralement, toute valeur non nulle désigne la valeur vraie. La négation est représentée par l'opérateur `!`. Ainsi

```
(i < N) && (a[i] != '\n') && !exception
```

donnera la valeur 1 si `i` $<$ `N` et si `a[i]` $\neq$ *newline* et si `exception` $= 0$. Son résultat sera 0 si `i` $\geq$ `N` ou si `i` $<$ `N` et `a[i]` $=$ newline, ... Les opérateurs `&&` et `||` sont les seuls opérateurs dont l'ordre d'évaluation est bien précisé: ils s'évaluent de la gauche vers la droite.

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 autorisée (comme en Pascal) et 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 B.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*)

```
int atoi (char s[])
{
     int i, n;

     n = 0;
     for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
         n = 10 * n + (s[i] - '0');
     return n;
}
```

On peut donc remarquer que `s[i] - '0'` permet de calculer l'entier qui représente la différence dans le code ASCII entre `s[i]` et `'0'`. En C, le résultat de la conversion d'un caractère en un entier est laissé dépendant de la machine, pour ce qui est de l'extension de signe. Seuls les caractères imprimables sont sûrs de ne pas changer de signe et donc de valeur, lors de leur conversion.

L'opérateur `=` d'affectation étant un opérateur comme les autres dans les expressions, il subit 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 reconverti dans le type de l'expression à gauche de l'affectation.

Attention: dans les appels de fonctions, il y a en fait une opération similaire à une affectation pour passer les arguments, et donc des conversions implicites des arguments sont possibles. Pour éviter tout ennui, il faut déclarer la signature de la fonction avant son utilisation. Il n'est alors pas nécessaire d'avoir défini toute la fonction. Seul le

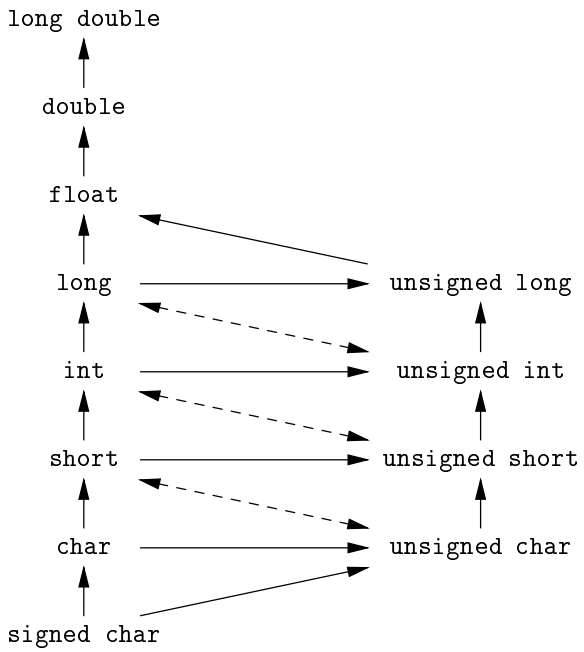


Figure B.1 : Conversions en C

type, donné par la notation des prototypes (cf. la section B.2.6), suffit. Sinon, il peut se passer des catastrophes en fonction du degré ANSI du compilateur. Il faut donc appliquer rigoureusement la règle suivante

> En C, on doit toujours déclarer les types des fonctions avant leur utilisation.

Les conversions suivent la figure B.1. Pour toute opération, on convertit toujours au le plus petit commun majorant des types des opérandes. Les pointillés montrent les relations qui dépendent de la machine. Enfin, des conversions explicites sont aussi possibles, et recommandées dans le doute. On peut les faire par l'opération de cœrcion (*cast*) suivante

(*type-name*) *expression*

L'expression est alors convertie dans le type indiqué entre parenthèses devant l'expression. Par exemple, la racine carrée de la bibliothèque mathématique dans `<math.h>` attend un argument `double`. Donc si `n` est entier, sa racine carrée est obtenue par

```
sqrt((double) n)
```

A nouveau, en l'absence de définition de prototype, la conversion par défaut aura lieu, laissant `n` invariant, et un résultat incohérent résultera. Les définitions prototype de la bibliothèque mathématique sont définies dans `<math.h>` et l'insertion de la ligne

```
#include <math.h>
```

assurera l'insertion des définitions prototype de toute la bibliothèque mathématique. Dans l'exemple précédent, c'est la valeur de l'expression `n` qui est convertie, et non la

variable `n`. Beaucoup d'autres prototypes se trouvent dans `<stdlib.h>`. Un bon exemple de conversion est le générateur portable de nombres aléatoires de la bibliothèque C standard

```
unsigned long int next = 1;

/* Pour avoir un nombre aléatoire entre 0 et 32767 = 2^15 - 1 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* Pour initialiser le générateur de nombres aléatoires. */
void srand(unsigned int seed)
{
    next = seed;
}
```

Affectation

Quelques opérateurs sont moins classiques: l'affectation, les opérations d'incrémentation et les opérations sur les *bits*. L'affectation est un opérateur qui rend comme valeur la valeur affectée à la partie gauche. On peut donc écrire simplement

```
x = y = z = 1;
```

pour

```
x = 1; y = 1; z = 1;
```

Une expression qui contient une affectation modifie donc la valeur d'une variable pendant son évaluation. On dit alors que cette expression a un *effet de bord* . Les effets de bord sont à manipuler avec précautions, beaucoup des opérateurs n'ayant pas d'ordre d'évaluation bien défini, ainsi que l'ordre d'évaluation des arguments d'une fonction, (pour permettre l'optimisation de la compilation des expressions).

Expressions d'incrémentation

D'autres opérations dans les expressions peuvent changer la valeur des variables. Les opérations de pré-incrémentation, de post-incrémentation, de pré-décrémentation, de post-décrémentation permettent de donner la valeur d'une variable en l'incrémentant ou la décrémentant avant ou après de lui ajouter ou retrancher 1. Supposons que `n` vaille 5, alors le programme suivant

```
x = ++n;
y = n++;
z = --n;
t = n--;
```

fait passer `n` à 6, met 6 dans `x`, met 6 dans `y`, fait passer `n` à 7, puis retranche 1 à `n` pour lui donner la valeur 6 à nouveau, met cette valeur 6 dans `z` et dans `t`, et fait passer `n` à 5. Plus simplement, on peut écrire simplement

```
++i;
j++;
```

pour

```
i = i + 1;
j = j + 1;
```

De manière identique, on pourra écrire

```
if (c != ' ')
    s[i++] = c;
```

pour

```
if (c != ' ') {
    s[i] = c;
    ++i;
}
```

En règle générale, il ne faut pas abuser des opérations d'incrémentation. Si c'est une commodité d'écriture comme dans les deux cas précédents, il n'y a pas de problème. Si l'expression devient incompréhensible et peut avoir plusieurs résultats possibles selon un ordre d'évaluation dépendant de l'implémentation, alors il ne faut pas utiliser ces opérations et on doit casser l'expression en plusieurs morceaux pour séparer la partie effet de bord.

En C, on ne doit pas faire d'effets de bord dépendants de l'implémentation

Expressions sur les bits

Les opérations sur les *bits* peuvent se révéler, elles, très utiles. On peut faire `&` (*et* logique), `|` (*ou* logique), `^` (*ou* exclusif), `<<` (décalage vers la gauche), `>>` (décalage vers la droite), `~` (complément à un). Ainsi

```
x = x & 0xff;
y = y | 0x40;
```

mettent dans `x` les 8 derniers bits de `x` et positionne le $6^{\text{ème}}$ bit à partir de la droite dans `y`. Il faut bien distinguer les opérations logiques `&&` et `||` à résultat booléens 0 ou 1 des opérations `&` et `|` sur les *bits* qui donnent toute valeur entière. Par exemple, si `x` vaut 1 et `y` vaut 2, `x & y` vaut 0 et `x && y` vaut 1.

Les opérations `<<` et `>>` décalent leur opérande de gauche de la valeur indiquée par l'opérande de droite. Ainsi `3 << 2` vaut 12, et `7 >> 2` vaut 1. Les décalages à gauche introduisent toujours des zéros sur les bits de droite. Pour les bits de gauche dans le cas des décalages à droite, c'est dépendant de la machine; mais si l'expression décalée est `unsigned`, ce sont toujours des zéros.

Le complément à un est très utile dans les expressions sur les *bits*. Il permet d'écrire des expressions indépendantes de la machine. Par exemple

```
x = x & ~0x7f;
```

remet à zéro les 7 bits de gauche de `x`, indépendamment du nombre de bits pour représenter un entier. Une notation, supposant des entiers sur 32 bits et donc dépendante de la machine, serait

```
x = x & 0xffff8000;
```

Autres expressions d'affectation

A titre anecdotique, les opérateurs d'affectation peuvent être plus complexes que la simple affectation et permettent des abréviations parfois utiles. Ainsi, si *op* est un des opérateurs `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, ou `|`,

e_1 *op*`=` e_2

est un raccourci pour

e_1 `=` e_1 *op* e_2

Expressions conditionnelles

Parfois, on peut trouver un peu long d'écrire

```
if (a > b)
    z = a;
else
    z = b;
```

L'expression conditionnelle

e_1 `?` e_2 `:` e_3

évalue e_1 d'abord. Si non nul, le résultat est e_2, sinon e_3. Donc le maximum de `a` et `b` peut s'écrire

```
z = (a > b) ? a : b;
```

Les expressions conditionnelles sont des expressions comme les autres et vérifient les lois de conversion. Ainsi si e_2 est flottant et e_3 est entier, le résultat sera toujours flottant.

Précédence et ordre d'évaluation

Certains opérateurs ont des précédences évidentes, et limitent l'utilisation des parenthèses dans les expressions. D'autres sont moins clairs, particulièrement en C où leur nombre est plus grand qu'en Pascal. Voici la table donnant les précédences dans l'ordre décroissant et le parenthésage en cas d'égalité

Opérateurs	Associativité
`() [] -> .`	gauche à droite
`! ~ ++ -- + - = * & (`*type*`) sizeof`	droite à gauche
`*  /  %`	gauche à droite
`+  -`	gauche à droite
`<<  >>`	gauche à droite
`<  <=  >  >=`	gauche à droite
`==  !=`	gauche à droite
`&`	gauche à droite
`^`	gauche à droite
`|`	gauche à droite
`&&`	gauche à droite
`||`	gauche à droite
`?:`	droite à gauche
`= += -= /= %= &= ^= |= <<= >>=`	droite à gauche
`,`	gauche à droite

En règle générale, il est conseillé de mettre des parenthèses si les précédences ne sont pas claires. Par exemple

```
if ((x & MASK) == 0) ...
```

Remarquons encore que l'ordre d'évaluation des arguments n'est pas précisé pour la plupart des opérateurs. Les seuls opérateurs qui sont toujours évalués séquentiellement de la gauche vers la droite sont `&&`, `||`, `?:` et `,`. Il en va de même pour l'appel des fonctions. L'ordre d'évaluation des arguments n'est pas précisé, et donc on peut avoir des résultats inattendus si on fait des effets de bord dans les arguments. Par exemple

```
printf ("%d %d\n", ++n, 2 * n);
```

est du C mal écrit. Si `n` = 1, on peut obtenir "2 2" ou "2 4" selon l'ordre d'évaluation qui n'est pas précisé. Il faut donc écrire

```
++n;
printf ("%d %d\n", n, 2 * n);
```

pour obtenir un résultat indépendant de l'ordre.

B.2.5 Instructions

Toute expression suivie d'un point-virgule devient une instruction. Ainsi

```
x = 3;
++i;
printf(...);
```

sont des instructions (une expression d'affectation, d'incrémentation, un appel de fonction suivi de point-virgule). Donc point-virgule fait partie de l'instruction, et n'est pas un séparateur comme en Pascal. De même, les accolades `{ }` permettent de regrouper des instructions en séquence. Ce qui permet de mettre plusieurs instructions dans les alternatives d'un `if` par exemple.

Les instructions de contrôle sont à peu près les mêmes qu'en Pascal. D'abord les instructions conditionnelles `if` sont de la forme

```
if (E)
    S1
```

ou

```
if (E)
    S1
else
    S2
```

Remarquons bien qu'une instruction peut être une expression suivie d'un point-virgule (contrairement à Pascal). Donc l'instruction suivante est complètement licite

```
if (x < 10)
    c = '0' + x;
else
    c = 'a' + x - 10;
```

Il y a la même convention qu'en Pascal pour les `if` emboîtés. Le `else` se rapportant toujours au `if` le plus proche. Une série de `if` peut être remplacée par une instruction par cas. En C, elle se nomme instruction `switch`. Elle a la syntaxe suivante

```
switch (E) {
    case c1: instructions
    case c2: instructions
    ...
    case cn: instructions
    default: instructions
}
```

Cette instruction a une idiosyncrasie bien particulière. Pour sortir de l'instruction, il faut exécuter une instruction `break`. Sinon, le reste de l'instruction est fait en séquence. Cela permet de regrouper plusieurs alternatives, mais peut être particulièrement dangereux. Par exemple, le programme suivant

```
switch (c) {
    case '\t':
    case ' ':
        ++ nEspaces;
        break;
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        ++ nChiffres;
        break;
    default:
        ++ nAutres;
        break;
}
```

permet de factoriser le traitement de quelques cas. On verra que l'instruction `break` permet aussi de sortir des boucles. Il faudra donc bien faire attention à ne pas oublier le `break` à la fin de chaque cas, et à ce que `break` ne soit pas intercepté par une autre instruction.

Les itérations sont réalisées par les instructions `for`, `while`, et `do...while`. L'instruction `while` a le même sens qu'en Pascal. On itère l'instruction S tant que la condition E est vraie (non nulle) par

```
while (E)
     S
```

et on fait de même en effectuant au moins une fois la boucle par

```
do
     S
while (E);
```

Remarque: cette dernière instruction s'arrête quand la condition n'est plus vraie, contrairement au `repeat` de Pascal. L'instruction d'itération la plus puissante est l'instruction `for`. Sa syntaxe est

```
for (E1; E2; E3)
    S
```

qui est équivalente à

```
E1;
while (E2) {
     S;
```

E_3 ;
}

Elle est donc beaucoup plus complexe qu'en Pascal et peut donc ne pas terminer, puisque les expressions E_2 et E_3 sont quelconques. L'exemple suivant ressemble à Pascal

```
for (i = 0; i < 100; ++i)
     a[i] = 0;
```

mais l'itération suivante est plus complexe (voir page 41)

```
for (i = h(x, l); i != -1; i = col[i])
    if (strcmp (x, nom[i]) == 0)
        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. Finalement, l'instruction maudite `goto` permet de rejoindre toute étiquette désignée par un identificateur. Attention, cette étiquette ne peut être que dans la procédure courante (sinon il faut utiliser les horribles fonctions `setjmp` et `longjmp` dont l'usage est très limité).

B.2.6 Procédures, fonctions, structure d'un programme

La syntaxe des fonctions et procédures a déjà été vue dans l'exemple du carré magique. Un programme est une suite linéaire de fonctions ou procédures, non emboîtées. Par convention, le début de l'exécution est donné à la procédure `main`. Nous adoptons la convention ANSI pour les déclarations des paramètres. Ainsi

```
int strlen (char s[])
{
...
}
```

est la fonction qui retourne la longueur de la chaîne de caractères `s`. La méthode non ANSI, que l'on retrouve encore dans les vieux sources de fonctions de bibliothèque C écrivait le programme précédent sous la forme moins naturelle suivante

```
int strlen (s)
char s[];
{
...
}
```

L'instruction

`return` e;

sort de la fonction en donnant le résultat e En C ANSI, le type spécial `void` indique qu'une procédure n'a pas de résultat ou qu'une fonction n'a pas d'argument. Ainsi

```
void Imprimer (int a[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf ("%d ", a[i]);
    printf ("\n");
}
```

ou

```
int rand(void)
{
    cf. page 240
}
```

L'utilisation de `rand` se fait simplement en appelant la fonction `rand()`. Il faut bien noter que contrairement à Pascal on met des parenthèses à l'appel d'une fontion ou procédure dans tous les cas (même s'il n'y a pas d'arguments).[1]

Il peut y avoir des variables locales dans une procédure, plus exactement dans toute instruction composée entourée par des accolades. Les variables globales sont 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. Il y a toutefois une exception pour les variables `static` qui ont une valeur rémanente, et qui reprennent leur dernière valeur lorsqu'on revient dans la fonction. Les variables locales normales (dont la valeur est fugitive) sont dites variables automatiques.

Variables locales statiques et variables globales sont initialisées au début de l'exécution du programme, souvent par l'éditeur de liens (*linker* ou *loader* en Unix), et ne peuvent être que des expressions simples à calculer (grosso modo des constantes). Voici quelques exemples de telles initialisations.

```
int Registre (int n)
{
     static int r = 0;
     int oldr;

     oldr = r;
     r = n;
     return (oldr);
}
```

qui peut se combiner en

```
int Registre (int n)
{
     static int r = 0;
     int oldr = r;
     r = n;
     return (oldr);
}
```

[1] Dans le cours, nous avons dévié de la convention ANSI pure et dure en autorisant les déclarations de fonctions de zéro arguments sans mettre `void` dans l'argument. Beaucoup de compilateurs C acceptent une telle écriture. Toutefois, nous exigerons de mettre `void` dans les déclarations de prototype.

Les variables globales s'initialisent comme suit

```
int nMois = 12;
int nJours[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Dans le cas des tableaux, l'initialisation peut définir la taille du tableau.

Les fonctions doivent toujours être déclarées avant leur utilisation. Il s'agit de ne donner que la signature de la fonction pour permettre de vérifier les types. Le nom des paramètres est donc superflu. Les déclarations de *prototypes* de fonctions suivantes, que l'on retrouve souvent dans les fichiers *include*, sont par exemple dans `<math.h>`

```
double acos(double);
double asin(double);
double atan2(double, double);
double ceil(double);
double cosh(double);
double floor(double);
double fmod(double, double);
double frexp(double, int *);
double ldexp(double, int);
double log10(double);
double modf(double, double *);
double pow(double, double);
double sinh(double);
double tanh(double);
```

Il n'y a que l'appel par valeur pour évaluer les arguments des fonctions. Il n'y a pas d'appel par référence. Pour simuler l'appel par référence, on doit passer explicitement la référence à un objet, et utiliser cet objet par la suite comme un pointeur(!) dont nous verrons la syntaxe dans la section suivante. En attendant, une manière simple de simuler l'appel par référence est de faire comme suit

```
void ProcedureAvecAppelParReference (int *ap)
{
    int   a = *ap;

    On utilise partout a comme en Pascal
    *ap = a;
}
```

B.2.7 Pointeurs et tableaux

C'est un des points les plus troublants en C pour le débutant. Les pointeurs ressemblent beaucoup à Pascal, seule la syntaxe diffère. Les opérateurs de base sont `&` pour référencer un objet et `*` pour déréférencer, écrits en préfixe. Ainsi

```
int   x = 1, y = 2, z[10];
int   *ip;        /* ip est un pointeur vers un entier */

ip = &x;          /* ip pointe vers x */
y = *ip;          /* y vaut maintenant 1 */
*ip = 0;          /* x vaut maintenant 0 */
ip = &z[0];       /* ip pointe maintenant vers z[0] */
```

De même, la fonction d'échange s'écrira

```
int   x, y;

void Echange (int *xp, int *yp)
{
    int z = *xp;
    *xp = *yp;
    *yp = z;
}
...
    Echange (&x, &y);
```

Nous verrons plus tard (cf. section B.2.8) les fonctions `malloc` et `free` de la librairie C qui permettent à un pointeur de manipuler des objets du tas (comme `new` et `dispose` en Pascal). La constante `NULL`, déinie dans le fichier `<stdio.h>` est la valeur standard pour un pointeur vide (le `nil` de Pascal). Mais, en plus des opérations usuelles d'égalité sur les pointeurs, on peut faire des opérations arithmétiques (additions et soustractions). Nous éviterons d'en faire trop dans ce cours. En général, il s'agit de pointer sur l'élément d'un tableau et de passer sur ses éléments voisins. Tout se dérive de l'égalité suivante, que l'on peut considérer comme l'équation de C (pour le meilleur et pour le pire!)

Equation des pointeurs: $\quad$ `&a[i]` $=$ `a + i`

Les corollaires de cette équation sont nombreux. D'abord si `i` $= 0$, cela veut dire qu'une expression contenant le nom d'un tableau `a` est un raccourci pour signifier la valeur d'un pointeur pointant sur son premier élément (toujours à l'indice 0). Ainsi on comprend mieux les signatures des fonctions `Erreur` de l'exemple du carré magique (page 233), de `atoi` (page 238), de `strlen` (page 245), `Imprimer` (page 245). Ensuite, comme $*$ `&a[i]` $\equiv$ `a[i]`, on a $*$ `(a + i)` $\equiv$ `a[i]`. Nous utiliserons peu cette écriture inélégante, mais le type de ces deux notations permet d'écrire autrement la signature des fonctions précédentes

```
void Erreur (char *s)   {...}
```

Ecrire sous une forme ou l'autre est une affaire de goût, mais dans le cas présent la première est plus naturelle, puisque c'est vraiment une chaîne de caractères que l'on veut donner comme argument. Enfin, si `p` $=$ `&a[i]` $=$ `a + i` , l'instruction

```
p = p + j;
```

implique que `p` $=$ `a + i + j` $=$ `&a[i + j]`. Donc, si on incrémente un pointeur, on passe à l'élément suivant du tableau, quelque soit le type de cet élément.

Il y a un type de tableau qui est très utilisé: les chaînes de caractères. Une chaîne de caractères est un simple tableau de caractères. La fin d'une chaîne de caractères est le caractère `'\0'`. Ainsi

```
char   ch[256];

    ch[0] = 'P'; ch[1] = 'a'; ch[2] = 'u'; ch[3] = 'l'; ch[4] = '\0';
```

décrit un tableau `ch` contenant la chaîne de caractères `"Paul"` de longueur 4. On aurait pu obtenir le même résultat avec la fonction `strcpy` de la bibliothèque C, dont le prototype est dans `<string.h>`, en faisant

```
strcpy (ch, "Paul");
```

Le fichier *include* `<string.h>` contient des opérations de comparaisons, de recherche de sous-chaîne.

Enfin, il faut bien comprendre qu'il existe des différences entre tableaux et pointeurs. Au point de vue mémoire, un pointeur est un simple "mot-mémoire" contenant une référence vers un élément du langage. Un tableau est un ensemble de cases mémoire pour chacun de ses éléments. Ainsi un tableau `a` de pointeurs vers des caractères est différent, d'un tableau `b` de caractères à deux dimensions

```
char    *a[10];
char     b[10][256];
```

Le premier occupe 10 cases mémoires, le deuxième 2560 octets, même si `a` et `b` peuvent être passés comme argument d'une même procédure

```
void F(char **x) {...}
```

En Unix, la fonction `main` a comme argument le nombre d'argument `argc` de la ligne de commande appelant le programme, et un tableau de pointeurs vers les chaînes arguments `argv`. (`argv[0]` est le nom de la commande elle-même). Ainsi la commande `echo` du système Unix s'écrit

```
int main (int argc, char *argv[])
{
    int  i;

    while (--argc > 0)
        printf ("%s%s", argv[i], (i < argc - 1) ? " " : "");
    printf ("\n");
    return 0;
}
```

Il n'y a pas de variables fonctionnelles (comme en Scheme ou ML). Mais, il existe des pointeurs sur des fonctions. Ainsi on peut faire dépendre une fonction d'une autre fonction, en lui passant en argument un pointeur vers une fonction. On peut donc reprendre le programme de recherche de zéro d'une fonction quelconque (cf. page 209).

```
#include  <math.h>
#define Pi         3.14
#define Epsilon    1.0e-7
#define Nmax       100

double Zero (double (*f)(double), double a,b)
{
    int     n;
    double   m;

     n = 1;
     while (fabs (b - a) < Epsilon && n < Nmax) {
         m = (a + b) / 2;
         if ((*f) (m) > 0 == (*f) (a) > 0)
             a = m;
         else
             b = m;
         ++n;
     }
     return a;
}
...
```

```
    Zero (sin, Pi/2, 3*Pi/2);
    Zero (cos, 0, Pi);
```

Par convention, une fonction représente un pointeur sur elle-même, un peu comme pour les tableaux, et le déréférencement implicite se fera (à un niveau). On peut donc simplement écrire `f(m)` pour `(*f)(m)`. La signature de `Zero` peut aussi être simplifiée. L'écriture ressemble alors à celle de Pascal.

B.2.8 Structures

Ce sont le pendant des enregistrements de Pascal.

```
enum  Mois {Jan, Fev, Mar, Avr, Mai, Juin, Juil,
                      Aou, Sep, Oct, Nov, Dec);

struct Date {
        int         j;   /* Jour */
        enum Mois   m;   /* Mois */
        int         a;   /* Année */
};

struct Date    berlin, bastille;
...
    berlin.j = 10; berlin.m = Nov; berlin.a = 1989;
    bastille.j = 14; bastille.m = Juil; bastille.a = 1789;
```

On peut déclarer les types en utilisant le mot-clé `typedef`. Le type défini se met au même endroit où on déclarerait une variable de ce type. L'exemple précédent se réécrit, sous forme plus pascalienne,

```
typedef int   Jour;
typedef enum  {Jan, Fev, Mar, Avr, Mai, Juin, Juil,
               Aou, Sep, Oct, Nov, Dec) Mois;
typedef int   Annee;
typedef struct {
        Jour   j;
        Mois   m;
        Anne   a;
} Date;

Date    berlin, bastille;
...
    berlin.j = 10; berlin.m = Nov; berlin.a = 1989;
    bastille.j = 14; bastille.m = Juil; bastille.a = 1789;
```

L'égalité des types en C est plus structurelle qu'en Pascal. On redescend à un type de base `int`, `char`, ou à un type `struct` et on compare les expressions de types. Donc si `berlin` est défini sous la deuxième forme, et `bastille` sous la première, on peut toujours affecter une variable à l'autre. Techniquement, les structures sont les seules constructions génératives de type (cf. page 205).

Il existe des variantes possibles dans les structures, les *unions*. Il n'y a pas de notion d'indicateur comme en Pascal. Tout est laissé à la responsabilité du programmeur. Ainsi

```
union Complexe {
    struct {
        float re;
```

```
        float im;
    } cartesiennes;
    struct {
        float rho
        float theta
    } polaires;
};

union Complexe x;

...
    x.cartesiennes.re = 0; x.cartesiennes.im = 1;
    x.polaires.rho = 1; x.polaires.theta = PI/2;
```

Bien sûr, un champ union peut se retrouver dans le champ d'une autre structure, comme une structure peut être une sous-structure d'une autre. Il faudra cependant bien faire attention à l'égalité de la taille mémoire des différentes possibilités, si on ne veut pas de problème.

Plusieurs opérations sont possibles sur les structures ou unions. Nous avons vu implicitement le point postfixe pour accéder aux champs comme en Pascal. Comme les opérateurs postfixes ont précédence sur les préfixes, il existe une notation spéciale pour les pointeurs sur les structures qui facilite l'écriture

```
p -> x
```

pour

```
(*p).x
```

Enfin, la taille d'une structure s'obtient par `sizeof`. Ainsi `sizeof(Date)` donnera la taille en octets de la structure `Date`. C'est particulièrement important pour la fonction `malloc` d'allocation mémoire qui permet de donner à un pointeur la valeur d'une référence vers une nouvelle structure (ou quelconque objet) du tas. Une instruction très fréquente est

```
p = (type *) malloc (sizeof (type));
```

qui est l'équivalent du `new(p)` de Pascal. En C, on ne dispose que de la fonction `malloc` qui prend un nombre d'octets en argument et retourne un pointeur vers `void`. (Les pointeurs vers `void` peuvent être convertis à tout autre expression sans perdre d'information). L'expression `free(p)` libère un espace alloué précédemment. Enfin, pour définir des structures de données récursives, on est ennuyé comme en Pascal, puisque les objets doivent être définis avant leur utilisation. On utilise le fait que les pointeurs sont tous de taille fixe, et on définira ainsi une structure arborescente

```
struct Noeud {
    int             contenu;
    struct Noeud    *filsG;
    struct Noeud    *filsD;
} *a, *b;
```

B.2.9 Entrées-Sorties

Les entrées sorties sont à l'extérieur du langage, et s'inspirent fortement de celles du système Unix. Les prototypes des fonctions sont dans le fichier *include* `<stdio.h>`. Les entrées sorties formattées ont déjà été vues. Elles s'obtiennent par les deux fonctions

```
int printf( char *format, arg1, arg2, ...)
int scanf( char *format, arg1, arg2, ...)
```

On imprime le format premier argument où chaque trou désigné par un % suivi d'un attribut est remplacé par les arguments arg_i dans l'ordre. Il doit donc y avoir autant de tels arguments que de % dans le format. Les attributs principaux sont `d` pour décimal, `f` pour `double`, `s` pour les chaînes de caractères, `c` pour un caractère. Ainsi

```
printf ("x = %d\n", 100)      donne     x = 100 newline
printf ("x = %5d:", 100)                x =   100:
printf ("x = %-5d:", 100)               x = 100  :
printf ("y = %6.2f:", 3.1415)           y =   3.14:
printf ("y = %-6.2f:", 3.1415)          y = 3.14  :
printf ("c = '%c'", 'A')                c = 'A'
printf ("Enfin, %s:, "la fin")          Enfin, la fin
```

Le résultat de `printf` est le nombre de caractères imprimés. L'opération duale est `scanf` sauf que les blancs et tabulations du format sont ignorés. Les fonctions `sprintf` et `sscanf` sont indentiques, mais lisent ou écrivent dans la chaîne de caractère donnée en premier argument avant le format.

On peut ne pas utiliser les impressions formatées. Ainsi, `putchar` et `getchar` permettent d'écrire ou de lire un caractère. L'entier EOF de défini dans `<stdio.h>` dit si le caractère lu est la fin de l'entrée. On peut donc écrire tout ce que l'on lit dans la fenêtre de texte par le programme suivant

```
#include <stdio.h>

int main()
{
    int   c;

    while ((c = getchar()) != EOF)
        putchar (c);
}
```

On peut aussi manipuler des fichiers, grâce aux pointeurs vers les structures fichiers, définies dans `<stdio.h>`. Ainsi le programme précédent se réécrit pour copier les fichiers de nom `MonFichier` dans celui dénommé `MaCopie`.

```
#include <stdio.h>

int main()
{
    FILE   *ifp, *ofp;
    void   FileCopy (FILE *, FILE *);

    if ((ifp = fopen ("MonFichier", "r")) == NULL) {
        printf ("On ne peut ouvrir %s.\n", "MonFichier");
        return 1;
    }
    if ((ofp = fopen ("MaCopie", "w")) == NULL) {
        printf ("On ne peut ouvrir %s.\n", "MaCopie");
        return 1;
    }
    FileCopy (ifp, ofp);
    fclose (ifp);
```

```
    fclose (ofp);
    return 0;
}

void FileCopy (FILE *ifp, FILE *ofp)
{
    int   c;

    while ((c = getc(ifp)) != EOF)
        putc (c, ofp);
}
```

La procédure de recopie ressemble au programme précédent. Il existe 3 pointeurs spéciaux vers des fichiers, prédéfinis dans `<stdio.h>`: `stdin` l'entrée standard (de la fenêtre de texte), `stdout` la sortie standard (dans la fenêtre de texte), `stderr` la sortie standard des erreurs (qui n'a vraiment de sens que dans le système Unix). On voit donc que `getchar()` et `putchar(c)` sont des raccourcis pour `getc(stdin)` et `putc (c, stdout)`.

Les entrées sorties formatées peuvent se faire dans des fichiers, dont on met le pointeur vers la structure `FILE` associée en premier argument avant le format. Ces fonctions s'appellent `fprintf` et `fscanf`. L'association entre les fichiers et les noms de fichiers se fait par les fonctions de la librairie standard `fopen` et `fclose`. On peut remarquer le deuxième argument qui dit si on veut lire `"r"`, écrire `"w"`, ou faire les deux `"rw"`. On peut se positionner à un endroit quelconque dans un fichier avec `fseek`

```
int fseek(FILE *fp, long offset, int origin);
```

qui se place dans le fichier `fp` en `offset` avec le mode défini par `origin` (0 en absolu depuis le début du fichier, 1 en relatif par rapport à la position courante, 2 en absolu à partir de la fin).

B.2.10 Fonctions graphiques

Les fonctions sont les mêmes qu'en Pascal. Il existe toutefois une différence importante: l'environnement graphique n'est pas défini par défaut dans l'environnement *ThinkC*. Il faut donc l'initialiser par la procédure `InitQuickDraw()` [2] et inclure le fichier *include* `<MacLib.proto.h>` de Philippe Chassignet. Le plus simple est de se fournir le *projet VoidC* qui contient ce prélude et la bibliothèque d'interface graphique.

```
typedef int  ShortInt;

typedef struct {
    ShortInt  v, h;
} Point;

typedef struct {
    ShortInt  top, left, bottom, right;
} Rect;
```

Et les fonctions correspondantes (voir page 217)

```
void SetRect (Rect *, ShortInt, ShortInt, ShortInt, ShortInt);
void UnionRect (Rect *, Rect *, Rect *);
```

[2] Attention, il ne faut pas oublier les parenthèses dans l'appel de la procédure sans argument `InitQuickDraw()`. C'est une erreur classique pour les débutants en C, et, après une légère réflexion, on se rend compte que la syntaxe autorise l'absence de parenthèse et qu'on se contente alors de retourner un pointeur vers la procédure sans l'appeler!!!.

```
void FrameRect (Rect *);
void PaintRect (Rect *);
void EraseRect (Rect *);
void InvertRect (Rect *);
void FrameOval (Rect *);
void PaintOval (Rect *);
void EraseOval (Rect *);
void InvertOval (Rect *);
typedef int Byte;;
typedef Byte Pattern [8];;
void FillRect (Rect *, Pattern);
void FillOval (Rect *, Pattern);
void FrameArc (Rect *, ShortInt, ShortInt);
void PaintArc (Rect *, ShortInt, ShortInt);
void EraseArc (Rect *, ShortInt, ShortInt);
void InvertArc (Rect *, ShortInt, ShortInt);
int Button (void);
void GetMouse (Point *);
```

Toutes ces définitions sont aussi sur Vax dans les fichiers *include* `<MacLib.h>` sur Mac et dans `<MacLib.h>` et `<MacLib.proto.h>` dans le *directory* `/usr/local/tgix/c`. Le programme de *pong*, écrit en Pascal dans le chapître précédent, devient instantanément le programme suivant.

```
#include "MacLib.proto.h"

#define  C    5              /* Le rayon de la balle */
#define  X0   5
#define  X1   250
#define  Y0   5
#define  Y1   180

void GetXY (int *xp, int *yp)
{
#define   N   2
    Rect    r;
    Point   p;
    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 */
    *xp = x; *yp = y;
}

int main()
{
    int   x, y, dx, dy;
```

```
    Rect  r, s;
    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(&x, &y);              /* On note les coordonnées du pointeur */
    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 = 1; i <= 2500; ++i)
            ;                    /* On temporise */
        InvertOval(&r);          /* On efface la balle */
    }
}
```

B.3 Syntaxe BNF de C

translation-unit:
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

function-definition:
declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

declaration:
declaration-specifiers init-declarator-list$_{opt}$;

declaration-list:
declaration
declaration-list declaration

declaration-specifiers:
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

storage-class-specifier: one of
`auto register static extern typedef`

type-specifier: one of
`void char short int long float double signed`
`unsigned` *struct-or-union-specifier enum-specifier typedef-name*

type-qualifier: one of
`const volatile`

struct-or-union-specifier:
struct-or-union identifier$_{opt}$ `{` *struct-declaration-list* `}`
struct-or-union identifier

struct-or-union: one of
`struct union`

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

init-declarator-list:
init-declarator
init-declarator-list `,` *init-declarator*

init-declarator:
declarator
declarator `=` *initializer*

struct-declaration:
specifier-qualifier-list struct-declarator-list `;`

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
struct-declarator
struct-declarator-list `,` *struct-declarator*

struct-declarator:
declarator
declarator$_{opt}$ `:`*constant-expression*

enum-specifier:
`enum` *identifier*$_{opt}$ `{` *enumerator-list* `}`
`enum` *identifier*

enumerator-list:
enumerator
enumerator-list `,` *enumerator*

enumerator:
identifier
identifier `=` *constant-expression*

declarator:
pointer$_{opt}$ *direct-declarator*

direct-declarator:
identifier
`(` *declarator* `)`
direct-declarator `[` *constant-expression*$_{opt}$ `]`
direct-declarator `(` *parameter-type-list* `)`

direct-declarator (*identifier-list*$_{opt}$)

pointer:
* *type-qualifier-list*$_{opt}$
* *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier

parameter-type-list:
parameter-list
parameter-list , ...

parameter-list:
parameter-declaration
parameter-list ,*parameter-declaration*

parameter-declaration:
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
identifier
identifier-list ,*identifier*

initializer:
assignment-expression
{ *initializer-list* }
{ *initializer-list* ,}

initializer-list:
initializer
initializer-list ,*initializer*

type-name:
specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
pointer
pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
(*abstract-declarator*)
direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
direct-abstract-declarator$_{opt}$ (*parameter-type-list*$_{opt}$)

typedef-name:
identifier

statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement

labeled-statement:
identifier : *statement*

`case` *constant-expression* `:` *statement*
`default :` *statement*

expression-statement:
expression$_{opt}$ `;`

compound-statement:
`{` *declaration-list*$_{opt}$ *statement-list*$_{opt}$ `}`

statement-list:
statement
statement-list statement

selection-statement:
`if (` *expression* `)` *statement*
`if (` *expression* `)` *statement* `else` *statement*
`switch (` *expression* `)` *statement*

iteration-statement:
`while (` *expression* `)` *statement*
`do` *statement* `while (` *expression* `) ;`
`for (` *expression*$_{opt}$ `;` *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

jump-statement:
`goto` *identifier* `;`
`continue ;`
`break ;`
`return` *expression*$_{opt}$ `;`

expression:
assignment-expression
expression `,` *assignment-expression*

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of
`= *= /= %= += -= <<= >>= &= ^= |=`

conditional-expression:
logical-OR-expression
logical-OR-expression `?` *expression* `:` *conditional-expression*

constant-expression:
conditional-expression

logical-OR-expression:
logical-AND-expression
logical-OR-expression `||` *logical-AND-expression*

logical-AND-expression:
inclusive-OR-expression
logical-AND-expression `&&` *inclusive-OR-expression*

inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression `|` *exclusive-OR-expression*

exclusive-OR-expression:
AND-expression

exclusive-OR-expression `^` *AND-expression*

AND-expression:
equality-expression
AND-expression `&` *equality-expression*

equality-expression:
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

relational-expression:
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression `>>` *additive-expression*

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

multiplicative-expression:
cast-expression
multiplicative-expression `*` *cast-expression*
multiplicative-expression `/` *cast-expression*
multiplicative-expression `%` *cast-expression*

cast-expression:
unary-expression
`(` *type-name* `)` *cast-expression*

unary-expression:
postfix-expression
`++` *unary-expression*
`--` *unary-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof (` *type-name* `)`

unary-operator: one of
`&` `*` `+` `-` `~` `!`

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `(` *argument-expression-list*$_{opt}$ `)`
postfix-expression `.` *identifier*
postfix-expression `->` *identifier*
postfix-expression `++`
postfix-expression `--`

primary-expression:

identifier
constant
string
(*expression*)

argument-expression-list:
assignment-expression
argument-expression-list , *assignment-expression*

constant:
integer-constant
character-constant
floating-constant
enumeration-constant

Et voici la syntaxe pour le préprocesseur:

control-line:
`#` `define` *identifier token-sequence*
`#` `define` *identifier*`(`*identifier* , ... ,*identifer*`)` *token-sequence*
`#` `undef` *identifier*
`#` `include` <*filename*>
`#` `include` "*filename*"
`#` `include` *token-sequence*
`#` `line` *constant* "*filename*"
`#` `line` *constant*
`#` `error` $token\text{-}sequence_{opt}$
`#` `pragma` $token\text{-}sequence_{opt}$
`#`
preprocessor-conditional

preprocessor-conditional:
if-line text elif-parts $else\text{-}part_{opt}$ `#` `endif`

if-line:
`#` `if` *constant-expression*
`#` `ifdef` *identifier*
`#` `ifndef` *identifier*

elif-parts:
elif-line text
$elif\text{-}parts_{opt}$

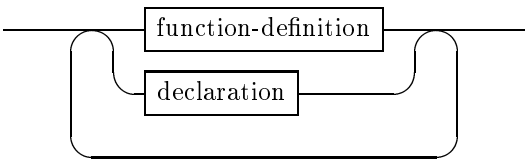
elif-line:
`#` `elif` *constant-expression*

elif-part:
else-line text

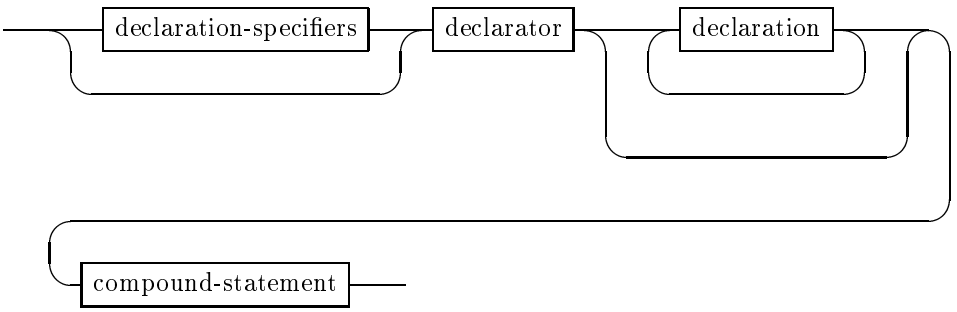
else-line:
`#` `else`

B.4 Diagrammes de la syntaxe de C

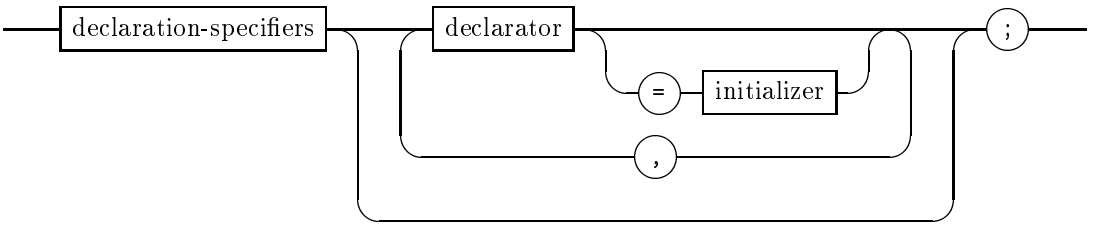
translation-unit



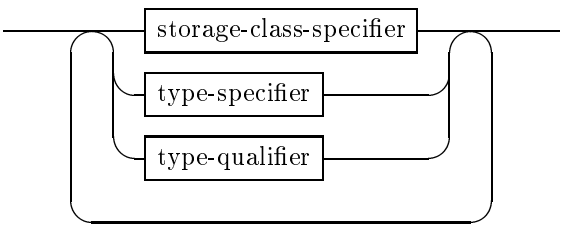
function-definition



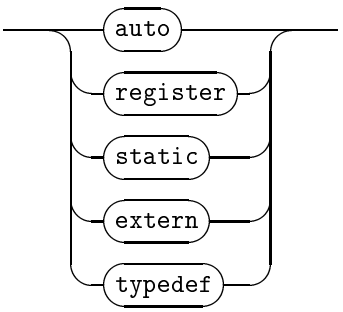
declaration



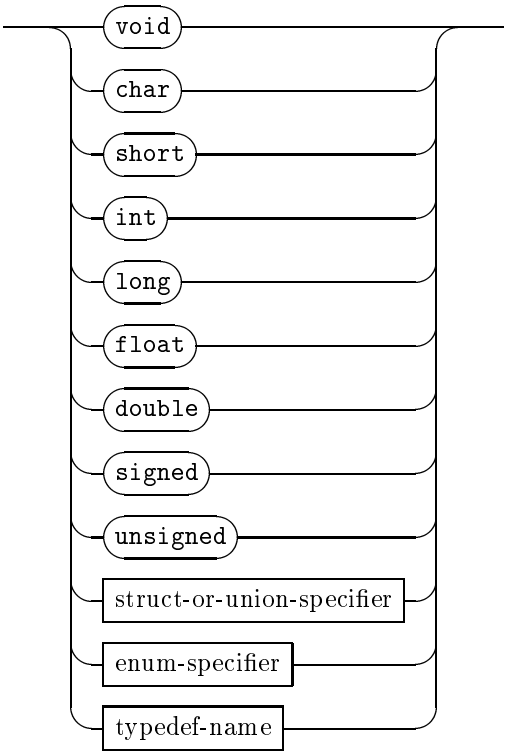
declaration-specifiers



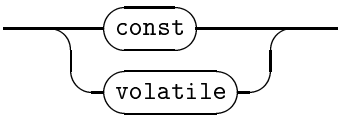
storage-class-specifier



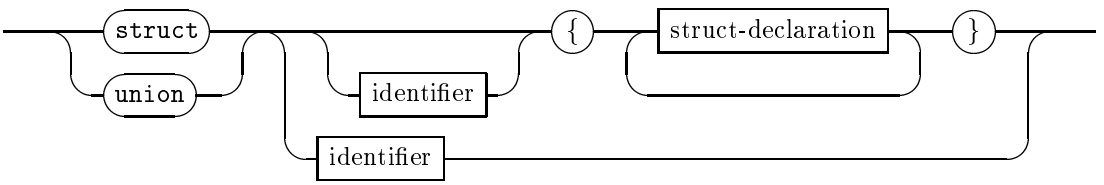
type-specifier



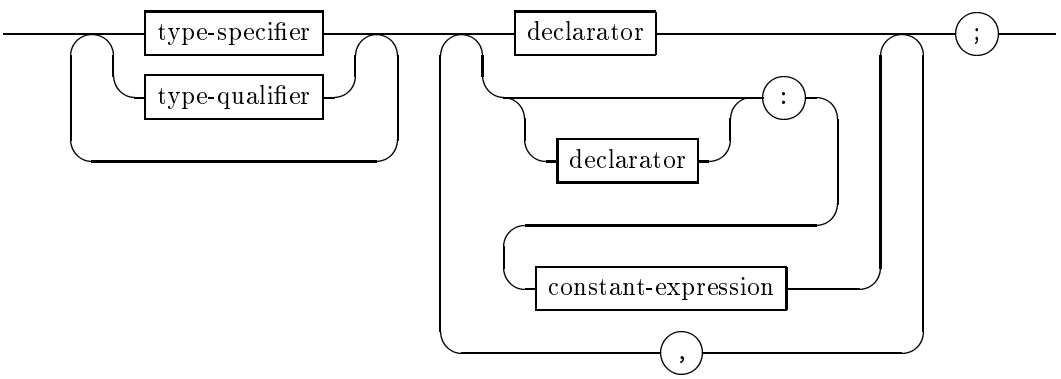
type-qualifier



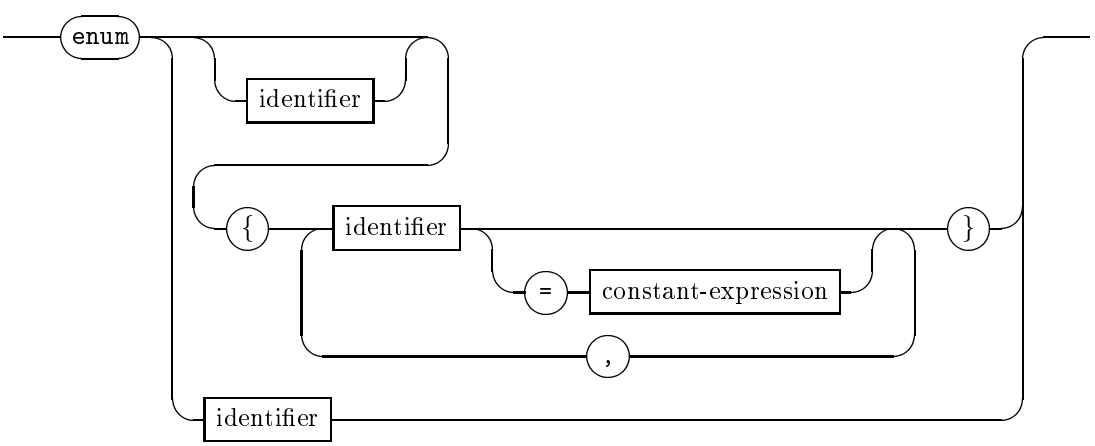
struct-or-union-specifier



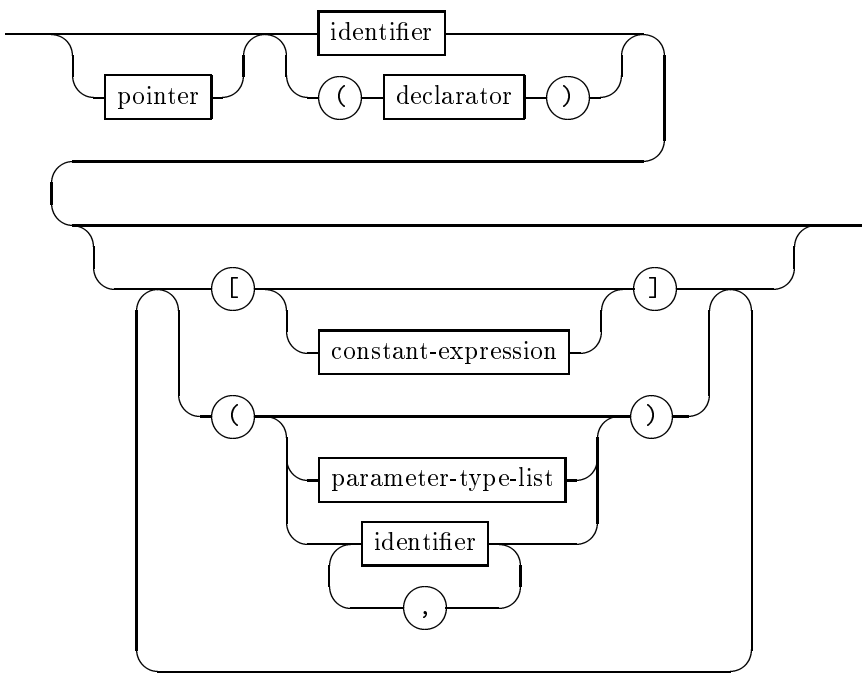
struct-declaration



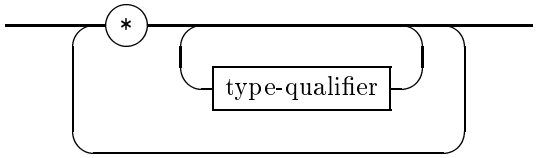
enum-specifier



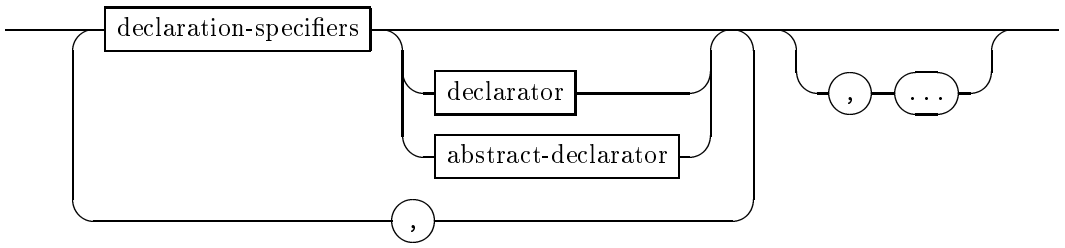
declarator



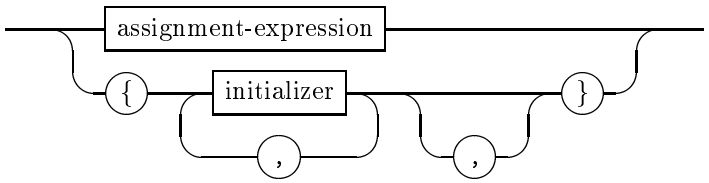
pointer



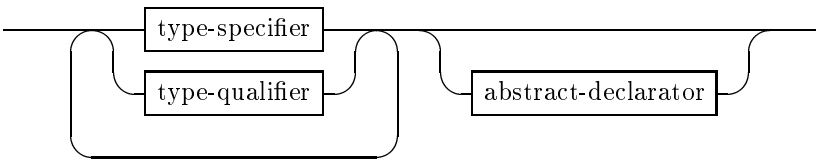
parameter-type-list



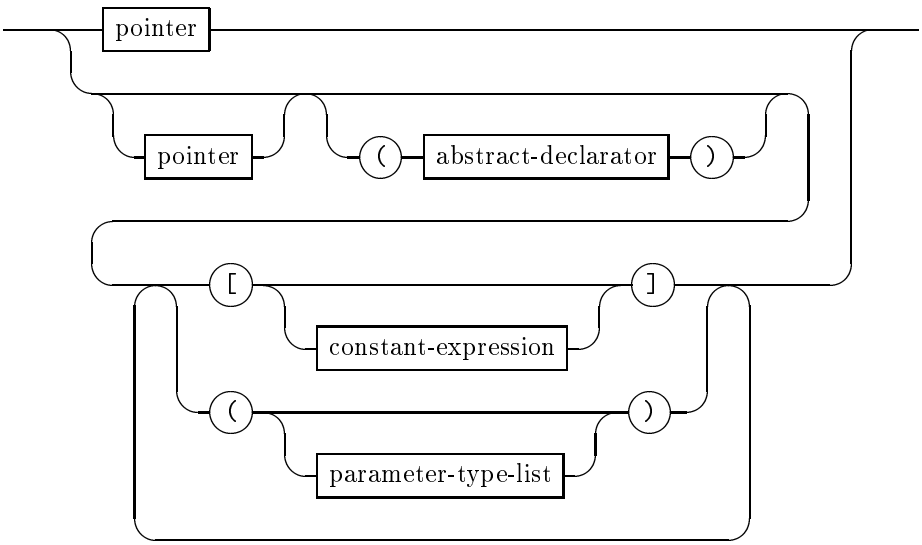
initializer



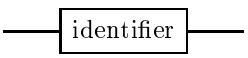
type-name



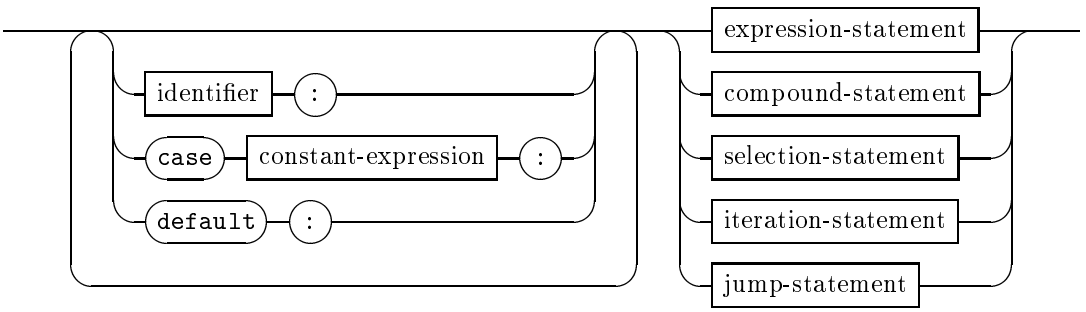
abstract-declarator



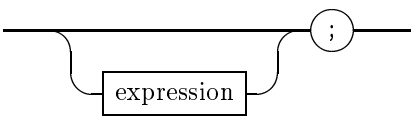
typedef-name



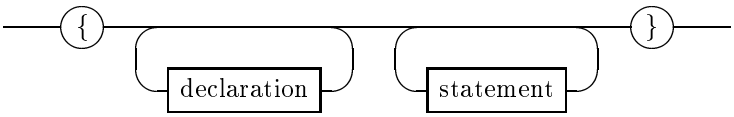
statement



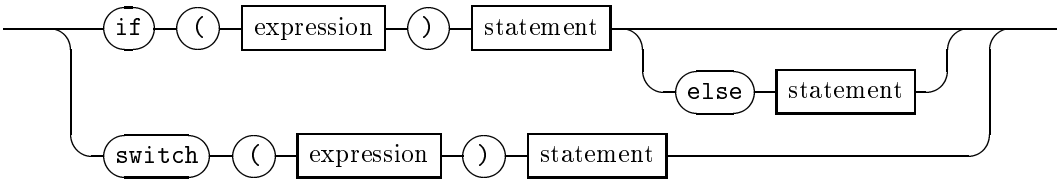
expression-statement



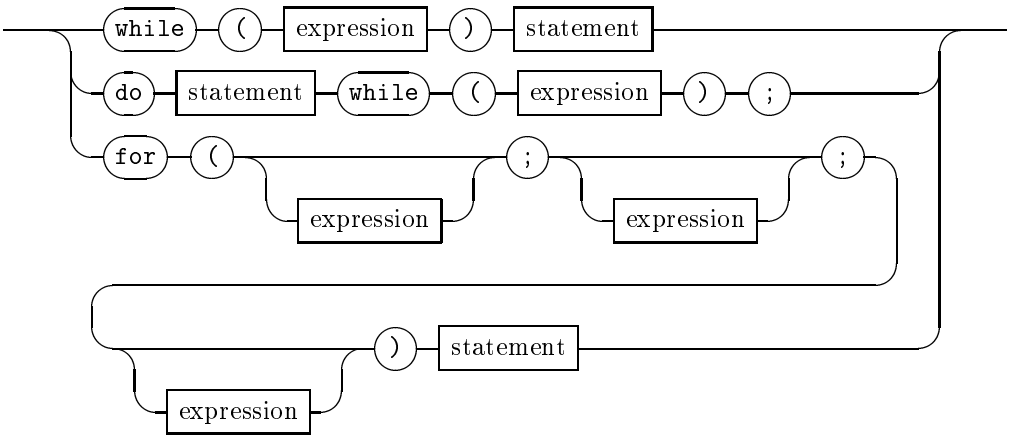
compound-statement



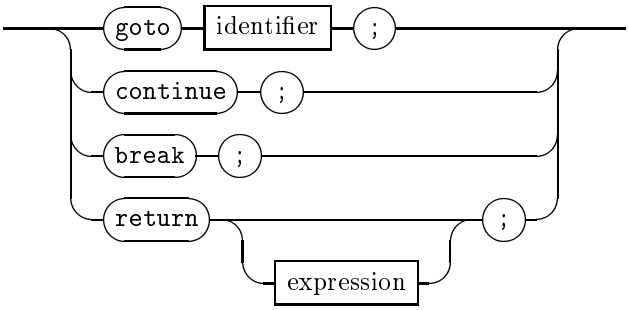
selection-statement



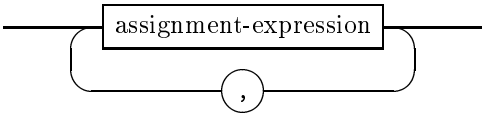
iteration-statement



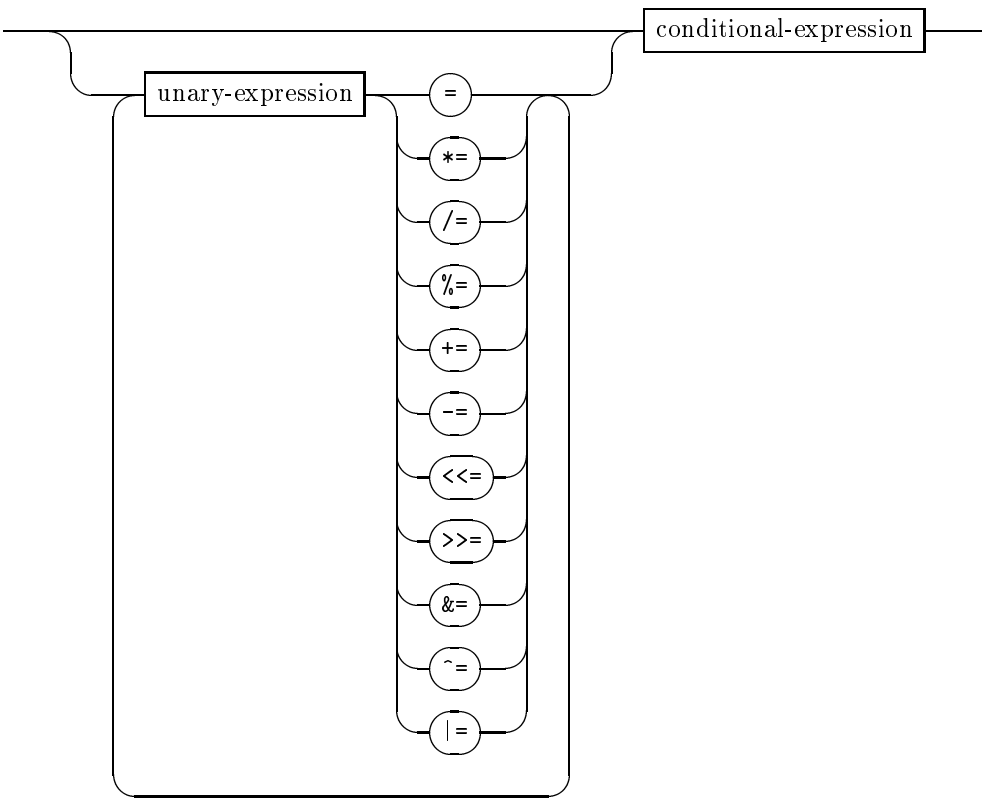
jump-statement



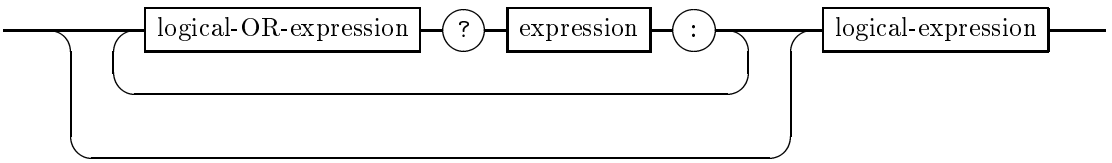
expression



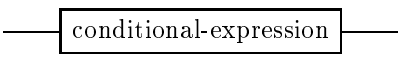
assignment-expression



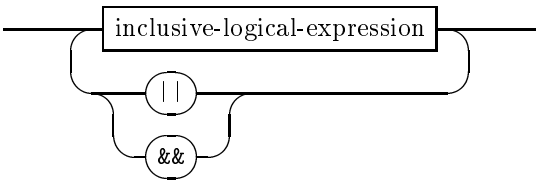
conditional-expression



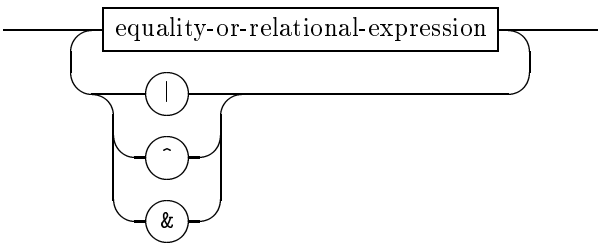
constant-expression



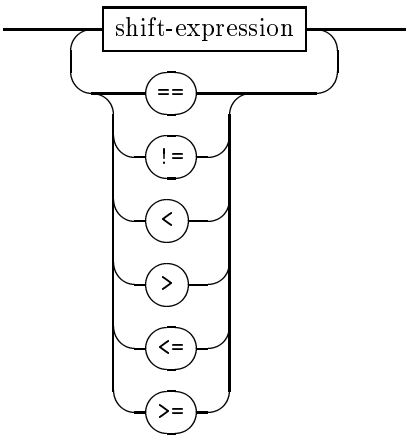
logical-expression



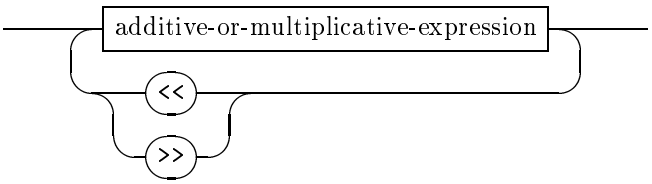
inclusive-logical-expression



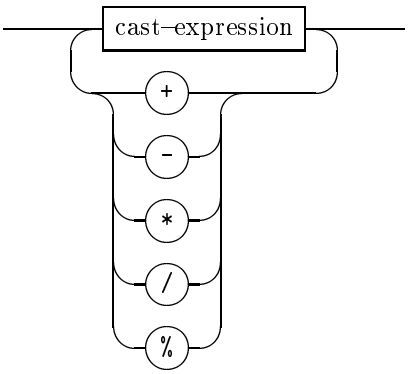
equality-or-relational-expression



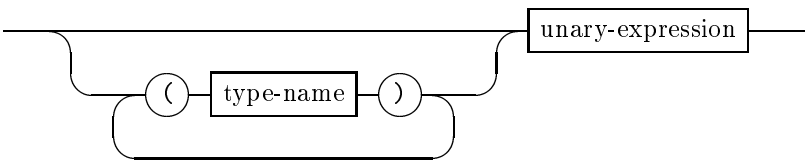
shift-expression



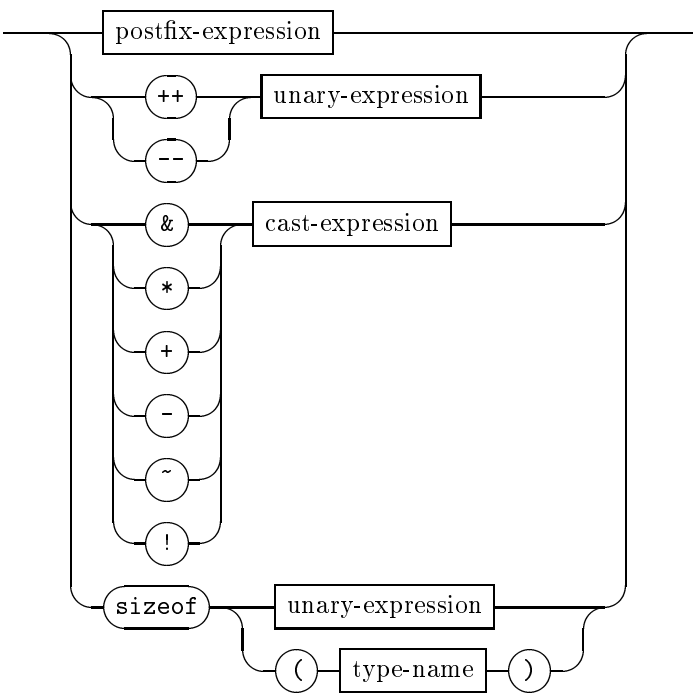
additive-or-multiplicative-expression



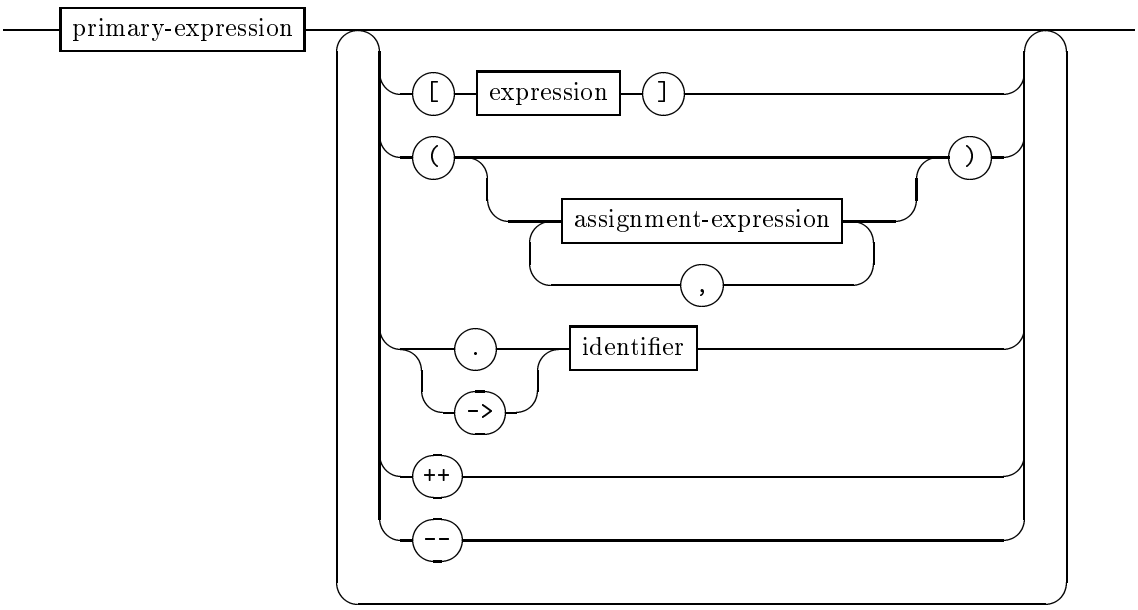
cast-expression



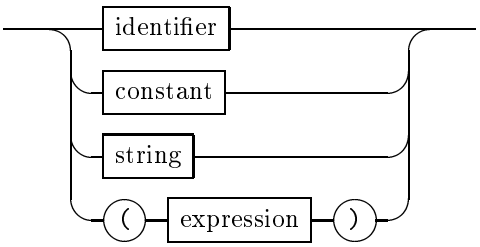
unary-expression



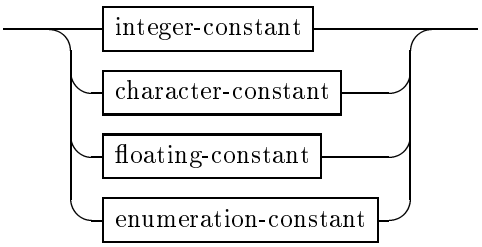
postfix-expression



primary-expression



constant



Annexe C

Caml

Le langage Caml est un langage typé conçu et développé à l'INRIA à partir de 1984. C'est un successeur du langage ML[7] conçu par Robin Milner en 1978 comme un langage typé de manipulation symbolique, qui servait de métalangage au système de démonstration automatique LCF. Outre Caml, le langage ML a eu de multiples descendants en particulier SML [9], développé à AT&T Bell laboratories et CMU. Les langages de la famille ML sont de haut niveau, ils autorisent les définitions algébriques des structures de données et la gestion automatique de la mémoire. Ils 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. À ce titre, Caml est assez différent de Pascal et C, quoiqu'il propose aussi des aspects impératifs qui autorisent un style assez proche du style impératif traditionnel. C'est celui que nous adoptons ici pour rester aussi près que possible des codages C ou Pascal des algorithmes décrits.

On trouve une introduction didactique au langage SML 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.

C.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 `n` × `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.

C.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.

C.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.

C.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.)

C.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 $(1, 2, 3) \neq ((1, 2), 3)$.

C.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 273). 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
```

C.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.

C.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.

C.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.

C.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).

C.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);;
```

C.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 `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.

C.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.

C.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.

C.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 ::= *nconstr 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* `)`
| *nconstr 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.

C.4 Programmes en Caml

Chapitre 1

```
#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 lit le tableau *)
 initialisation ();
 (* 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 tri_sélection* a =
 let min = ref 0 in
 for i = 0 to vect_length a - 2 do
  min := i;
  for j = i + 1 to vect_length a - 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 a = make_vect 10 0 in
 initialisation a;
 tri_selection* a;
 impression a;;

(* Tri bulle, voir page 22 *)
let tri_bulle a =
 for i = vect_length a - 1 downto 0 do
  for j = 1 to i do
   if a.(j - 1) > a .(j) then
    let t = a.(j - 1) in
    a.(j - 1) <- a.(j); a.(j) <- t
  done
 done;;

(* Tri par insertion, voir page 24 *)
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);
   j := !j - 1
  done;
  a.(!j) <- !v;
 done;;

(* Tri shell, voir page 27 *)
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;;

(* Avec une exception pour sortir de
   la boucle dès qu'on a trouvé *)
let bottin =
 [| "paul",   2811;
    "roger",  4501;
    "laure",  2701;
    "anne",   2702;
    "pierre", 2805;
    "yves",   2806 |];;

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 *)
let bottin =
 [| "paul",   2811;
    "roger",  4501;
    "laure",  2701;
    "anne",   2702;
    "pierre", 2805;
    "yves",   2806;
    "",       0    |];;

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;;

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

(* Recherche dichotomique, voir page 30 *)
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 31 *)
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 *)
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 *)
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 32 *)
let nom = make_vect nmax "";;
let tel = make_vect nmax 0;;
let col = make_vect nmax (-1);;

let nNoms = ref n;;

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

(* Recherche avec adressage ouvert,
   voir page 34 *)
let recherche x =
 let i = ref (h x) in
 while nom.(!i) <> x && nom.(!i) <> "" do
   i := (!i + 1) mod nmax
 done;
 if nom.(!i) = x then tel.(!i) else -1;;

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

```
 nom.(!i) <- x;
 tel.(!i) <- val;;
```

Chapitre 2

```
(* Fibonaci, voir page 43 *)
let rec fib n =
 if n <= 1 then 1
 else fib (n - 1) + fib ( n - 2);;

(* Factorielle, voir page 43 *)
let rec fact n =
  if n <= 1 then 1
  else n * fact (n - 1);;

(* Triangle de Pascal, voir page 43 *)
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 45 *)
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 46 *)
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 46 *)
let rec f91 n =
 if n > 100 then n - 10
 else f91 (f91 (n + 11));;

(* La fonction de Morris, voir page 46 *)
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;;

(* La courbe du dragon, voir page 51 *)
#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;;

dragon 15 120 120 50 50;;

(* La courbe du dragon, voir page 52 *)
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 53 *)
let a = [| 1; 7; 3; 8; 0 |];;

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 54 *)
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); i := !i + 1
   end else begin
    a.(k) <- b.(!j); j := !j - 1
   end
  done
 end;;
```

Chapitre 3

Remarque: nous suivons les signatures des procédures C. Les listes étant polymorphes et prédéfinies en Caml, on les utilise plus naturellement sans déclarer un type liste particulier. La manipulation des listes est facilitée en Caml par la gestion automatique de la mémoire: allocation et libération sont implicites. Enfin les listes prédéfinies ne sont pas mutables.

```
type element == int;;

type liste = Nil | Cons of cellule

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

(* Ajouter, voir page 65 *)
let lajouter x ap =
 ap := Cons {contenu = x; suivant = !ap};;
lajouter :
 element -> liste ref -> unit = <fun>

(* Recherche, voir page 65 *)
let lrecherche 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;;
lrecherche : element -> liste -> bool = <fun>

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

(* Longueur d'une liste, voir page 66 *)
let llongueur 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 lsupprimer x ap =
 let rec supprime precedente = function
 | Nil -> ()
 | Cons ({contenu = c; suivant = reste}
         as cellule) ->
    if c = x then precedente.suivant <- reste
    else supprime cellule reste in
 match !ap with
 | Nil -> ()
 | Cons ({contenu = c; suivant = reste}
          as cellule) ->
    if c = x then ap := reste
    else supprime cellule reste;;

(* Liste des nombres premiers, voir page 69 *)
let liste_premier n =
 let a = ref Nil in
 for i = n downto 2 do lajouter 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
      lsupprimer (j * !k) a done;
     b := s
 done;
```

```
 a;;

(* Déclarations et opérations sur les piles,
   voir page 71 *)

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

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

let pvide p = p.hauteur = 0;;

let pajouter x p =
 p.contenu.(p.hauteur) <- x;
 p.hauteur <- p.hauteur + 1;;

let pvaleur p =
 let i = p.hauteur - 1 in
 p.contenu.(i);;

let psupprimer p = p.hauteur <- p.hauteur - 1;;

(* Evaluation des expressions préfixées,
   voir page 72 *)
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 -> pajouter x p
 | Nombre n ->
    if pvide p then pajouter x p else
    match pvaleur p with
    | Symbole c as y -> pajouter x p
    | Nombre m ->
       psupprimer p;
       match pvaleur p with
       | Symbole c ->
         psupprimer 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
  match u.(i) with
  | Nombre _ | Symbole `+`
  | Symbole `*` as x -> inserer x p
  | _ -> ()
 done;
 match pvaleur 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

(* Les files avec des vecteurs,
   voir page 75 *)
let maxf = 100;;

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

let fairefvide f = f.debut <- 0; f.fin <- 0;;

let successeur x = if x = maxf then 1 else x + 1;;

let fvide f = f.fin = f.debut;;

let fpleine f = f.debut = successeur f.fin;;

let fvaleur f = f.contenu.(successeur f.debut);;

let fajouter x f =
 f.contenu.(f.fin) <- sucesseur (f.debut);;

let fsupprimer f = f.debut <- sucesseur (f.debut);;

(* Les files avec des listes, voir page 76 *)

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 fairefvide f =
 let b =
```

```
  {contenu = f.debut.contenu;
   suivant = Nil} in
 f.debut <- b; f.fin <- f.debut;;

let successeur c = c.suivant;;

let fvide f = f.fin == f.debut;;

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

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

let fsupprimer f =
 match successeur f.debut with
 | Nil -> ()
 | Cons cell -> f.debut <- cell;;

(* Tail et cons, voir page 78 *)
let tail = function
| Nil -> failwith "tail"
| Cons cell -> cell.contenu;;

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

(* Append et nconc,
   voir page 79 *)
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
     | _ -> ();;

(* Plus conforme au style Caml:
   on utilise une fonction récursive locale,
   on fait l'opération physique sur la liste,
   mais on rend un résultat *)
let rec nconc a b =
 let rec nconc_aux c =
  match c with
  | Nil -> b
  | Cons ({suivant = Nil; _} as cell) ->
      cell.suivant <- b; a
  | Cons {suivant = l; _} -> nconc_aux l in
 nconc_aux a;;

(* Nreverse et reverse,
   voir page 80 *)
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 habituelle:
   on renverse la liste en place
   et l'on rend la nouvelle tête de liste *)
let nreverse l =
 let rec nreverse_aux a b =
  match a with
  | Nil -> b
  | Cons ({suivant = s; _} as cell) ->
     cell.suivant <- b;
     nreverse_aux 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 80 *)
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

```
(* Ajouter à un tas, voir page 92 *)
type 'a tas =
{ mutable cardinal : int;
  tas : 'a vect };;
```

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

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;;

let a = [| 2; 1; 2; 3; 0 |];;
heapsort a;;
```

```
#a;;
- : int vect = [|0; 1; 2; 2; 3|]

(* Déclaration d'un arbre, voir page 96 *)
(* Cas binaire *)
type 'a arbre = Vide | Noeud of 'a noeud
and 'a noeud =
{contenu : 'a; filsG : 'a arbre; filsD : 'a arbre};;

(* Déclaration d'un arbre, voir page 96 *)
(* Cas n-aire, les fils sont
   implémentés par un vecteur d'arbres. *)
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, voir page 96 *)
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 rec imprimer a tab =
 match a with
 | Vide -> ()
 | Noeud {contenu = c; filsG = fg; filsD = fd} ->
    printf "%3d     " c; imprimer fd (tab + 8);
    if fg <> Vide then
     printf "\n%s" (make_string tab ' ');
    imprimer fg tab;;

let imprimer_arbre a =
  imprimer a 0; print_newline();;

#imprimer_arbre (main ());;
 20      25      28
         21
  3      12      13
          8
          6
  3
- : unit = ()

(* 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 99 *)
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;;

(* Purement fonctionnel *)
let rec ajouter v = function
| 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;;

(* Ajout par effet mémoire *)
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 = function
| Vide -> nouvel_arbre v Vide Vide
| Noeud
   ({contenu = c; filsG = fg; filsD = fd}
     as noeud) as a ->
   if v <= c
   then noeud.filsG <- ajouter v fg
   else noeud.filsD <- ajouter v fd;
   a;;

(* Exemple *)
let a1 = ajouter 1 Vide in
let a2 = ajouter 2 a1 in
let a3 = ajouter 0 a2 in
ajouter 3 a3;;

(* On ne modifie que lorsque nécessaire,
   c'est-à-dire pour remplacer
   un arbre vide par un arbre non vide. *)
let rec ajouter v a =
 match a with
 | Vide -> nouvel_arbre v Vide Vide
 | Noeud
    ({contenu = c;
      filsG = fg;
      filsD = fd} as noeud) when v <= c ->
    let fg' = ajouter v fg in
    if fg = Vide then noeud.filsG <- fg';
    a
 | Noeud
    ({contenu = c;
      filsG = fg;
      filsD = fd} as noeud) ->
    let fd' = ajouter v fd in
    if fd = Vide then noeud.filsD <- fd';
    a;;

(* Ajout dans un AVL, voir page 101 *)

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 };;

#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";;

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

let rotD = function
| 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;;

let rotG = function
| 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;;

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 (nouvel_arbre, incr) =
         ajouter v fg in
        noeud.balance <-
          noeud.balance - incr;
        if fg = Vide
        then noeud.filsG <- nouvel_arbre;
        incr
       end else begin
         let (nouvel_arbre, incr) =
          ajouter v fd in
         noeud.balance <-
          noeud.balance + incr;
         if fd = Vide
         then noeud.filsD <- nouvel_arbre;
         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);;
```

Chapitre 5

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

(* La matrice d'adjacence du graphe
   donné dans l'exemple.
   Attention les sommets sont numérotés
   à partir de 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 lignes m = vect_length m
and colonnes m =
 if vect_length m = 0 then failwith "colonnes"
 else vect_length m.(0);;

(* Calcule le produit des matrices m1 et m2 *)
let multiplier m1 m2 =
 if colonnes m1 <> lignes m2
 then failwith "multiplier" else
 let résultat =
  make_matrix (lignes m1) (colonnes m2) 0 in
 for i = 0 to lignes m1 - 1 do
  for j = 0 to colonnes m2 - 1 do
   let aij = ref 0 in
   for k = 0 to colonnes m1 - 1 do
     aij := m1.(i).(k) * m2.(k).(j) + !aij;
   done;
   résultat.(i).(j) <- !aij
  done
 done;
 résultat;;

(* Calcule la somme des matrices m1 et m2 *)
let ajouter m1 m2 =
 if colonnes m1 <> lignes m2
 then failwith "ajouter" else
 let résultat =
  make_matrix (lignes m1) (colonnes m2) 0 in
 for i = 0 to lignes m1 - 1 do
  for j = 0 to colonnes m2 - 1 do
   résultat.(i).(j) <- m1.(i).(j) + m2.(i).(j)
  done
 done;
 résultat;;

(* 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));;

(* On connaît le nombre de chemins
   d'ordre i en calculant puissance m i *)
let nombre_de_chemin_de_longueur_n n i j m =
 (puissance m n).(i).(j);;

(* Il y a deux chemins de longueur 2
   entre les sommets 0 et 5
  (sommets 1 et 6 de la figure 5.3) *)
#nombre_de_chemin_de_longueur_n 2 0 5 g;;
- : int = 2

(* Calcule la somme des puissances n-ième
   (pour n <= i) de la matrice m:
   sigma i m donne le nombre de chemins
```

```
   de longueur inférieure à i *)
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 (colonnes m) m).(i).(j) <> 0;;

#existe_chemin 2 1 g;;
- : bool = true

(* Si u est en relation avec x,
   alors on ajoute les arcs u,v
   pour tous les v successeurs de x *)
let phi m x =
 for u = 0 to colonnes m - 1 do
  if m.(u).(x) = 1 then
   for v = 0 to colonnes m - 1 do
    if m.(x).(v) = 1 then m.(u).(v) <- 1
   done
 done;;

let fermeture_transitive m =
 (* Allocation du résultat *)
 let résultat =
  make_matrix (lignes m) (colonnes m) 0 in
 (* Recopie de m dans le résultat *)
 for i = 0 to lignes m - 1 do
  for j = 0 to colonnes m - 1 do
   résultat.(i).(j) <- m.(i).(j)
  done
 done;
 (* Pour tous les sommets on applique phi *)
 for x = 0 to colonnes m - 1 do
  phi résultat x done;
 résultat;;

(* On accède transitivement à tous les sommets,
   sauf au sommet 0 *)
#fermeture_transitive g;;
- : int vect vect =
[|[|0; 1; 1; 1; 1; 1|]; [|0; 1; 1; 1; 1; 1|];
  [|0; 1; 1; 1; 1; 1|]; [|0; 1; 1; 1; 1; 1|];
  [|0; 1; 1; 1; 1; 1|]; [|0; 1; 1; 1; 1; 1|]|]

(* Listes de successeurs *)

type graphe_point == (int list) vect;;

let omega = -1;;

let transforme_mat_suc m =
 let succ =
   make_matrix (lignes m) (colonnes m) 0 in
 let k = ref 0 in
 for i = 0 to lignes m - 1 do
  k := 0;
  for j = 0 to 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,
   voir page 118 *)
#transforme_mat_suc g;;
- : int vect vect =
[|[|1; 2; -1; 0; 0; 0|]; [|2; 3; 5; -1; 0; 0|];
  [|5; -1; 0; 0; 0; 0|]; [|4; -1; 0; 0; 0; 0|];
  [|1; -1; 0; 0; 0; 0|]; [|3; -1; 0; 0; 0; 0|]|]

(* Transforme une matrice d'adjacence
   en vecteur de listes de successeurs *)
let transforme_mat_list_suc m =
 let gpoint = make_vect (colonnes m) [] in
 for i = 0 to lignes m - 1 do
  for j = 0 to colonnes m - 1 do
   if m.(i).(j) = 1
   then gpoint.(i) <- j :: gpoint.(i)
  done
 done;
 gpoint;;

#transforme_mat_list_suc g;;
- : int list vect =
 [|[2; 1]; [5; 3; 2]; [5]; [4]; [1]; [3]|]
```

Chapitre 6

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

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 erreur (i) end
  else
  if f.[!i] = `a\
  then begin
```

```
   incr i;
   Feuille `a\ end
  else erreur (i);;

(* Prédicat définissant les non-terminaux *)
let 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_récursive 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 begin
  print_string "analyse réussie \n";
  true end
 else
   let y = u.[!pos] in
   if 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);
     if !b
     then printf "règle %d du non-terminal %c \n" !i y
     else incr i
    done;
    !b end
   else false;;

(* Analyse LL(1), voir page 159 *)
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 162 *)
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;;
```

Annexe D

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 le Vax 9000 ou les stations des salles Dec, HP et Sun pour les élèves. 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 élémentaires, un approfondissement pour apprendre les commandes élémentaires. Enfin, le béaba du réseau figure dans une troisième section.

D.1 Trousse de Survie

D.1.1 Se connecter

- Depuis les stations Alpha, Hp ou Sun et les terminaux X des salles de TD: taper son nom de login, puis son mot de passe.

- Depuis une autre machine reliée au réseau Internet: taper `telnet 129.104.252.1` ou simplement `telnet poly.polytechnique.fr` [1], puis login/password comme précédemment.

- Par Minitel (il faut un Minitel modèle 1B ou 2): se connecter au 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 ou Sun à la place de `poly`.) Puis login/password comme précédemment.

- Par terminal ou émulation terminal, on peut aussi appeler le 69 33 38 38: il faut règler ses paramètres de communication à 8 bits, sans parité, sans Xon/Xoff, 1 bit stop, puis `login/passwd` comme précédemment.

[1] A l'extérieur de l'X, il vaut mieux utiliser `telnet 192.48.98.14`, ou `telnet sil.polytechnique.fr`

D.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.

D.1.3 Le système de fenêtres par défaut

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	désiconifier	
Fond d'écran	Applications Xlock Elm (courrier) Editeurs de texte Communication Applications ⋮		Sessions Xterm poly (Vax 9000) sil (vers l'Internet) Stations DEC Stations SUN ⋮ 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
`sil`	ouverture d'une fenêtre shell sur sil
`Editeurs de texte`	éditeur de textes (emacs, notepad, Xedit, vi)
`Elm`	pour lire et envoyer du courrier
`Communication`	pour discuter à plusieurs ou lire les *news*

[2]Cet environnement est celui qui a été installé par Dominique Moret et Laurent de Munico, mais il est bien sûr entièrement reconfigurable avec un peu d'habileté.

D.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.

`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`, `polyaf`, `maple`), on peut souvent obtenir de l'aide en tapant `?` ou `h`. Enfin, on peut aussi poser des questions aux utilisateurs habituels de la salle Sun ou Dec; certains en savent très long.

D.1.5 Changer son mot de passe

La commande est `passwd`. Critères de choix d'un mot de passe:

- au moins 6 caractères (mais 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.

D.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 `login` pour du courrier local, ou `login@machine` pour du courrier vers l'extérieur. Le `login` est le nom de login du destinataire. (Certains sites admettent aussi le nom complet du destinataire, sous la forme `prénom.nom`.) La `machine` est le nom complet de la machine de destination. Il ne faut pas mettre d'espaces dans une adresse de courrier: `login @ machine` ne marche pas. Votre adresse de courrier sur les machines de l'X est:

nom-de-login`@poly.polytechnique.fr`

ou

prénom.nom`@polytechnique.fr`

On omet les accents; on met un tiret - pour les noms composés; on met un caractère souligné _ pour représenter un blanc ou une apostrophe. 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`

(Si vous n'êtes pas sûrs 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

D.1.7 Polyaf

Un système de messagerie électronique inter-écoles peut être obtenu par la commande `polyaf`. (Il est recommandé au début d'utiliser la commande `z` de remise à zéro pour ne pas avoir trop de messages à lire). Résumé des commandes:

`h`	aide en ligne
`q`	quitter `polyaf`
`l`	montre la liste des groupes
`g` *groupe*	va dans le groupe de nom *groupe*
`g`	va au prochain groupe avec des nouveaux messages
`Return`	montre le prochain message
$-$	montre le message précédent
n	montre le message numéro n
`r`	donne un résumé des messages du groupe courant
`m`	rentre un nouveau message
`R`	rentre une réponse au message courant
`z`	marque lus tous les messages du groupe courant

D.1.8 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:

←, →, ↑, ↓	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 ◇ ou `esc`)
`Control-X Control-S`	écrit le fichier modifié
`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.

D.1.9 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*.

D.1.10 Cycles de mise au point

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.
`emacs prog.c`	On corrige le source et on recommence jusqu'à ce que ça marche.

Textes LaTeX

`emacs memoire.tex`	On crée le texte source.
`latex memoire.tex`	On compose 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 que ça soit correct.
⋮	
`dvips -o memoire.ps memoire.dvi`	On transforme le fichier `memoire.dvi` en un fichier PostScript
`lpr -Pps20 memoire.ps`	On imprime le résultat sur l'imprimante Vax recto-verso
`lpr -Psun memoire.ps`	Itou sur l'imprimante de la salle Sun non recto-verso
`lprm -Psun` *user*	Pour stopper l'impression de *user*

D.1.11 Types de machines

Il y a 4 types de machines Unix à l'X: Vax 9000, Hp, Sun et stations Alpha. Chacune a un processeur différent: vax pour le 9000, HP-PA pour les Hp, sparc pour les Sun, Alpha pour les stations Alpha. Les fichiers exécutables sont donc différents sur ces 4 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 32 terminaux X qui n'ont pas de processeur et qui peuvent atteindre toute machine sur le réseau.

D.2 Approfondissement

D.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
```

- `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`

D.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).

D.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.

D.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`.

D.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. 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
```

D.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`.

D.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
```

D.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:

```
#!/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`.)

[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.

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 lancée 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).

D.3 Unix et le réseau de l'X

Le réseau Internet relie 6,6 millions de machines ou de réseaux locaux dans le monde en juillet 95, soit 40 millions de personnes. 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 labos et pour l'administration. Le réseau des élèves relie les chambres, les salles de TD (salles Dec, Sun ou Hp), la salle DEA, le vax poly et 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 vraisemblablement à 100 Mbit/s. Dans une salle de TD, les stations Unix et les imprimantes sont déjà

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 en dur (`129.104.252.1` pour le vax poly) ou symbolique (`poly.polytechnique.fr` pour le vax poly) qui sont les mêmes adresses que pour le courrier électronique. A l'intérieur de l'X, le suffixe `polytechnique.fr` est inutile. Les stations Alpha ont des noms d'os (`radius`, `cubitus`, ...), les stations Hp des noms de poissons (`carpe`, `lieu`, ...), les stations Sun des noms de voitures (`ferrari`, `bugatti`, ...). 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 des bibliothèques multi-media du *World Wide Web*. Quelques adresses pour commencer:

`http://www.polytechnique.fr/`	le serveur officiel de l'X,
`http://www.polytechnique.fr/~www/`	le serveur des élèves 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://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/`	le système Inferno,
`http://www.lmet.fr/Homefr.html`	la librairie du Monde en Tique,
`http://www.technical.powells.portland.or.us/`	*Powell's Technical Books*,
`http://www.radio-france.fr/`	Radio France,
`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 vrai Louvre
`http://www.cnam.fr/louvre/`	et celui de Nicolas Pioch,
`http://www.urec.fr/France/cartes/France.html`	le Web en France
`http://www.yahoo.com/`	ou sur la planète,
`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.

D.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] David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmeticc*, Computing Surveys, 23(1), Mars 1991.

[14] Gaston H. Gonnet, Riccardo Baeza-Yates, *Handbook of Algorithms and Data Structures, In Pascal and C*, Addison Wesley, 1991.

[15] 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.

[16] Ron L. Graham, Donald E. Knuth, Oren Patashnik, *Concrete mathematics: a foundation for computer science*, Addison Wesley, 1989.

[17] Samuel P. Harbison, *Modula-3*, Prentice Hall, 1992.

[18] John H. Hennessy, David A. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishers, Inc. , 1990.

[19] Kathleen Jensen, Niklaus Wirth, *PASCAL user manual and report : ISO PASCAL standard*, Springer, 1991. (1ère édition en 1974).

[20] Gerry Kane, *Mips, RISC Architecture*, MIPS Computer Systems, Inc., Prentice Hall, 1987.

[21] 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.

[22] Brian W. Kernighan, *PIC–a language for typesetting graphics*, Software Practice & Experience 12 (1982), 1-20.

[23] Dick B. Kieburtz, *Structured Programming And Problem Solving With Algol W*, Prentice Hall, 1975.

[24] Stephen C. Kleene, *Introduction to Metamathematics*, North Holland, 6ème édition, 1971. (1ère en 1952).

[25] Donald E. Knuth, *The TEXbook*, Addison Wesley, 1984.

[26] Donald E. Knuth, *The Metafont book*, Addison Wesley, 1986.

[27] Donald E. Knuth, *Fundamental Algorithms. The Art of Computer Programming*, vol 1, Addison Wesley, 1968.

[28] Donald E. Knuth, *Seminumerical algorithms, The Art of Computer Programming*, vol 2, Addison Wesley, 1969.

[29] Donald E. Knuth, *Sorting and Searching. The Art of Computer Programming*, vol 3, Addison Wesley, 1973.

[30] Leslie Lamport, LaTeX, *User's guide & Reference Manual*, Addison Wesley, 1986.

[31] 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.

[32] Jan van Leeuwen, *Handbook of theoretical computer science*, volumes A et B, MIT press, 1990.

[33] M. Lothaire, *Combinatorics on Words*, Encyclopedia of Mathematics, Cambridge University Press, 1983.

[34] Udi Manber, *Introduction to Algorithms, A creative approach*, Addison Wesley, 1989

[35] Bob Metcalfe, D. Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks*, Communications of the ACM 19,7, Juillet 1976, pp 395–404.

[36] Robin Milner, *A proposal for Standard ML*, In ACM Symposium on LISP and Functional Programming, pp 184-197, 1984, ACM Press, New York.

[37] Robin Milner, Mads Tofte, Robert Harper, *The definiton of Standard ML*, The MIT Press, 1990.

[38] Greg Nelson, *Systems Programming with Modula-3*, Prentice Hall, 1991.

[39] Eric Raymond, *The New Hacker's Dictionary*, dessins de Guy L. Steele Jr., MIT Press 1991.

[40] Brian Randell, L. J. Russel, *Algol 60 Implementation*, Academic Press, New York, 1964.

[41] Martin Richards, *The portability of the BCPL compiler*, Software Practice and Experience 1:2, pp. 135-146, 1971.

[42] Martin Richards, Colin Whitby-Strevens, *BCPL : The Language and its Compiler*, Cambride University Press, 1979.

[43] 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).

[44] Hartley Rogers, *Theory of recursive functions and effective computability*, MIT press, 1987, (édition originale McGraw-Hill, 1967).

[45] A. Sainte-Laguë, *Les réseaux (ou graphes)*, Mémoire des Sciences Mathématiques (18), 1926.

[46] Bob Sedgewick, *Algorithms*, 2nd edition, Addison-Wesley, 1988. En français: *Algorithmes en langage C*, trad. par Jean-Michel Moreau, InterEditions, 1991.

[47] Ravi Sethi, *Programming Languages, Concepts and Constructs*, Addison Wesley, 1989.

[48] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, 1986.

[49] Robert E. Tarjan, *Depth First Search and linear graph algorithms*, Siam Journal of Computing, 1, pages 146-160, 1972.

[50] 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).

[51] Pierre Weis, *The Caml Reference Manual, version 2-6.1*, Rapport technique 121, INRIA, Rocquencourt, 1990.

[52] Pierre Weis, Xavier Leroy, *Le langage Caml*, InterEditions, 1993.

Table des figures

i.1 Bornes supérieures des nombres entiers . . . 15
i.2 Valeurs spéciales pour les flottants IEEE . . . 16
i.3 Formats des flottants IEEE . . . 16

1.1 Exemple de tri par sélection . . . 21
1.2 Exemple de tri bulle . . . 23
1.3 Exemple de tri par insertion . . . 25
1.4 Un exemple de table pour la recherche en table . . . 27
1.5 Hachage par collisions séparées . . . 33
1.6 Hachage par adressage ouvert . . . 35

2.1 Appels récursifs pour `Fib`(4) . . . 45
2.2 Les tours de Hanoi . . . 49
2.3 Flocons de von Koch . . . 51
2.4 La courbe du Dragon . . . 52
2.5 Partition de Quicksort . . . 54

3.1 Ajout d'un élément dans une liste . . . 64
3.2 Suppression d'un élément dans une liste . . . 65
3.3 Pile d'évaluation des expressions . . . 73
3.4 File gérée par un tableau circulaire . . . 76
3.5 File d'attente implémentée par une liste . . . 77
3.6 Queue d'une liste . . . 78
3.7 Concaténation de deux listes par *Append* . . . 79
3.8 Concaténation de deux listes par *Nconc* . . . 79
3.9 Transformation d'une liste au cours de *Nreverse* . . . 80
3.10 Liste circulaire gardée . . . 81

4.1 Un exemple d'arbre . . . 89
4.2 Représentation d'une expression arithmétique par un arbre . . . 90
4.3 Représentation en arbre d'un tas . . . 91
4.4 Représentation en tableau d'un tas . . . 92
4.5 Ajout dans un tas . . . 93
4.6 Suppression dans un tas . . . 93
4.7 Exemple d'arbre de décision pour le tri . . . 95
4.8 Ajout dans un arbre de recherche . . . 99
4.9 Rotation dans un arbre AVL . . . 101
4.10 Double rotation dans un arbre AVL . . . 102
4.11 Exemple d'arbre 2-3-4 . . . 103
4.12 Eclatement d'arbres 2-3-4 . . . 109

4.13 Arbres bicolores 109

5.1 Le graphe de De Bruijn pour $k = 3$ 112
5.2 Le graphe des diviseurs, $n = 12$ 113
5.3 Un exemple de graphe et sa matrice d'adjacence 114
5.4 Un graphe et sa fermeture transitive 115
5.5 L'effet de l'opération Φ_x: les arcs ajoutés sont en pointillé 116
5.6 Une arborescence et son vecteur *pere* 121
5.7 Une arborescence préfixe 122
5.8 Emboitement des descendants dans une arborescence préfixe 122
5.9 Une arborescence des plus courts chemins de racine 10 124
5.10 Exécution de l'algorithme de Trémaux 127
5.11 Les arcs obtenus par Trémaux 129
5.12 Composantes fortement connexes du graphe de la figure 5.11 130
5.13 Un exemple de sous-arborescence 133
5.14 Les points d'attaches des sommets d'un graphe 134

6.1 Arbre de dérivation de *aabbabab* 153
6.2 Arbre de dérivation d'une expression arithmétique 154
6.3 Arbre de syntaxe abstraite de l'expression 154

7.1 File de caractères 168
7.2 Adresse d'un caractère par base et déplacement 169
7.3 Compilation séparée 173
7.4 Dépendances dans un *Makefile* 176
7.5 Un exemple de graphe acyclique 177

8.1 Un graphe symétrique et l'un de ses arbres recouvrants 184
8.2 Un arbre recouvrant de poids minimum 185
8.3 Huit reines sur un échiquier 187
8.4 Un graphe aux arcs valués 190

A.1 Le code ASCII en hexadécimal 201
A.2 Les boîtes de dialogue 212

B.1 Conversions en C 239

Index

`*` , 280
`++` , 235, 240
`--` , 240
`->` , 251
`;` , 235, 243, 283
`_` , 284

`abs`, 277
Ackermann, 46
affectation, 205, 235, 240, 242
ajouter
    dans une file, 74
    dans une liste, 64
    dans une pile, 70
aléatoire, 20, 37
AlgolW, 195
Alto, 9
analyse
    ascendante, 161
    descendante, 153
analyse syntaxique, 145
ancêtre, 120
ANSI, 10
appel par référence, 198, 203, 204, 235, 247
appel par valeur, 47, 204, 281
arborescence, 120
    de Trémaux, 126
    des plus courts chemins, 124
    préfixe, 121
    sous-arborescence, 132
arbre, 89
    implémentation, 96
    impression, 97
arbre binaire, 89
arbre de recherche, 98
arbres équilibrés, 100
    arbres 2-3, 103
    arbres 2-3-4, 103
    arbres AVL, 100
    arbres bicolores, 103
    rotations, 101
arc, 111
    de retour, 129, 133
    transverse, 129, 133
argument fonctionnel, 209, 277
ASCII, 201
attache, 133

`begin`, 206, 283
Bentley, 55
binaires relogeables, 173
bloc, 204, 282
BNF
    C, 255
    Caml, 298
    Pascal, 220
`bool`, 279
`boolean`, 199
booléens, 199, 279
`break`, 244

C++, 10
Caml, 5, 271, 335
caractères, 199, 236, 279
carré magique, 195, 233, 271
`case`, 206
cast, 239
chaînage, 64
chaîne de caractères, 198, 207, 208, 218, 236, 238, 248, 279
`char`, 199, 236, 279
chemin, 112
    calcul, 128
    plus court, 124
collision, 32
compilation, 145
compilation séparée, 172
composante
    fortement connexe, 130, 135
conversions, 238, 239
    explicites, 239
courbe du dragon, 50

De Bruijn, 112

Depth First Search, 126
dérivation, 147
descendant, 120
dessins, 216, 253, 295
diagrammes syntaxiques
    C, 261
    Pascal, 224
`dispose`, 216
Divide and Conquer, 56, 57
`do`, 244
`double`, 236
dragon, 50

effet de bord, 203, 241, 243
égalité de type, 205, 250
`end`, 206, 283
`End_of_file`, 286
enregistrement, 213, 289
ensembles, 63
entiers, 15, 236, 279
    non signés, 237
`enum`, 237
`EOF`, 252
`eof`, 211
`eoln`, 211
Eratosthène, 69
erreurs, 285
évaluation d'expressions, 73
exceptions, 285
`exit`, 235
expressions, 201, 280
    affectation, 240, 242
    bits, 241
    conditionnelles, 242
    évaluation, 202, 238, 242, 281
    incrémentation, 240
expressions arithmétiques, 149

factorielle, 43
`fclose`, 252
fermeture transitive, 115
    calcul, 117
    exemple, 115
feuille, 89
Fibonacci, 43
    itératif, 45
fichier, 210, 252, 285
`FILE`, 252
file, 74, 125
    d'attente, 74
    de caractères, 167
    de priorité, 90
    gardée, 76
`filepos`, 211
fils, 120
`float`, 236, 279
flocon de von Koch, 50
fonction, 202, 245, 274
fonction 91, 46
fonction de Morris, 46
`fopen`, 252
`for`, 206, 235, 244, 284
fractales, 50
`free`, 248, 251
`fseek`, 253
fusion, 56

`get`, 212
`getc`, 252
`getchar`, 252
glouton, 181
Gödel, 47
`goto`, 207, 245
grammaires, 147
graphe, 111
    de De Bruijn, 112
    fortement connexe, 131
    orienté, 111
    symétrique, 111
graphique, 216, 253, 295

hachage, 31
    adressage ouvert, 34
    multiple, 36
hacker, 7
Hanoi, 48
heap, 90, 216
Heapsort, 94
Hennessy, 9
Hoare, 53

identificateur, 199, 236, 278
IEEE, 16
`if`, 205, 243
`in`, 209
incrémentation, 235, 240
indécidable, 47
`input`, 197
instruction vide, 207
`int`, 236, 279
interclassement, 56
interface, 78, 170, 171

Kernighan, 9, 10, 233
Kleene, 50
Knuth, 8
Koch, 50
Kruskal, 183

l'équation de C, 248
LaTeX, 8
`let rec`, 277
librairies, 173
liste, 64
    de successeurs, 118
    de successeurs
      chaînée, 119
    des nombres premiers, 69
    gardée, 68
    image miroir, 79
    vide, 64
LL(1), 158
`long`, 236
LR(1), 162

MacCarthy, 46
`main`, 249
majuscules, 199
Makefile, 176
`malloc`, 248, 251
Maple, 5, 12, 27
`match`, 284, 290
`math.h`, 247
matrice
    d'adjacence, 113
    produit, 115
`maxint`, 15, 197, 199
menu déroulant, 212
Milner, 271
minuscules, 199
ML, 10, 271
module, 78, 170, 171
Morris, 46
mots clés, 199, 278
`mutable`, 289

n-tuplets, 280
nœud, 89
nœud interne, 89
Nelson, 10
`new`, 216
`NewFileName`, 212
`nil`, 215, 248
nombre aléatoire, 20, 37
nombres flottants, 16
`NULL`, 248
numérotation
    de Trémaux, 132
    préfixe, 123

`OldFileName`, 212
Omega, 118
ω, 118
`open`, 211
`ord`, 200
ordre infixe, 100
ordre postfixe, 100
ordre préfixe, 100
`otherwise`, 206
`output`, 197

`packed array`, 207
parcours
    en largeur, 125
    en profondeur, 126
Patterson, 9
père, 120
pile, 70, 128
    vide, 70
pile Pascal, 216
plus courts chemins, 188
point d'attache, 133, 134
point-virgule en C, 235, 243
pointeur, 215, 247, 248
polymorphisme, 291
portée des variables, 204, 282
postfixée
    notation, 72
PostScript, 9
précédence des opérateurs, 242
prédécesseur, 112
préprocesseur, 234, 260
`printf`, 234, 235, 251, 275, 285
procédure, 202, 245, 275
profondeur, 120
programmation dynamique, 188
`put`, 212
`putc`, 252
`putchar`, 252

QuickDraw, 216, 218, 253
Quicksort, 53

racine, 89, 120
`raise`, 285
`rand`, 37
Randell, 210

`random`, 20
`read`, 210
`read_line`, 275
`readln`, 198, 210
`real`, 199
recherche
  dans une liste, 65
  dichotomique, 29
  en table, 27
  par interpolation, 30
`record`, 213, 289, 290
récursivité croisée, 52
réels, 16, 199, 236, 279
`ref`, 273, 280
  affectation, 273
références, 280
`repeat`, 206
`reset`, 210
résultat d'une fonction, 203, 245
`return`, 245
`rewrite`, 211
rien, 279
RISC, 10
Ritchie, 10, 233
Rogers, 50
Russel, 210

sac à dos, 185
`scanf`, 234, 251
Scheme, 10
Sedgewick, 10
`seek`, 211
sentinelle, 26, 28
`short`, 236
`sizeof`, 251
sommet, 111
sous-séquences, 191
`spell`, 37
`srand`, 37
Stallman, 9
`string`, 208, 279
`struct`, 250
structures, 250
successeur, 112
supprimer
  dans une file, 74
  dans une liste, 66
  dans une pile, 71
`switch`, 243
syntaxe
  abstraite, 153
  C, 255, 261
  Caml, 298
  concrète, 152
  Pascal, 220, 224

tableaux, 247, 248
  dimension, 235
  taille, 197, 274
Tarjan, 126, 130
tas, 90
tas Pascal, 216
TEX, 8
`text`, 211
TGiX, 216, 253
tours de Hanoi, 48
`trace`, 283
Trémaux, 126
tri
  borne inférieure, 94
  bulle, 22
  fusion, 56
  Heapsort, 94
  insertion, 24
  Quicksort, 53
  sélection, 20
  Shell, 26
  topologique, 177
triangle de Pascal, 44
`try`, 285
Turing, 47
`type`, 287
  chaîne, 207
  déclaré, 204, 250, 290
  enregistrement, 213, 250, 289
  ensemble, 208
  énuméré, 200, 237, 287
  fichier, 210, 252
  intervalle, 200
  ordinal, 200
  pointeur, 215, 247
  polymorphe, 291
  somme, 287
  string, 279
  structure, 250
  tableau, 202, 247
  union, 250, 287
type abstraits, 290
`typedef`, 250

`union`, 250, 287
`unit`, 279

Unix, 9
`unsigned`, 237

variables
    globales, 204, 246, 282
    initialisation, 246
    locales, 204, 246, 282
    locales statiques, 246
`vect`, 279
vecteur
    pere, 120
vecteurs, 279
virgule fixe, 15
virgule flottante, 16
`void`, 235, 245

`while`, 206, 244, 285
Wirth, 10, 195
`with`, 214
World Wide Web, 5, 330
`write`, 198, 199
`writeln`, 199