École Centrale Paris

1^{iere} année d'études

Algorithmique

Jean-Jacques Dhénin

1998–1999

Ce document a été composé sous Unix [1] par le logiciel TEX (domaine public). Les figures ont été dessinées sous X Window (domaine public) avec le logiciel `xfig` (domaine public) et intégrées directement dans le document final. L'impression a été réalisée sur une imprimante à laser.

1. Unix est une marque déposée des Laboratoires Bell d'ATT

Table des matières

Première partie

La programmation structurée
Itération, récurrence et récursivité

Algorithme

Algorithme [2]

n. m., XIII^{e}s., voir § III.

I. (a) **Un algorithme est une succession de manœuvres à accomplir toujours dans le même ordre et de la même façon, manœuvres qui sont en nombre fini et s'appliquent à un nombre fini de données.**

Par exemple, on connaît depuis l'enfance des algorithmes de calcul, ceux qui permettent de trouver ce que vaut la somme, le produit, la différence ou le quotient de deux nombres quand ils s'écrivent avec plus d'un chiffre. Il ne s'agit pas d'autre chose quand on entend dire « je ne sais plus faire une division », par exemple celle de 793 par 32 ; en fait, c'est l'algorithme de calcul du quotient qu'on n'a plus en mémoire, c'est-à-dire le « j'ai deux chiffres au diviseur, j'en prends deux au dividende, je dis en 79 combien de fois 32 ou en 7 combien de fois 3, il y va deux fois, etc »

(b) Les livres donnent volontiers comme exemples d'algorithmes ceux qui consistent à trouver le nouveau prix d'un objet s'il y a au moment de l'achat une baisse ou une hausse de, mettons, 10 % ; si le prix initial est x, la hausse ou la baisse est les 10/100 de x, soit $0,1x$; le nouveau prix est donc :

— en cas de baisse : $x - 0,1x = 0,9x$,
— en cas de hausse : $x + 0,1x = 1,1x$.

(c) Un algorithme est généralement répétitif ; c'est le cas si on place une somme d'argent à la banque à un taux de 4 % ; au bout d'une année, elle devient les 104/100 de ce qu'elle était, au bout de deux années les 104/100 des 104/100 de ce qu'elle était, etc.

En désignant cette somme par S, on a donc, au bout de n années, une somme :

$$S_n = \underbrace{(104/100)(104/100)\dots(104/100)}_{n\text{fois}} S = 1,04^n S$$

II. On désigne par algorithme un procédé automatique que l'on peut confier à un ordinateur et qu'il répétera autant de fois qu'il le faudra pour arriver au résultat. Imaginons qu'on l'ait programmé pour la suite d'opérations suivantes : à partir d'un entier naturel n quelconque, si n est pair le diviser par 2 ; s'il est impair, prendre son triple et ajouter 1, c'est-à-dire fabriquer $3n + 1$; dans chaque cas, recommencer avec le nouveau nombre obtenu. Essayons avec quelques nombres :

Nombre de départ												
16	8	4	2	1								
17	52	26	13	40	20	10	5	16	8	4	2	1
18	9	28	14	7	22	11	34	17	...			
19	58	29	88	44	22	11	34	17	...			

Il est clair qu'en arrivant à 1, le processus va 'se boucler' sur lui-même, puisqu'on aura la suite 1, 4, 2, 1, 4, 2, 1, etc. On voit que c'est ce qui se produit pour 16 et 17 ; or, pour 18 et 19, on retombe sur 17, donc on arrivera aussi à 1.

La **conjecture** qui s'établit à partir de plusieurs tentatives qui, toutes, amènent à 1 est que, quel que soit le nombre de départ et le nombre d'étapes, cet algorithme produira toujours 1 ; mais elle n'est, malgré son apparente simplicité, toujours pas démontrée aujourd'hui. Ce 'beau' problème s'appelle "le problème de Collatz", du nom du professeur de Hambourg qui l'a lancé [G$_{17}$].

III. *Algorithme* s'est d'abord dit *algorisme*, du bas latin *algorismus*, déformation d'après le mot grec *arithmos*, "nombre", du nom propre Al-Khwarizmi.

2. *Dictionnaire de mathématiques élémentaires* Stella Baruk, Seuil, p. 77

Sept épatant [1]

Le véritable problème fut posé quand le père Mathieu revint de la foire, poussant devant lui les vingt-huit moutons acquis le matin même. Jusqu'alors, les opérations s'étaient déroulées sans aucune difficulté. Mais il fallait maintenant répartir ces vingt-huit bêtes dans les sept bergeries que comportait la ferme, et ça, croyez-en le père Mathieu, ce n'était pas une mince affaire.

– Toine, dit-il à son fils aîné, tu vas me prendre ces vingt-huit bêtes et me les installer dans nos sept bergeries. T'en mettras le même nombre dans chacune.

– Et ça en fait combien donc dans chaque ? Questionna le Toine.

– Décidément, Toine, t'es pas bien futé. Apprends que, pour faire un partage, on pose une division. Tiens prends une feuille de papier, je vas te montrer.

Et le père Mathieu expliqua au Toine les subtilités de l'opération :

– Vingt-huit divisé par sept : en 8 combien de fois 7 ? Il y va une fois. Une fois sept fait 7 ; ôté de 8 il reste 1. J'abaisse le 2. En 21 combien de fois 7 ? Il y va 3 fois. 3 fois 7 font 21 ; ôté de 21, il reste 0. Tu mettras donc 13 moutons dans chaque bergerie.

28	7
21	13
0	

– Bien, père, fit le Toine, convaincu par la science.

Il partit incontinent, pour procéder à la répartition. Une heure plus tard, Mathieu le vit revenir tout piteux :

– J'y arrive pas, père. Il doit y avoir une erreur.

– Écoute-moi bien, lui dit son père. Y a pas d'erreur possible. D'ailleurs pour te le prouver, on va procéder autrement. Je t'ai dit 13 moutons dans chaque bergerie. Si on multiplie 13 par 7, on doit retrouver les 28 têtes. Allons-y :

Treize multiplié par sept : 7 fois 3 font 21 ; et 7 fois 1 fait 7. Tu vois que 21 et 7, ça fait bien 28.

D'ailleurs, pour être plus sûr, on va faire la preuve par neuf :

3 et 1 font 4. Je pose 4 en haut et j'écris 7 en dessous. 7 fois 4 font 28. 8 et 2 font 10. J'écris 1 à gauche. Maintenant le résultat : 8 et 2 font 10. J'écris 1 à droite. Tu vois bien que c'est juste. Allez, va-t-en me mettre treize bêtes dans chaque bergerie.

(Ici, normalement, Mathieu aurait dû s'inquiéter, puisque 7 fois 13, comme 7 fois 4 font également 28. Mais s'il fallait encore s'attacher à tant de menus détails, on n'avancerait jamais. On continua donc).

$$\begin{array}{r} 13 \\ \times\quad 7 \\ \hline 21 \\ 7 \\ \hline 28 \end{array}$$

$$\begin{array}{cc} & 4 \\ 1 & 1 \\ 7 & \end{array}$$

C'est un Toine effondré qui revint une heure plus tard.

– J'y arrive toujours pas. Y a sûrement quelque chose qui ne va pas dans les comptes.

– Y a surtout qu't'es pas bien malin, fils, dit le père Mathieu. La division, la multiplication, c'est trop fort pour toi. L'addition, ça doit aller mieux.

J'écris 13, sept fois de suite, et j'additionne : 3 et 3, 6 ; et 3, 9 ; et 3, 12 ; et 3, 15 ; et 3, 18 ; et 3, 21 ; et 1, 22 ; et 1, 23 ; 24 ; 25 ; 26 ; 27 ; 28.
Es-tu convaincu, cette fois ? Allez, va.
Et le Toine repartit encore une fois, loger les maudites bêtes.
Et en fin de soirée, il revint triomphant.
– Ça y est, père, tous les moutons sont rentrés !
– Comment que t'as fait ?
– Je les ai fait rentrer un par un en faisant le tour des bergeries. Et pour être tout à fait sûr, quand ils ont été placés, moi aussi, j'ai fait mes comptes : j'ai compté les pattes ; j'ai trouvé 16 pattes dans chaque bergerie.

$$\begin{array}{r} 13 \\ 13 \\ 13 \\ 13 \\ 13 \\ 13 \\ 13 \\ \hline 28 \end{array}$$

– Attends voir, dit le Père Mathieu. Faut pas s'emballer. Étant donné qu'un mouton a 4 pattes, si je divise 16 par 4, je saurai combien tu as mis de bêtes dans chacune.

Et la nouvelle division fut posée : seize divisé par quatre : en 6 combien de fois 4 ? Il y va une fois. Une fois 4 fait 4 ; ôté de 6, il reste 2. J'abaisse mon 1. En 12 combien de fois 4 ? Il y va 3 fois. 3 fois 4 font 12 ; ôté de 12, il reste 0.

16	4
12	13
0	

1. A. Thuizat, in Le petit Archimède N^o 3, Association pour le développement de la culture scientifique.

Les variables

Les scalaires

Il ne faut pas confondre l'*identificateur* d'une variable et la *valeur* de la variable. La valeur d'une variable est conservée dans une zone mémoire.

Si `SOMME` vaut `10`, la notation algorithmique

```
SOMME = SOMME + 5
```

affecte [1] à la zone `SOMME` la *valeur précédente* de `SOMME` + 5 (soit `15`) .

Autre exemple :

```
SOMME = SOMME + SOMME
```

fait passer la zone valeur `SOMME` de `10` à `20`.

Les termes de *left value* et *right value* désignent respectivement la zone mémoire et la valeur qu'elle recèle.

On considère des objets informatiques et non plus mathématiques. Un objet informatique est caractérisé par le fait qu'il a non seulement une valeur mais une identité (ou une place dans la mémoire si l'on préfère). Ainsi, en mathématique il n'existe qu'un objet 392 alors que dans un système informatique on pourra avoir deux objets différents qui ont cette valeur (deux places dans la mémoire qui contiennent cette valeur). Il s'agira donc de bien faire la distinction entre identité (le même objet) et égalité (la même valeur).

Un ordinateur manipule des *représentations* de valeurs, qui sont des configurations de bits, d'octets ou de mots de la mémoire. Comme les représentations physiques varient selon les objets, on est conduit à spécifier leurs types.

Les entiers

sont en nombre fini dans l'ordinateur, donc certaines opérations peuvent créer un débordement, *i.e.* fournir un résultat hors des limites de l'intervalle. Le langage C ne teste jamais le résultat.

Les réels

L'ensemble est discontinu, la norme IEEE 754 décrit les nombres à virgule flottante.

La longueur effective (en *octets*), et donc les bornes extrêmes, dépendent évidemment de la machine. L'information est souvent disponible.

```
#include <stdio.h>
#include <machine/limits.h>
#include <float.h>

main()
{
    printf ("Max int : %12d\n",
                        INT_MAX) ;
    printf ("Min float : %g\n",
                        FLT_MIN) ;
    exit (0) ;
}
```

Les caractères

ne sont pas les mêmes selon le pays, l'ensemble des caractères est un sous-ensemble des entiers [2]. Pour Unix un ensemble de variables définit la langue dans laquelle s'affichent les messages et l'ordre du tri alphabétique.

Tableaux

Il est fréquent de manipuler des données de même type formant une collection que l'on nomme *tableau*. Chaque valeur est désignée par le *nom* du tableau et d'un ou plusieurs *indice(s)* ; ce nom peut être manipulé comme celui d'une variable. Un tableau à une seule dimension est souvent appelé *vecteur*. On peut utiliser des tableaux à n dimensions.

Les variables composites

Il est parfois nécessaire de manipuler des ensembles de données formant un tout ; ces ensembles sont nommés agrégats, enregistrements ou structures [3].

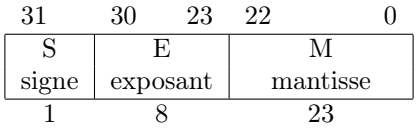
Les pointeurs

Un pointeur est une variable dont la valeur donne l'accès à une *autre* variable (elle contient en quelque sorte l'adresse de celle-ci) [4].

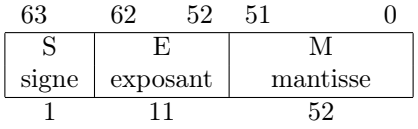
1. Page 9 la propriété de l'affectation
2. Page 84 une table de caractères
3. B. Mammeri, *Programmation*, p. 112.
4. B. Mammeri *Programmation*, p. 104.

La norme IEEE 754

La norme IEEE [5] *Standard for binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754 - 1985) a été définie dans le but d'améliorer la qualité du calcul flottant et la portabilité des applications. Ce standard est maintenant utilisé et respecté par tous les acteurs principaux du calcul scientifique. Deux formats principaux 32 et 64 bits (voir figure) et quatre modes d'arrondis (vers ∞, vers $-\infty$, vers 0, au plus près) sont définis, ainsi que des formats dits *étendus*.



Format flottant simple précision



Format flottant double précision

Le nombre représenté par le flottant (S,E,M) est $(-1)^s \times (1+M) \times 2^{(\text{ exposant - biais})}$ où : *biais = 127* pour les flottants simple précision et *biais = 1023* pour les flottants double précision [6] ; la mantisse (M) est codée sur 23 bits pour les flottants simple précision et sur 52 bits pour les flottants double précision.

Les microprocesseurs MIPS R10000 et UltraSPARC supportent les formats flottants IEEE 32 et 64 bits. De plus, le jeu d'instructions SPARC V9 intègre les flottants 128 bits ; cependant ces opérations ne sont pas supportées par matériel sur l'UltraSPARC, mais émulées. L'unité flottante du PentiumPro comme les autres microprocesseurs xxx86 manipule seulement des flottants 80 bits dans un format dit <<étendu>> : 64 bits de mantisse, 15 bits d'exposant et 1 bit de signe.

Source : *flloansi@IRISA.irisa.fr Tue Jun 4 09 :57 MET DST 1996*

La Norme 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 calculs de racines de fonctions éventuellement discontinues.

La norme IEEE 754 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

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 [7] 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 (*overflow*). Ces derniers sont en général plus souvent testés que les dépassements vers le bas.

5. The *Institut of electrical and Electronics Engineers*.

6. 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 à 0, et on additionne la partie exposant du mot machine flottant de e_{min}, c'est-à-dire de 127 en simple précision, ou de 1023 en double précision.

7. Page 10

L'affectation

Propriété de l'affectation

(P) /* $x > n$ */
$x = x + 1$;
(Q) /* $x > n + 1$ */

Dans l'exemple ci-contre, partant de la situation (P) $x > n$, on exécute l'instruction $x = x + 1$; pour établir la relation (Q) $x > n + 1$, Ce type d'affectation est si fréquemment utilisé qu'il porte un nom : **incrémentation**, et une écriture abrégée en langage C : `x++`.

Lorsque l'on fait une affectation [1], on poursuit un objectif : se rapprocher d'un résultat escompté. la nouvelle valeur de la variable modifie l'état du programme, c'est à dire les relations entre les variables.

1. Exemple présenté page 7 : SOMME = SOMME + 5.
2. Page 5 la définition.
3. Page 9

(P) /* $0 < x < 1$ */
$x = \frac{1}{x} + y$;
(Q) /* $x > y + 1$*/

Autre exemple :
Pour avoir la relation (Q) $x > y+1$, lorsque l'on a la relation (P) $0 < x < 1$, il faut exécuter
$x = \frac{1}{x} + y$.
$\Rightarrow \frac{1}{x_0} + y > y + 1$
$0 < x_0 < 1 \Rightarrow \frac{1}{x_0} > 1 \Rightarrow \frac{1}{x_0} + y > y + 1$

En supposant que, dans la relation (P) $0 < x < 1$, il suffit d'exécuter $x = \frac{1}{x} + y$ pour avoir la relation (Q), x vaut $\frac{1}{x_0} + y$

Déroulement linéaire

La forme la plus simple de l'algorithme [2] est le déroulement linéaire (enchaînement).

- Le déroulement linéaire ne comporte aucune prise de décision. Les lignes de programme qui s'y trouvent seront toujours exécutées dans le même ordre.
- Le déroulement linéaire est le mode implicite d'exécution d'un programme. Sans instruction qui suspend ou modifie le déroulement linéaire, l'ordinateur *passe* toujours à l'instruction suivante après l'exécution de l'instruction courante.
- La caractéristique d'un algorithme linéaire est de n'utiliser que l'enchaînement séquentiel d'actions donc la plus simple est l'affectation [3].

Propriété de l'enchaînement

Une succession d'instructions poursuit un objectif : se rapprocher d'un résultat prévu ; les relations entre les variables sont modifiées. Bien entendu, l'état final est le résultat de la succession des états intermédiaires.

Exemple :

Supposons les réels x et y, avec
(P) /* $0 < x < 1$ */
Pour obtenir
(Q) /* $x > y + 2$ */

on exécute
$x = \frac{1}{x} + y$;
$x = x + 1$;

En effet, nous avons
/* $0 < x < 1$ */
$x = \frac{1}{x} + y$;
/* $x > y + 1$ */

et que
/* $x > n$ */
$x = x + 1$;
/* $x > n + 1$ */

L'affectation - Un exemple (dé)trompeur.

Considérons l'algorithme suivant :

`Saisir`(x) ; `Saisir`(y) ;
$x = x + y$;
$y = x - y$;
$x = x - y$;
`Afficher`(x) ; `Afficher`(y) ;

Par nature, une instruction d'**affectation** réalise une transformation : elle change l'**état** d'une variable. Pour expliquer l'effet de la séquence

$x = x + y$; $y = x - y$; $x = x - y$;

il faut décrire la suite engendrée par les 3 instructions. Nous noterons /* ... */ une description d'état, c-à-d une relation entre les variables du programme et des constantes. Soient donc a et b les valeurs initiales des variables x et y

/* $x == a$ et $y == b$ */ $x = x + y$;

Si l'on exécute l'instruction $x = x + y$ seul x est modifié

/* $x == a$ et $y == b$ */ $x = x + y$; /* $x == a + b$ et $y == b$ */

L'instruction suivante modifie y

/* $x == a + b$ et $y == b$ */ $y = x - y$; /* $x == a + b$ et $y == (a + b) - b == a$ */

La dernière instruction modifie x

/* $x == a + b$ et $y == a$ */ $x = x - y$; /* $x == (a + b) - a == b$ et $y == a$ */

Ainsi donc les valeurs finales de x et y sont la permutation des valeurs initiales.

On appelle "assertion" l'affirmation d'une relation vraie entre les variables du programme en un point donné. Dire comment une instruction modifie l'assertion qui la précède (pré-assertion) pour donner celle qui la suit (post-assertion), c'est définir la *sémantique* de cette instruction.

Supposons que les nombres a et b soient des réels [1] et que b soit très petit devant a. les calculs étant faits avec un nombre constant de chiffres significatifs, à la précision des calculs b est négligeable devant a et l'addition de b à a ne modifie pas a

1. Page 7, l'ensemble des réels est discontinu

$/* x == a$ et $y == b */$ $\quad x = x + y\,;$
$/* x == a$ et $y == b */$ $\quad y = x - y\,;$

De même, retrancher b de a ne change pas a

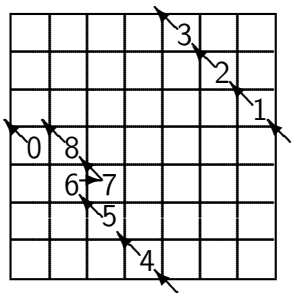
$/* x == a$ et $y == a */$ $\quad x = x - y\,;$
$/* x == 0$ et $y == a */$

L'échange des valeurs ne s'est pas fait. Il y a 0 en x et a en y. Ainsi le mécanisme des assertions peut être un mécanisme très fin décrivant même la façon dont les calculs sont exécutés dans l'ordinateur. Il permet une interprétation très précise de l'effet d'une séquence d'instructions. C'est lui qui nous permettra de donner un sens à un programme.

Carrés magiques [2]

Remplir un carré magique : c'est disposer les nombres 0 à p dans un carré de $n \times n$ cases, de telle sorte que la somme des nombres dans chaque ligne, chaque colonne et sur les deux diagonales soit la même. Dès que le carré est supérieur à 5×5 cases, il devient nécessaire de disposer d'une méthode. Nous allons considérer un exemple.

Remplissons une grille de 7×7.



Nous plaçons tout d'abord 0 dans la case du milieu du bord gauche. En suivant la petite flèche nous sortirions du carré. Il faut donc placer 1 dans la case correspondante sur le bord opposé, et ainsi de suite jusqu'à 6. Il n'est plus possible de continuer puisque la case suivante est occupée. Il suffit de placer 7 à droite de 6 puis de reprendre la progression.

Essayez de terminer seul ; vous pourrez vérifier que vous avez réalisé un *carré magique* puisque la somme des nombres placés dans chaque ligne, chaque colonne et chaque diagonale est constante (168).

L'inconvénient majeur de cette solution c'est l'espace mémoire occupé puisqu'il faut se doter d'un tableau de $n \times n$ cases.

Un autre algorithme consiste à produire la valeur de chaque case dans l'ordre de lecture, de gauche à droite et de bas en haut.

Supposons, pour simplifier, un *carré magique* de 5×5 déjà réalisé et étudions sa composition :

14	15	21	2	8
7	13	19	20	1
0	6	12	18	24
23	4	5	11	17
16	22	3	9	10

$2n+4$	$3n+0$	$4n+1$	$0n+2$	$n+3$
$n+2$	$2n+3$	$3n+4$	$4n+0$	$0n+1$
$0n+0$	$n+1$	$2n+2$	$3n+3$	$4n+4$
$4n+3$	$0n+4$	$n+0$	$2n+1$	$3n+2$
$3n+1$	$4n+2$	$0n+3$	$n+4$	$2n+0$

En exprimant le contenu de chaque case par rapport au côté du carré n la progression d'une case à la suivante est simple : les coefficients progressent régulièrement de 0 à $n-1$.

$X_{(ij)+1} = (a_{ij} + 1 \times n) + b_{ij} + 1$ ou a et $b \in \{0, 1, 2 \ldots n-1\}$

Lorsqu'une ligne est remplie, on passe à la première case de la ligne suivante en retranchant 1 au contenu de la dernière case de la ligne qui vient de s'achever. Enfin le premier terme du *carré magique* est obtenu par : $x_0 = \frac{n-1}{2} \times n + n - 1$

D'où l'algorithme :

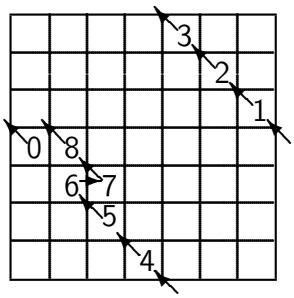
2. *in PPC* Dhénin Jean-Jacques

```
cote = SAISIR () ;
b = cote - 1 ;
a = (cote - 1) / 2 ;

for (ligne = 0 ; ligne < cote ; ligne++)
  {
  for (col = 0 ; col < cote ; col++)
    {
    IMPRIMER (a * cote) + b ;
```

.../...

```
    if (col != (cote - 1))
      {
      a++ ; a = a modulo cote ;
      b++ ; b = b modulo cote ;
      }
    }
  b-- ;
}
```


3
2
1
0
8
6
7
5
4

Deuxième partie

Les types de données abstraits

Les types de données abstraits

Dans cette deuxième partie, nous allons étudier quelques structures de données utilisées de façon intensive en informatique. Il s'agit de gérer des ensembles de données dont le nombre n'est pas fixé *a priori*. On ne s'intéresse pas aux éléments de l'ensemble considéré mais plutôt aux *méthodes* de gestion de ces éléments.

L'*analyse du domaine* sert à décomposer les objets réels complexes et leurs interrelations en objets et relations plus simples. Par exemple l'objet « plan d'une maison » se décompose en plans des différents étages et des façades, qui eux-mêmes se décomposent en murs, portes, fenêtres, etc.

L'*abstraction des données* va dans l'autre sens. À partir des objets informatiques de base (types prédéfinis, tableaux, enregistrements, pointeurs) on construit des objets plus abstraits (listes, ensembles, ...) qui servent eux-mêmes à construire des objets encore plus abstraits (graphes, dessins, ...). Jusqu'à ce qu'on arrive au niveau où il devient simple de représenter les objets du réel que l'on considère.

Définition de types de données
La définition d'un nouveau type de donnée, c.-à-d. d'une nouvelle abstraction, consiste à définir l'ensemble des valeurs possibles pour les objets de ce type et l'ensemble des opérations de traitement de ces valeurs.

La gestion des ensembles doit, pour être efficace, respecter au mieux deux critères parfois contradictoires : un minimum de place mémoire utilisée [1] et un minimum d'instructions pour réaliser une opération. Pour bien faire, la place mémoire utilisée devrait être voisine du nombre d'éléments de l'ensemble multiplié par la taille d'un élément.

Pour accueillir les valeurs du type il est nécessaire de définir une structure interne de données construite à partir des types de base du langage et/ou des types déjà définis. Les opérations seront définies par des procédures ou fonctions du langage de programmation.

1. Cf. page **??**

La qualité du logiciel

La qualité du logiciel est l'objectif du génie logiciel. Ce n'est pas une idée simple mais un ensemble de *facteurs*. Les recherches de ces dernières années concernent la rapidité, la fiabilité, la lisibilité et la maintenabilité. Nous avons, dans la première partie, abordée la notion de programmation structurée. Au cours des travaux pratiques, nous avons utilisé la programmation modulaire.

Du point de vue *externe* de l'utilisateur, la qualité d'un logiciel se manifeste par son ergonomie, son efficacité, sa facilité d'emploi.

Du point de vue *interne* du concepteur, la qualité se fonde sur la facilité de mise au point, la lisibilité du source et la réutilisabilité.

La maintenance du logiciel est chose coûteuse...en efforts. (cf. Le bug de l'an 2000). Il convient donc de s'orienter vers une simplification et une clarification du travail par :

La modularité. On décompose le programme en autant de fichiers que nécessaire afin de réduire les difficultés. Cette décomposition logique et sensée vise à favoriser la réutilisabilité des modules. En particulier, il est adroit de décomposer le problème afin qu'un changement de spécification n'induise la réécriture que d'un seul module.

D'autre part, les données d'un module lui sont propres [1] et protégées. L'accès aux données se fait par *interfaces* c'est à dire des méthodes (*i.e.* des fonctions) appartenant aux modules, accessibles depuis d'autres modules.

La réutilisabilité. Il est agréable de pouvoir utiliser le même module quelque soit le type de données traitées. Pourquoi réécrire un calcul de moyenne sur des entiers si l'on dispose déjà du même calcul sur des réels ?

De même, les travaux récents sur les interfaces homme/machine (IHM), révèle l'importance de disposer de modules indépendants de la présentation et pouvant être utilisés avec différentes présentations (HTML, FVWM ou TK).

La certification. Il n'existe pas (encore) de solution pour une réalisation parfaite des logiciels. Il est cependant possible d'inclure des éléments de spécification parmi les lignes de code. C'est notamment le cas en langage C avec la macro `assert`. Nous l'avons abondamment utilisé dans la première partie. En voici un autre exemple :

```
#include <assert.h>
main()
{
     int a, b, u, x , y ;

     scanf ("%d%d", &x, &y) ;

     a = x ; b = y ;

     u = x + y ;
     u = u + u ;
     x = x + y ;
     u = u - x ;

     assert ( u == (a + b)) ;
     assert ( x == (a + b)) ;

     printf ("u :%d, x = %d\n", u, x) ;
}
```

On utilisera cette technique afin de garantir que les conditions d'utilisation des modules sont respectées. Par exemple, la méthode de retrait d'un élément d'une pile s'assurera que la pile n'est pas vide.

Les assertions contribuent à réaliser du logiciel correct, aident à la documentation, facilitent le déboguage et la tolérance de pannes.

1. C'est l'encapsulation. page suivante

L'encapsulation

La modification imperceptible des variables globales est une des principales difficultés rencontrées dans la maintenance des programmes. C'est pourquoi la première recommandation est de **ne pas utiliser de variable globale**.

Dans la construction des types de données abstraits, la systématisation de cette idée conduit à n'admettre l'accès aux données qu'au moyen de fonctions associées au types de données.

Règles des espaces de visibilité :

- Un programme C est composé de nombreux éléments qui sont des variables ou des fonctions. Afin d'éviter les conflits de noms, le langage permet de restreindre la visibilité de certains objets qui n'ont aucune raison d'être globaux (comme les indices de boucles).
- L'espace de visibilité d'une variable ou d'une fonction correspond à la partie de source pour laquelle elle est définie (connue).
- De manière générale une variable est définie depuis sa déclaration jusqu'à l'accolade fermente qui la suit ou jusqu'à la fin du module si elle n'est pas à l'intérieur d'une paire d'accolades.
- Une variable définie dans une fonction n'est donc visible que dans cette fonction. Une variable déclarée hors de toute fonction est visible jusqu'à la fin de son module. Elle peut, dans ce cas, être rendue visible depuis les autres modules de l'application (c'est à dire exportée) ou rester privée au module qui la déclare.
- Une fonction est visible dans l'ensemble du fichier qui la contient.
- Elle peut être rendue visible des autres modules ou conservée privée. Exemple :

```
/* Hors du module,
 * importee
 */
extern int V1 ;

/* Globale au module,
 * exportee vers les autres
 */
int V2 ;

/* Globale au module,
 * non visible depuis les autres
 */
static int V3 ;

/* V1, V2, V3, F1 et F2
 * sont visibles
 */

/* Fonction non exportee */
static int F1 ()
{
    int V4 ; /* Locale a F1 */
    int V2 ;
    /* La variable V2 globale
     * est cachee par celle-ci
     * V1, V2, V3, V4, F1 et F2
     * sont visibles
     * La version globale de V2
     * est inaccessible
     */
}

/* V1, V2, V3, F1 et F2
 * sont visibles
 */
int V5 ;

/* V1, V2, V3, V5, F1 et F2
 * sont visibles
 */

int F2 () /* Fonction exportee */
{
    /* Variable locale permanente */
     static V6 ;
    /* V1, V2, V3, V5, V6, F1 et F2
     * sont visibles
     */
}

/* V1, V2, V3, V5, F1 et F2
 * sont visibles
 */

/* V2, V5 et F2
 * sont visibles depuis
 * d'autres modules
 */
```

- Si l'on veut qu'une variable déclarée dans un autre fichier soit visible dans le module courant, elle doit être déclarée externe dans celui-ci. *A contrario*, une fonction appelée dans un module est supposée externe.
  Si le compilateur la rencontre dans ce même module, il supprime son importation. En d'autres termes, on peut appeler toutes les fonctions de tous les modules sans avoir à annoncer explicitement qu'elles sont externes.

La dernière étape de construction de l'application consiste à collecter l'ensemble de ses constituants pour créer l'exécutable. C'est l'édition des liens. L'application ne peut être correctement construite que si chacun des objets importés par chacun des modules est publié par exactement

un autre module. C'est à dire que tous les objets que les modules importent, d'autres les exportent.

L'instanciation

La programmation simpliste d'un type de données abstrait, consiste à réécrire chaque fois que nécessaire l'ensemble des fonctions applicables aux données manipulées. Ainsi on écrit une pile de nombres pour une calculatrice et une pile de blocs libres pour la gestion d'un disque et une pile pour la saisie des caractères. De même on écrit un programme de gestion d'un arbre pour les répertoires et un autre pour l'analyse syntaxique d'une expression.

Une autre approche réalise une fois pour toutes un module qui ne représente aucun objet concret, mais seulement un *modèle* de la structure de données, c'est à dire son fonctionnement. L'ensemble des fonctions est prévu pour s'appliquer aussi bien à des entiers, à des flottants, des tableaux, des structures ou autre.

Le programmeur-utilisateur de ce module doit alors explicitement créer un *objet* à l'aide d'une opération appelée *constructeur* définie dans le module. On dit que l'on crée une *instance* du type abstrait. La fonction *constructeur* associe les données du programme effectivement traitées aux fonctions préalablement définies.

Dans l'exemple ci-dessous[2], on commence par indiquer que la liste s'applique à des entiers (`liste(entier)`), puis dans le `main`, on créé une variable `l` du type liste d'entier (`entierliste l`) et enfin on instancie cette liste d'entiers au moyen du constructeur (`l = entierliste_creer (entier_copier, 0, entier_editer) ;`).

Après cette phase d'initialisation, on peut utiliser toutes les fonctions du type de données choisi en répondant à la question *que faire ?* et non plus *comment faire ?* .

```
/* Ecp - 1ere Annee -
 * jjd                          Liste1.c
 */
#include "Liste.tda"
typedef int  *entier ;     /* Creation du
nouveau type pointeur sur int */
liste(entier) ;

/* fonctions particulieres au nouveau type
*/

void entier_editer(entier i)
{ printf("%d", *i) ; }

entier entier_copier(entier i)
{
    entier      j ;
    j = (entier) malloc(sizeof(*i)) ;
    *j = *i ;
    return j ;
}

void main()
{
    entierliste   l ;
    entier        x ;
    int           i ;

/* Instanciation */
    l = entierliste_creer(entier_copier, 0,
entier_editer) ;

    for (i = 0 ; i < 5 ; i++)
    {
        x = entier_copier((entier) & i)
;
        entierliste_insererapres(l, (entier)
x) ;
    }
    entierliste_afficher(l) ;
}
```

2. Le programme complet est en annexe

Les pointeurs de fonctions

Quelle importance ?

L'écriture de type de données abstrait présuppose une bonne connaissance des pointeurs tant pour le maniement des structures [3] que pour la mise en place des méthodes.

Rappel

— En C, une fonction elle-même n'est pas une variable mais il est possible de définir des pointeurs de fonctions que l'on peut affecter, placer dans des tableaux (de pointeurs de fonctions), passer en argument à des fonctions ou faire retourner par des fonctions...

Déclaration

— La déclaration d'un pointeur de fonction se fait de la façon suivante :
`int (*ptf) (int,char *) ;`
Déclaration d'un pointeur de fonction pour une fonction qui retourne un entier et a deux arguments, un entier et un pointeur de type char.

— Si on écrit : `int *ptf (int,char *);` on écrit un prototype... Les parenthèses autour de l'identificateur sont là pour préciser qu'il s'agit d'un pointeur et non d'une fonction bien précise.

Initialisation et utilisation

— Un pointeur déclaré peut être initialisé comme toute variable, mais ici avec un nom de fonction.
Exemple : `int LireLigne (char *) ;`

```
main ()
{
  char    tab[256]         ;
  int     n                ;
  int     (* ptf) (char *) ;

  ptf = LireLigne ;            /* Initialisation.    */

  n = (*ptf) (tab) ;           /* Appel de LireLigne.    */
}
```

— Dans le module `Liste.c` [4], on initialise les pointeurs de fonction de la structure `fonctions` définie dans le fichier `Liste.h` :

```
void * lier_liste ()
{
    Liste.creer        = &creer        ;
    Liste.copier       = 0             ;
    Liste.detruire     = &detruire     ;
    ...
    Liste.retablir     = &retablir     ;
    Liste.afficher     = &afficher     ;
}
```

3. Voir p. II.
4. c.f. Annexe

Les macros (chaînes de remplacement)

La première règle est : *ne les utilisez pas* si vous n'avez pas à le faire[5]. Il a pu être observé qu'à peu près chaque macro démontre un défaut dans la programmation. Puisqu'elles réorganisent le texte d'un programme avant que le compilateur ne les voie, les macros sont un problème majeur pour la plupart des outils de développement (débogueurs, profileurs...).

Rappel

Une directive (macro-instruction[6]) de prétraitement de la forme :

`#define` *identificateur chaine-symbole*

provoque le remplacement par le précompilateur de toutes les instances suivantes de l'identificateur avec la séquence de symboles donnée. Les espaces entourant la séquence de symboles de remplacement sont annulés. Par exemple :

```
#define COTE 8
char jeu[COTE][COTE] ;
```

devient après le passage du précompilateur :

```
char jeu [8] [8] ;
```

Opérateur

Si une occurrence d'un paramètre dans une séquence de symboles de remplacement est immédiatement précédée par un symbole #, le paramètre et l'opérateur # seront remplacés dans le développement par un littéral chaîne contenant l'orthographe de l'argument correspondant. Un caractère \ est inséré dans un littéral chaîne avant chaque occurrence d'un \ ou d'un " à l'intérieur d'une (ou délimitant) une constante caractère ou un littéral chaîne dans l'argument.

```
#define path(logid,cmd) "/users/" #logid
"/bin/" #cmd #define jjd dhenin
```

à l'appel

```
char * outil = path(jjd, listlp) ;
```

sera interprété :

```
char * outil = "/users/" "jjd" "/bin/"
"listlp" ;
```

qui sera ensuite concaténé pour devenir :

```
char * outil = "/users/jjd/bin/listlp" ;
```

Opérateur

Si un opérateur `##` apparaît entre deux symboles dans une séquence de symboles de remplacement, et si l'un ou l'autre des symboles est un paramètre, il est tout d'abord remplacé, puis l'opérateur `##` et tous les espaces l'entourant sont ensuite supprimés. L'effet de l'opérateur `##` est donc une concaténation.

Par exemple dans la création du TDA liste (`Liste.h` et `Liste.tda`)

```
#define nom(a,b) a##b
#define liste(type_objet)            \
void nom (type_objet, liste_premier) \
    (nom(type_objet, liste) l)       \
{                                    \
    (*Liste.premier) (l->rep) ;      \
}

liste(Entier) ;
```

produira :

```
   void Entierliste_premier(Entierliste l)
{
    (*Liste.premier) (l->rep) ;
}
```

Mais toute macro utilisée comme un des symboles adjacents à `##` n'est pas expansée, à l'inverse du résultat de la concaténation.

```
#define concat(a)       a##valise
#define mot             B
#define motvalise       rafiscotche

contate(mot)
```

donne

```
rafiscotche
```

et non pas

```
Bvalise

main ()
{
     int a = 1 ;
     int B = 2 ;
     int Bvalise = 3 ;
     int rafiscotche = 4 ;


     printf ("%d\n", concat(mot)) ;
}
```

5. On lira avec intérêt le fichier `/usr/include/ctype.h`
6. Mammeri M. *programmation* ÉCOLE CENTRALE DE PARIS 1997-1998 p. 112

Les définitions de types

Des déclarations contenant le *spécificateur-déclaration* `typedef` nomment des identificateurs pouvant être utilisés ensuite pour la désignation de types fondamentaux ou dérivés. Le spécificateur `typedef` ne peut pas être utilisé dans une *définition-fonction*.

La définition des prototypes, dans le manuel, conjointement avec les fichiers d'entête utilise abondamment le `typedef`. Par exemple

```
$ man malloc
...
SYNOPSIS
     #include <stdlib.h>

     void *
     malloc(size_t size)
...

$ more /usr/include/stdlib.h
...
#include <machine/ansi.h>
...
typedef _BSD_SIZE_T_    size_t;
...

$ more /usr/include/machine/ansi.h
...
#define _BSD_SIZE_T_  unsigned int
...
```

ce qui permet de traduire :

`size_t` est l'équivalent de `unsigned int` .

Utilisation pour les types de données abstraits

L'usage élémentaire consiste à définir un type `Boolean` [7].

```
typedef enum { FALSE, TRUE } Boolean ;
```

L'usage plus élégant consiste à renommer les structures [8] :

```
struct MAILLON
{
    TypDon           elem ;
    struct MAILLON * suiv ;
} ;

typedef struct MAILLON * File ;
```

On préférera l'écriture en deux étapes à celle plus propice à créer la confusion :

```
typedef struct MAILLON  /* A EVITER */
{
    TypDon           elem ;
    struct MAILLON * suiv ;
} * File ;
```

Le fichier `Liste.tda` est un gigantesque `typedef` qui utilise les possibilités des macros afin de créer des listes d'objets dont le type est défini au moment de l'utilisation.

```
#define nom(a,b) a##b

#define liste(type_objet)                                   \
typedef struct nom(type_objet, liste)                       \
{                                                           \
    void *     rep ;                                        \
    type_objet (*copier_objet)   (type_objet) ;             \
    void       (*detruire_objet) (type_objet) ;             \
    void       (*afficher_objet) (type_objet) ;             \
} * nom(type_objet, liste) ;                                \
nom(type_objet, liste) nom(type_objet, liste_creer)         \
    (type_objet (*copier)   (type_objet),                   \
     void       (*detruire) (type_objet),                   \
     void       (*editer)   (type_objet))                   \
{                                                           \
    nom(type_objet, liste) l ;                              \
    lier_liste () ;                                         \
    l = (nom(type_objet, liste)) malloc                     \
     (sizeof (struct nom (type_objet, liste))) ;            \
    l->rep = (*Liste.creer) () ;                            \
    l->copier_objet = copier ;                              \
    l->detruire_objet = detruire ;                          \
    l->afficher_objet = editer ;                            \
    return l ;                                              \
}                                                           \
...
void nom(type_objet, liste_afficher)                        \
    (nom(type_objet, liste) l)                              \
{                                                           \
    if (l->afficher_objet)                                  \
    (*Liste.afficher) (l->rep, l->afficher_objet) ;         \
    else                                                    \
    printf ("ERR : fonction d'affichage  nulle\n") ;        \
}
```

Remarquez qu'il s'agit d'une seule longue ligne logique, les \ réalisant la concaténation des lignes physiques.

7. Mammeri M. *programmation* ECOLE CENTRALE DE PARIS 1997-1998 p. 19 et 36
8. *ibid.* p.65

Correspondance par tables

Définition

Une table de correspondance ou plus simplement correspondance, est une fonction d'un ensemble d'éléments de type TypeElément vers un autre ensemble d'éléments de type différent ou éventuellement de même type. Nous exprimerons le fait qu'une correspondance C associe à l'élément s de type TypeSource l'élément b de type TypeBut par la relation $C(s) = b$.

Certaines correspondances, comme carré$(i)= i^2$, s'expriment de manière triviale à l'aide de fonctions de programmation standard en donnant l'expression arithmétique correspondante ou la méthode de calcul de $C(s)$ en fonction de s. Malgré tout, dans de nombreux cas, il n'existe pas d'autre moyen de caractériser C qu'en stockant la valeur de $C(s)$ pour chaque s de l'ensemble source.

Examinons quelles opérations risquent d'être effectuées sur une correspondance C. Étant donné un élément s d'un domaine particulier, on souhaitera vraisemblablement connaître la valeur de $C(s)$ ou savoir si $C(s)$ est défini (c'est-à-dire savoir si s appartient bien au domaine de définition de C). On peut aussi désirer introduire de nouveaux éléments dans le domaine de définition de C et connaître les valeurs du domaine d'arrivée qui leur sont associées.

Inversement, il sera peut-être aussi demandé de pouvoir changer la valeur des $C(s)$. Enfin, il devra être possible de remettre la correspondance *à zéro*, c'est-à-dire transformer son domaine de départ en ensemble vide. Ces opérations sont représentées par les trois commandes suivantes :

raz(C) Remise à zéro de la correspondance C.

assigner(C, s, b) Donner la valeur b à $C(s)$, que $C(s)$ ait été défini ou non auparavant.

calculer(C, s, b) Retourner `vrai` et placer la valeur de $C(s)$ dans la variable b si $C(s)$ est défini ; retourner `faux` sinon.

Mise en œuvre des correspondances par tableau

Assez souvent, le type des éléments du domaine de départ d'une correspondance est un type élémentaire que l'on peut utiliser comme domaine indiciel d'un tableau. En Pascal, les types indiciels comprennent tous les intervalles finis d'entiers, comme 1..100 ou 17..23, le type caractère (char) et les intervalles de caractères comme 'A'..'Z' et enfin les types énumérés comme nord, est, sud, ouest. Par exemple, un programme de décryptage contiendra peut-être une table de correspondance codage avec 'A'..'Z' comme domaine source et domaine but, telle que codage(LettreNormale) soit le code dont le programme a reconnu qu'il équivalait au caractère LettreNormale.

De telles correspondances permettent facilement une mise en œuvre par tableau, dès lors que le TypeBut contient une valeur particulière pouvant signifier « non défini ». On parle alors naturellement de « table de correspondance » ou « d'adressage ».

Les tableaux associatifs de Perl

Dans le language Perl un tableau associatif est une liste constituée de paires *clé, valeur*.

Un tableau associatif est déclaré par le symbole « `%` ». Exemple :

```
%caracteristiques = (
'xenon',   'philosophe',
'claude',  'solide',
'serge',   'moine',
'gus',     'clown',
'pascal',  'penseur',
'henri',   'mesure',
'simon',   'tranche' ) ;
```

On utilise donc la même notation que pour les tableaux simples.

Pour accéder à un élément d'un tableau associatif, on utilise les accolades et la clé :

```
$caract = $caracteristiques{'xenon'}
; # $caract = philosophe
$home = $ENV{'HOME'} ;
$SIG'UP' = 'IGNORE' ;
```

En général, on ne fait pas référence à un tableau associatif dans son entier, mais à ses éléments. Chaque élément est accédé par sa clé ; ainsi, les éléments du tableau associatif `%tab` sont accédés par `$tab{$cle}`, où `$cle` est une expression scalaire.

```
$tab{"coucou"} = "ca va ?" ;
# création de la clé "coucou",
à laquelle est associée "ca va
?"
$tab{123.5} = 4568 ;
# création de la clé 123.5,
à laquelle est associée la
valeur 4568
print "$tab{'coucou'}" ;
```

```
# affichage de "ca va ?"
$tab{123.5} += 3 ;
# la valeur associée à la clé
123.5 est maintenant 4571
```

Les opérateurs sur les tableaux associatifs de Perl

— L'opérateur `keys()`
Cet opérateur renvoie la liste des clés du tableau associatif qu'on lui passe en paramètre. Les parenthèses sont optionnelles. Exemples :

```
foreach $key (keys %tab)
{ print "la valeur associée
à la clé $key est $tab{$key}."
; }
```

— L'opérateur `values()`
Cet opérateur renvoie la liste des valeurs du tableau associatif qu'on lui passe en paramètre.

```
print values(%tab) ;
# affiche "ca va ?", 4568 ou
4568, "ca va ?"
```

— L'opérateur `each()`
Pour examiner tous les éléments d'un tableau associatif, on peut donc faire appel à `keys()` puis récupérer les valeurs correspondant aux clés. En utilisant `each()`, on obtient directement une paire (clé-valeur) du tableau passé en paramètre.
À chaque évaluation de cet opérateur pour un même tableau, la paire suivante est renvoyée, jusqu'à ce qu'il n'y ait plus de paire à accéder ; `each()` retourne alors la chaîne vide. L'exemple précédent peut alors s'écrire :

```
while (($cle, $valeur) =
each(%tab))
{ print "la valeur associée
à la clé $cle est $valeur."
; }
```

— L'opérateur `delete`
Élimine la paire (clé-valeur) du tableau associatif dont on a passé la clé en paramètre ; exemple :

```
%tab = ("coucou ", "ca va
?", 123.5, 4571) ;
delete $tab{"coucou"} ;
#%tab contient maintenant
(123.5, 4571)
```

Les structures

Rappels

— Un tableau est un agrégat d'éléments de même type ; une `struct` est un agrégat [1] d'éléments de type (presque) arbitraire. Par exemple :

```
struct client
{
    char * nom       ;
    int    telephone ;
    float  solde     ;
} ;
```

— Les types de données abstraits font un grand usage de structures et notamment de la structure MAILLON [2].

— Différentes méthodes sont utilisées pour renseigner les éléments d'une structure :

```
/* Designee par un nom */
struct client   lambda  ;
/* Designee par une adresse */
struct client * plambda ;

/* Un nom, donc un point */
lambda.nom   = "Tournesol" ;
/* Une adresse, donc une fleche */
plambda->nom = "Hadock"    ;
```

Ou comme pour les tableaux, au moment de l'instanciation :

```
struct client lambda =
    {"Milou", 1244, 5.0} ;
```

— Les objets de type structure peuvent être affectés, passés en argument et renvoyés comme résultat d'une fonction. Par exemple :

```
typedef struct client Client ;
Client lambda ;

Client Precedent (Client suivant)
{
    Client anterieur = lambda  ;
   lambda    = suivant ;
   return anterieur ;
}
```

— La comparaison des structures n'est pas possible en langage C.

— La taille d'un objet de type structure n'est pas nécessairement la somme de la taille de chacun de ses membres. En effet, de nombreuses machines exigent que les objets de certains types soient alloués sur des frontières dépendantes de l'architecture. Cette contrainte implique l'obligation d'utiliser `sizeof` et de ne pas coder la taille de la structure *en dur*.

— Le nom d'un type devient utilisable immédiatement après avoir été rencontré et non pas juste après la fin de la déclaration. Par exemple :

```
struct MAILLON
{
    TypDon           donn ;
    struct MAILLON * suiv ;
} ;
```

Pour permettre à deux structures de se référencer mutuellement, il est possible de leur donner préalablement un nom. Par exemple :

```
struct un ; /* Definie ulterieurement */

struct deux
{
    struct un   * p ;
    struct deux * q ;
} ;

struct un
{
    struct deux * r ;
} ;
```

Sans la première déclaration de `struct un`, la déclaration de `deux` aurait provoqué une erreur de syntaxe.

— Deux types de structures sont différentes même si elles ont les mêmes membres. Par exemple :

```
struct s1 { int a ; } ;
struct s2 { int a ; } ;

struct s1 x ;
struct s2 y = x ; /* erreur */
```

1. Mammeri M. *programmation* ECOLE CENTRALE DE PARIS 1997-1998 p. 112
2. *ibid.* p. 88

Les listes

Généralités

Par extension du sens habituel donné au mot, une liste [1] linéaire [2] est un *type de données abstrait* comprenant une suite d'informations (données) ainsi que les opérations (méthodes) nécessaires au maniement de ces données. La liste linéaire est perçue comme une suite d'éléments (*maillons*) ordonnée, caractérisée par le chaînage [3] de chaque maillon à son suivant.

La liste est une structure de base de la programmation [4]. Chaque élément permet l'accès à deux valeurs : l'objet associé à cet élément et l'élément suivant [5]. La recherche d'un élément dans la liste s'apparente à un << 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 l'objet, en ce lieu on découvre des informations sur un autre lieu où il a une chance de se trouver et ainsi de suite.

Une liste doit pouvoir être modifiée : on doit être capable de supprimer ou d'ajouter des données.

Pour accéder à une donnée appartenant à une liste, il convient de disposer de la liste mais aussi de la place (position) de cette donnée dans la liste.

Une liste peut être vide ; on peut vider une liste ; on peut accéder au $i$ème élément, en connaître le contenu ; on peut connaître la longueur de la liste (le nombre d'éléments qu'elle contient) ; on peut supprimer un élément de la liste, y insérer un nouvel élément et enfin accéder au successeur d'un élément dont on connaît la place.

Enfin, on peut vouloir fabriquer une liste à partir de deux autres.

La pile (LIFO) est une liste dans laquelle on insère et on supprime seulement à une extrémité ;

La file (FIFO) est une liste dans laquelle on insère à une extrémité et où on supprime à l'autre.

Il existe de nombreuses façons de mettre en œuvre les listes. La mise en œuvre dans un tableau facilite la compréhension, mais présente l'inconvénient de ne contenir qu'un nombre maximum prédéfini de cellules.

Mise en œuvre des listes dans un tableau

La figure II montre un tableau `ESPACE` contenant deux listes $\mathcal{L} = a, b, c$ et $\mathcal{M} = D, E$. Pour chaîner les éléments des listes on utilise un vecteur "suivant" de faux-pointeurs, c'est-à-dire le numéro de ligne du successeur.

Remarquez que toutes les cellules du tableau n'appartenant à aucune des deux listes sont chaînées en une pile [6] appelée **disponible**. Cette liste supplémentaire sert à trouver un espace libre pour insérer un nouvel élément, ou à récupérer les espaces libérés par la suppression d'éléments précédemment dans une liste en vue d'une utilisation ultérieure.

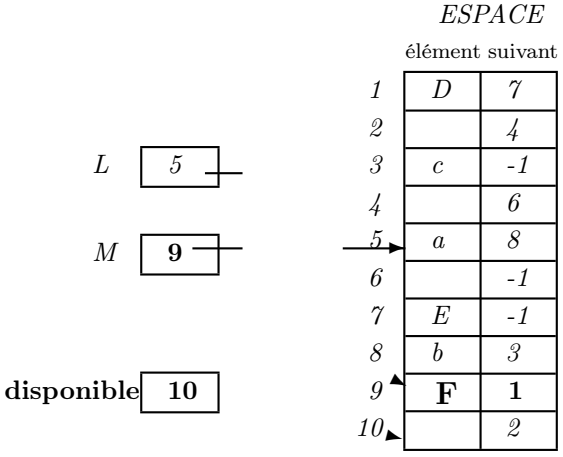


FIG. 24.1– *Une liste chaînée dans un tablea*

La variable L contient l'indice de ligne où commence la liste $\mathcal{L}$ et la variable M celui de la liste $\mathcal{M}$; la case ESPACE[L][element] contient le premier élément (a) et ESPACE[L][suivant] l'indice du successeur (8) c-à-d b. De la même façon, ESPACE[8][suivant] contient 3.

c n'ayant pas de successeur, il est le dernier élément, ESPACE[3][suivant] est -1.

1. Les listes peuvent être vues comme des formes simples d'arbres (cf. page 45) ; on peut en effet considérer qu'une liste est un arbre binaire dans lequel tout fils gauche est une feuille.
2. Knuth distingue les « listes linéaires » des « listes » au sens large qui désignent les arborescentes. Cependant l'usage ne maintient pas cette distinction.
3. Au moyen d'un pointeur par exemple. (cf. page 7)
4. Le langage sc lisp, conçu par John MacCarthy en 1960, utilise principalement cette structure qui se révèle utile pour le calcul symbolique.
5. Le numéro de ligne, lorsqu'il s'agit d'un tableau, comme dans la figure II, l'adresse en mémoire, lorsqu'il s'agit de *pointeurs*.
6. Cf. page 41.

Programmation d'une liste générique

La totalité des sources du TDA liste est donnée en annexe.

Le fichier d'*entête* du TDA liste (`Liste.h`), contient la définition de 3 structures :

- La structure *cellule*, qui contient un pointeur sur la donnée associée et un pointeur sur la cellule suivante,
- la structure de tête qui donne accès à la première cellule de la liste, à la dernière et à la cellule courante (*vue*),
- la structure donnant accès aux méthodes (fonctions [7]).

```
#define LISTE_H
#include <stdio.h>
#include <stdlib.h>

#define nom(a,b) a##b

typedef struct cell
{
    void        * objet ;
    struct cell * suiv  ;
} * cellule ;

typedef struct tete
{
    cellule prem, vue, der, s ;
} * liste ;

extern void * lier_liste () ;
struct fonctions
{
    liste  (*creer)      ()                          ;
    void   (*copier)     (void *)                    ;
    void   (*detruire)   (liste, void (*) (void*))   ;
    int    (*nulle)      (liste)                     ;
    void   (*premier)    (liste)                     ;
    void   (*dernier)    (liste)                     ;
    void   (*suivant)    (liste)                     ;
    int    (*fin)        (liste)                     ;
    void * (*lire)       (liste)                     ;
    void   (*insereravant) (liste, void *)           ;
    void   (*insererapres) (liste, void *)           ;
    void   (*remplacer)  (liste, void *, void (*) (void*)) ;
    void   (*oter)       (liste, void (*) ())        ;
    void   (*fixer)      (liste)                     ;
    int    (*retablir)   (liste)                     ;
    void   (*afficher)   (liste, void (*) ())        ;
} Liste ;
```

7. Voir les pointeurs de fonction p. II

Suppression/Insertion d'un maillon de liste

Suppression d'un élément

On a l'habitude de représenter une liste chaînée par une suite de boîtes (maillons) à 2 compartiments comme dans la figure II. Le compartiment de gauche contient une donnée (ou l'adresse d'une donnée) tandis que le compartiment de droite contient l'adresse du maillon *suivant*.

Une structure comprenant 3 adresses autorise l'accès immédiat au maillon de tête, au maillon courant et au dernier maillon.

FIG. 24.2.1– *Suppression de l'élément* `x`

La suppression du maillon qui contient x s'effectue en modifiant la valeur de *suivant* contenue dans le prédécesseur de x : la fonction *dispose* restitue à la liste *disponible* la place libérée, celle-ci pourra être réutilisée lors de la création d'un *nouveau* maillon.

```
static BOOL
LibereObject (List * l, void * ancien)
{
  Maillon * me = l->premier ;
  if (-1 != me)
```

```
  {
    Maillon * precedent ;
    while (-1 != me->suivant)
    {
      precedent = me   ;
      me = me->suivant ;
      /* La comparaison qui suit
       * doit etre adaptee */
      if (me->objet == ancien)
      {
        precedent->suivant = me->suivant ;
        dispose(me) ;
        return VRAI ; /* Succes */
      }
    }
  } /* objet non trouve ou liste vide */
  return FAUX ;
}
```

Insertion d'un élément

Pour insérer un élément x dans une liste L, on utilise la première cellule libre dans la liste disponible et on la place à la bonne position dans la liste L. L'élément x est ensuite placé dans le champ élément de cette cellule. Figure II.

Insérer un nouvel élément en tête de la liste $\mathcal{M}$ peut s'écrire de façon simplifiée :

```
inserer (M, e)
{
        temp = M ;
           M = premier_disponible() ;
  disponible = suivant(M) ;
  suivant(M) = temp ;
  element(M) = e ;
}
```

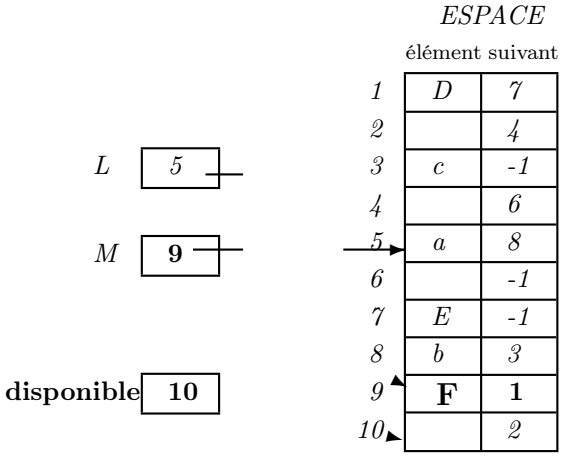


FIG. 24.2.2– *Insertion en tête de $\mathcal{M}$*

Affichage

Parcourir la liste pour en faire l'affichage peut se réaliser au moyen de la fonction `prochain(e)` :

Si l'on inverse les lignes commentées `(1)` et `(2)`, la liste est affichée en sens inverse.

```
lire(e)
{
    if (-1 == e) return ;
    afficher (e) ;       /* (1) */
    lire (suivant(e)) ;  /* (2) */
}
```

Utilisation d'une pile pour le traitement des expressions

Généralités

Intuitivement, une pile correspond à une pile de livres sur une table. La première opération consiste à poser un livre sur la table vide. L'opération suivante place un second livre sur le livre déjà posé sur la table. Chaque opération de rangement augmente la taille de la pile. Prendre un livre sans provoquer l'effondrement de la pile n'est possible qu'au sommet de la pile.

Traitement

En mode infixé, écrire une expression consiste à :

- écrire un opérateur entre 2 opérandes,
- écrire un opérateur unaire suivit de son opérande,
- modifier l'ordre des priorités à l'aide de parenthèses.

Le traitement informatique d'une expression infixée est malaisé. Il est préférable d'effectuer le traitement de l'expression équivalente en mode postfixé ou infixé. Par exemple, l'expression $(3+4)\times 2$ devient $3\ 4+2\times$ en notation postfixée.

Symboles	Pile	Actions
3	3	*push* 3
4	3, 4	*push* 4
+		*pop* 4 ; *pop* 3
		calculer $7 = 3 + 4$
	7	*push* 7
2	7, 2	*push* 2
×		*pop* 2 ; *pop* 7
		calculer $14 = 7 \times 2$
	14	*push* 14

Règles de transformation

1. Les variables successivement rencontrées dans l'expression infixée sont rangées directement dans l'expression préfixée.
2. Les opérateurs sont empilés en tenant compte de leur priorité :
   - priorité de l'opérateur ds l'expression > priorité de l'opérateur au sommet de la pile $\Longrightarrow$ empiler l'opérateur,
   - priorité de l'opérateur ds l'expression <= priorité de l'opérateur au sommet de la pile $\Longrightarrow$ dépiler et ranger l'opérateur dépilé dans l'expression postfixée.
3. Empiler systématiquement une parenthèse ouvrante car elle délimite une sous-expression,
4. la parenthèse fermante fait sortir tous les éléments de la pile jusqu'à la rencontre d'une parenthèse ouvrante.

Écriture

Supposons que nous disposions d'un type de données abstrait de *pile*, on peut empiler les termes, les facteurs et les opérateurs afin d'obtenir la traduction de l'expression infixée en expression préfixée :

```
struct item
{
   int type ;
   union   {
       char op  ; int  val ;
   } contenu ;

} ;
typedef struct item   ITEM ;
typedef struct item * Item ;
...
pile (Item) ; Itempile p ;

int ValLex ; int Symbole ;
...
Emettre (int Lex, int Val)
{
     ITEM x ; Item y ;

     switch (Lex)
     {
     case '+' : case '-' : case '*' : case '/' :
            x.type = OP  ;
        x.contenu = Lex ;
        y = Item_copier(&x) ;
        Itempile_empiler  (p, y) ;
        break ;

     case NB :  /* C'est un nombre */
             x.type = NB ;
        x.contenu = Val ;
        y = Item_copier(&x) ;
        Itempile_empiler  (p, y) ;
        break ;

     default :
     }
}

main ()
{

    p = Itempile_creer(Item_copier,
            Item_detruire, Item_editer) ;

    Analyse() ;
}
```

Les files

Peut-être plus encore que les piles, les files d'attente (ou simplement files) font partie de notre vie courante ; du moins en sommes-nous plus conscients, puisqu'il nous arrive quotidiennement de faire la queue devant un guichet.

Une file est un TDA formé d'un nombre variable, éventuellement nul, de données, sur lequel on peut effectuer les opérations suivantes :

— ajout d'une nouvelle donnée ;
— test déterminant si la file est vide ou non ;
— consultation de la première donnée ajoutée et non supprimée depuis (donc la plus ancienne) s'il y en a une ;
— suppression de la donnée la plus ancienne.

Cette conception s'accorde bien avec la conception intuitive que l'on a d'une file d'attente.

Les files d'attente ont une grande importance en informatique ; elles s'appliquent à deux types de problèmes :

— la simulation de files réelles ; les techniques modernes de communication (terminaux reliés à un ou plusieurs centres de communication). Il est devenu courant d'écrire des programmes qui étudient les comportements de réseaux.
— la résolution de problèmes purement informatiques, en particulier dans le domaine des systèmes d'exploitation.

Analyse fonctionnelle :

La file est constituée

— d'une suite d'éléments ordonnés $a_1, a_2, \ldots, a_n$, désignée ici par F, éventuellement vide.
— des primitives suivantes :

creer(F) Cette fonction crée une file de nom F et retourne la valeur `vrai` si l'opération a pu s'effectuer sans problème, sinon elle retourne `faux`.

filevide(F) Cette fonction teste la vacuité de la file F et retourne `vrai` si la file est vide `faux` sinon.

enfiler(x, F) Cette fonction ajoute un élément en queue de file, renvoie `faux` s'il l'élément n'a pas pu être introduit, `vrai` sinon.

defiler(F) Cette primitive retire l'élément de tête de la file.

premier(F) Cette fonction renvoie la valeur de l'élément en tête de file, si la pile n'est pas vide, sinon retourne un code d'erreur.

26.1.1 Description logique et fonctionnement

Implémentée dans un tableau, la file ne peut avoir qu'une taille maximale égale à la dimension du tableau, et peut être schématisée selon la figure 26.1.1

Notez que le fait de choisir les indices `TETE` et `QUEUE` comme nous l'avons fait, avec `TETE` désignant la position passée de la tête de la file, n'influe pas sur le problème. Cette convention facilite uniquement l'écriture des fonctions `vide` et `enfiler`.

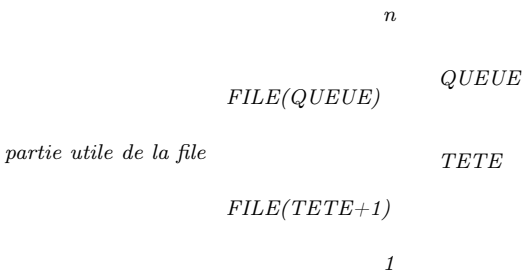


FIG. 26.1.1 – *Une file dans un tableau*

Un problème se pose : même si la taille de la file reste constamment en dessous du maximum permis n, la file "monte" inexorablement, puisque les défilages s'effectuent par le bas et les enfilages par le haut. Si l'on n'y prend garde, la file débordera du tableau au bout de n opérations `enfiler`.

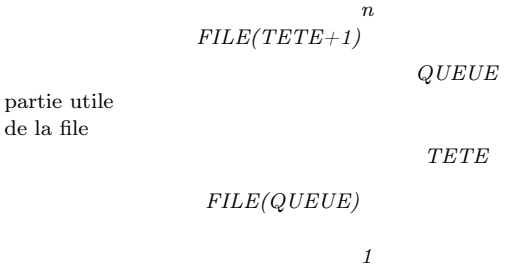


FIG. 26.1.2 – *Une file circulaire dans un tableau*

Plusieurs solutions sont possibles :

1. à chaque défilage, récupérer l'espace libéré en bas du tableau en "redescendant" toute la file d'un cran. C'est la solution simple à mettre en oeuvre, mais coûteuse sur les grandes files.
2. laisser la file monter tant qu'il reste de la place pour effectuer les enfilages ; quand il n'y a plus de place et que l'on veut opérer un enfilage, on récupère d'un seul coup la place libérée par les défilages en "redescendant" la file de toute la hauteur pos-

sible. Cette solution est plus économique que la précédente, mais exige encore des transferts d'informations inutiles.

3. quand la queue atteint le haut du tableau, effectuer les enfilages suivants à partir du bas du tableau, qui prend l'aspect de la figure 26.1.1

L'intérêt de cette méthode est qu'elle ne nécessite aucun décalage. On parle d'une représentation par file circulaire, on peut représenter chacune des deux figures précédentes par un anneau.

La programmation de cette solution présente un piège : le test déterminant si la file est vide s'écrit maintenant `TETE = QUEUE` si la file à n éléments. Pour pouvoir distinguer entre ces deux cas (file vide et file pleine), on imposera à la file la capacité maximale $n-1$ (et non n). Dans ces conditions `|TETE-QUEUE|`≥ 1 si la file n'est pas vide.

Une file générique

Il est aisé de former le type de données abstrait d'une file au moyen du code du type de la liste. Seul le fichier `File.tda` doit être écrit sur le modèle de `Liste.tda` ou `Pile.tda`.

```
#ifndef LISTE_H
#include "Liste.h"
#endif
#define file(type_objet)                        \
typedef struct nom(type_objet, file)            \
{                                               \
     void     * rep ;                           \
     type_objet (*copier_objet)   (type_objet) ;\
     void       (*detruire_objet) ()           ;\
     void       (*afficher_objet) (type_objet) ;\
} * nom(type_objet, file) ;                     \
nom(type_objet, file) nom(type_objet,file_creer)\
    (type_objet (*cop) (type_objet),            \
     void       (*det) (),                      \
     void       (*aff) (type_objet))            \
{                                               \
    nom(type_objet, file) f ;                   \
    lier_liste () ;                             \
    f = (nom(type_objet, file)) malloc          \
      (sizeof (struct nom (type_objet, file))) ;\
    f->rep = (*Liste.creer) () ;                \
    f->afficher_objet = aff ;                   \
    f->detruire_objet = det ;                   \
    f->copier_objet = cop ;                     \
    return f ;                                  \
}                                               \
void nom(type_objet, file_afficher)             \
                  (nom(type_objet, file) p)     \
{                                               \
    if (p->afficher_objet) (*Liste.afficher)    \
     (p->rep, p->afficher_objet) ;              \
    else                                        \
     printf                                     \
 ("ERR : fonction d'affichage d'obj nulle\n") ; \
}                                               \
void nom(type_objet, file_enfiler)              \
    (nom(type_objet, file) p, type_objet obj)   \
{                                               \
    (*Liste.dernier) (p->rep) ;                 \
    (*Liste.insererapres)                       \
            (p->rep, (p->copier_objet) (obj)) ; \
}                                               \
void nom(type_objet, file_desenfiler)           \
  (nom(type_objet, file) p)                     \
{                                               \
    (*Liste.premier) (p->rep) ;                 \
    (*Liste.oter) (p->rep, p->detruire_objet) ; \
}                                               \
int nom(type_objet, file_nulle)                 \
                  (nom(type_objet, file) p)     \
{                                               \
    return ((*Liste.nulle) (p->rep)) ;          \
}
```

Le type de données arbres : parcours préfixé, postfixé et infixé

Définition

Un *arbre* est une structure qui est :

- soit vide [1],
- soit composée d'un *nœud* chaîné à zéro un ou plusieurs sous-arbres ordonnés de gauche à droite.

Un sous-arbre est donc un nœud,
la *racine* est le nœud qui **n'est pas** un sous-arbre,
une *feuille* est un nœud qui **n'a pas** de sous-arbre.

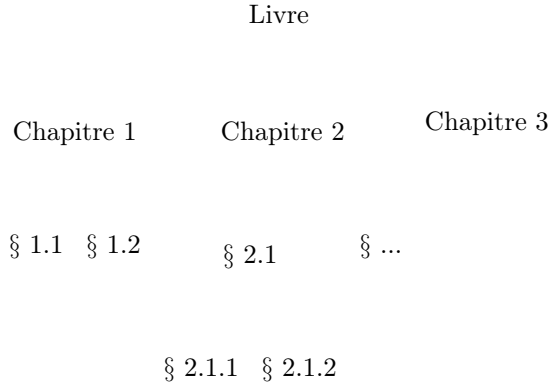


FIG. 27.1– *Une table des matières est un arbre*

Étiquettes et expressions

L'arbre *étiqueté* de la figure II représente l'expression arithmétique $(a+b)*(a+c)$. Les noms attribués aux nœuds sont $n_1, n_2, \ldots, n_7$ et les étiquettes apparaissent, comme c'est l'usage, à côté des nœuds. Les règles imposées à un arbre pour qu'il représente une expression sont les suivantes :

1. Chaque feuille est étiquetée par un opérande et est constituée de cet opérande uniquement. Ainsi la feuille n_4 représente l'expression a.
2. Chaque nœud interne n est étiqueté par un opérateur. Si n est étiqueté par un opérateur binaire θ, comme $+$ ou $*$, si son fils gauche représente l'expression E_1 et son fils droit l'expression E_2, alors n représente l'expression $(E_1)\theta(E_2)$. Les parenthèses peuvent être retirées si aucune ambiguïté n'est à craindre.

Par exemple, l'opérateur $+$ est associé au nœud n_2 et ses fils gauche et droit représentent a et b respectivement. Ainsi n_2 représente $(a) + (b)$, ou simplement $a + b$. Le nœud n_1 représente $(a+b)*(a+c)$, puisque $*$ est l'étiquette associée à n_1 et puisque $a+b$ et $a+c$ sont les expressions représentées par n_2 et n_3 respectivement.

Parcours d'arbres

Il existe plusieurs moyens de parcourir les nœuds d'un arbre. Les trois parcours les plus importants sont les parcours *préfixé*, *postfixé* et *infixé* ; ces parcours peuvent être définis récursivement.

Il existe un moyen pratique pour simuler les trois parcours d'arbre : imaginons que l'on parcourt l'arbre depuis sa racine, dans le sens contraire à celui des aiguilles d'une montre, en en restant toujours le plus près possible.

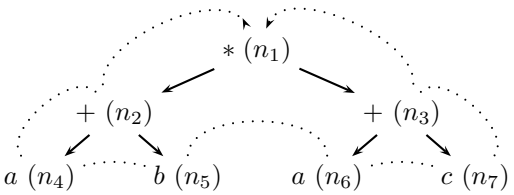


FIG. 27.3– *L'arbre d'une expression*

Dans un parcours préfixé, on ne considère que le premier passage par un nœud donné $(*+ab+ac)$; dans un parcours postfixé, on ne prend en compte que le dernier passage par un nœud, lors de la remontée vers son père $(ab+ac+*)$. Pour un parcours infixé, on liste une feuille la première fois qu'on la rencontre, mais on ne liste un nœud non terminal qu'à la deuxième rencontre : $(a+b)*(a+c)$.

1. que nous noterons Λ. Concrètement, Λ pourra être 0, -1 ou tout autre valeur significative dans un contexte particulier.

Parcours d'une arborescence de répertoires

Lister un répertoire

Écrire un programme `shell` *pour afficher les types (fichier ou répertoire) de tous les fichiers du répertoire courant.*

Pour éclairer cette question rappelons-nous qu'un répertoire dans le système Unix n'est qu'un fichier contenant sur 2 colonnes une liste de nom de fichiers [2] et le numéro de nœud associé :

homedir	
132	.
27	..
154	file1
159	file2
172	dir1
198	dir2

dir1	
172	.
132	..
201	fichier1
205	fichier2
210	rep2

rep2	
210	.
172	..
240	data

FIG. 27.3– *Exemple d'arborescence*

Dans l'exemple ci-dessus (fig. II), remarquons que le répertoire `dir1` contient un nom de fichier désigné par `.` (point) dont le numéro d'identification (`172`) est le même que le fichier `dir1` du répertoire `homedir`. Il en va de même pour le réperoire `rep2` (`210`) [3].

Remarquons aussi que le répertoire `dir1` contient un nom de fichier désigné par `..` (point, point) dont le numéro (`132`) est identique à celui du répertoire parent `homedir`. Respectivement, le nom de fichier `..` de `rep2` (`172`) correspond au répertoire parent `dir1`.

Ce travail nécessite

- une *boucle* pour effectuer la tâche de reconnaissance *pour chaque* fichier.
- une variable de boucle.

```
for FICHIER in *
do
   echo $FICHIER
done
```

On a réécrit `ls` sans option.

Pour déterminer le type de fichier, on utilise la commande `test` ou son abrégée : les crochets.

```
if [ -d $FICHIER ]
   then echo "$FICHIER repertoire"
   else echo "$FICHIER ordinaire"
fi
```

Remarques :

- $*$ est expansée par le shell [4]. Si le répertoire ne contient pas de fichier, $*$ n'est pas expansée,
- Les fichiers qui commencent par un point ne sont pas pris en compte afin d'éviter leur destruction par un `rm *`.

D'où la version définitive :

```
$ cat prog
 for FICHIER in `ls`
 do
    if [ -d $FICHIER ]
       then echo "$FICHIER repertoire"
       else echo "$FICHIER ordinaire"
     fi
done
```

FIG. 27.3– *Script-shell Types de fichiers*

La récursivité : parcours d'une arborescence

Adaptez le programme précédent pour explorer toute une arborescence à partir d'un répertoire passé en paramètre.

Le programme que l'on va construire maintenant, sera utilisé ainsi :

```
$ prog2          repertoire
commande         argument
   $0               $1
```

Pour chaque répertoire rencontré, on rapelle le programme et on poursuit l'exploration par récursivité ; il s'agit d'un parcours *préfixé*.

```
$ cat prog2
# recuperation de l'argument
REP=$1
for FICHIER in $REP/*
do
   if [ -d $FICHIER ]
      then echo "Repertoire : $FICHIER"
# fonctionne meme apres un mv
           $0 $FICHIER
      else echo "Fichier : $FICHIER"
   fi
done
```

FIG. 27.3– *Script-shell Types de fichiers dans une arborescence*

2. Il n'y s donc pas inclusion des fichiers dans les répertoires comme on en a l'illusion. Les répertoires sont comparables à une table des matières ; les chapitres n'y sont pas inclus.

3. Vous pouvez obtenir des résultats semblables en utilisant la commande `ls -i`.

4. C'est à dire que le shell remplace $*$ par la liste des fichiers du répertoire avant de lancer l'exécution du programme.

Mise en œuvre des arbres

Le TDA arbre

Les primitives les plus utiles relatives aux arbres [5] :

1. **PERE**(n, A). Cette fonction retourne le père du nœud n dans l'arbre A. Si n est la racine, le père n'existe pas et Λ est retourné (dans ce contexte, Λ est un "nœud vide").
2. **PREMIER_FILS**(n, A). Retourne le fils le plus à gauche du nœud n dans l'arbre A, ou Λ si n est une feuille (donc sans enfants).
3. **FRERE_DROIT**(n, A). Retourne le frère droit du nœud n dans l'arbre A, c'est-à-dire le nœud de même père p que n et situé immédiatement à sa droite dans l'ordre naturel des fils de p. Sur l'arbre de la figure, par exemple, PREMIER FILS$(n_2) = n_4$, FRERE DROIT$(n_4) = n_5$ et FRERE DROIT$(n_5) = \Lambda$.
4. **ETIQUETTE**(n, A). Retourne l'étiquette du nœud n dans l'arbre A. Ceci n'implique pas, malgré tout, que tout arbre soit étiqueté.
5. **CREER**(e, A). Construit un nouveau nœud d'étiquette e.
6. **RACINE**(A). Retourne la racine de l'arbre A, ou bien Λ si A est l'arbre vide.
7. **RAZ**(A). Transforme l'arbre A en arbre vide.

Représentation des arbres par premier fils et frère droit

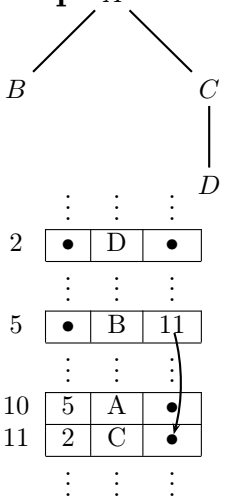


FIG. 27.4– *Un arbre*

FIG. 27.4– *Mise en œuvre d'un arbre*

Programmation du TDA arbre générique

La totalité du TDA arbre est donnée en annexe.

5. Cf. 45

Troisième partie

Annexes

Le TDA liste

A.1 Le fichier Liste.c

```
#include <stdio.h>
#include "Liste.h"

/* #define TRACE  */

static char * nom_module = "Liste generique" ;

/* Error */

static void ERR_vide(char *mess)
{
    printf ("\n Erreur dans la fonction %s in %s\n", mess, nom_module) ;

    exit (1) ;
}

/* Creation d'une cellule tete de liste */

static liste creer ()
{
    liste p ;

    p = (liste) malloc (sizeof (struct tete)) ;

    p->prem = p->vue = p->der = p->s = NULL ;

    return p ;
}

/* Destruction d'une liste */

static void detruire (liste l, void (*det_obj) (void *))
{
    l->vue = l->prem ;
    while (l->vue)
    {
    (*det_obj) (l->vue->objet) ;
    l->prem = l->vue ;
    l->vue = l->vue->suiv ;
    free (l->prem) ;
    }
    l->prem = l->der = l->s = NULL ;
}

/* Test d'une liste vide */

static int nulle (liste l)
{
    return (NULL == l->prem) ;
}

/* Test de fin de liste */

static int fin (liste l)
{
```

```
    return (NULL == l->vue) ;
}

/* Positionne la vue en premiere position */

static void premier (liste l)
{
    l->vue = l->prem ;
}

/* Positionne la vue en derniere position */

static void dernier (liste l)
{
    l->vue = l->der ;
}

/* Positionne la vue sur l'element suivant */

static void suivant (liste l)
{
    if (l->vue)
    l->vue = l->vue->suiv ;
    else ERR_vide("suivant") ;
}

static cellule precedent (liste p)
{
    cellule q ;

    q = p->prem ;

#ifdef TRACE
printf ("prem : %x vue : %x\n", p->prem, p->vue ) ;
#endif
    if (q == p->vue)
    return (NULL) ;
    else
    while (q->suiv != p->vue)
    {
#ifdef TRACE
printf ("Recherche = prem : %x vue : %x\n", p->prem, p->vue ) ;
#endif
        q = q->suiv ;
    }
   return q ;
}


/* Ajout d'un objet avant la vue */

static void insereravant (liste l, void * obj)
{
    cellule p, s ;

    p = (cellule) malloc (sizeof (struct cell)) ;
    p->objet = obj ; p->suiv = NULL ;

    if (!l->prem)
    l->prem = l->der = p ;
    else
    if (!l->vue)             /* insertion en derniere pos */
        l->der = l->der->suiv = p ;
    else
```

```
    {
        s = precedent (l) ;
        if (s)
        s->suiv = p ;
        else
        l->prem = p ;
        p->suiv = l->vue ;
    }
    l->vue = p ;
}


static void insererapres (liste l, void * obj)
{
    cellule p, s ;

#ifdef TRACE
printf ("Je vais insererapres\n") ;
#endif
    p = (cellule) malloc (sizeof (struct cell)) ;
    p->objet = obj ; p->suiv = NULL ;

    if (!l->prem)               /* liste vide */
    {
#ifdef TRACE
printf ("Premier = obj : %x prem : %x\n", obj, l->prem) ;
#endif
    l->prem = l->der = p ;
    }
    else
    {
    s = l->vue->suiv ;
#ifdef TRACE
printf ("vue : %x, s : %x \n", l->vue, s) ;
#endif
    if (!s)                   /* placer a la fin */
    {
        l->der = p ;
    }
    else
    {
        p->suiv = l->vue->suiv ;
    }
    l->vue->suiv = p ;
    }
    l->vue = p ;
#ifdef TRACE
printf ("apres ins = prem : %x vue : %x der : %x\n", l->prem, l->vue, l->der ) ;
#endif
}

/* Remplacement d'un objet */

static void remplacer (liste l, void *obj, void (*det_obj) (void *))
{
    if (!l->vue)
    ERR_vide("remplacer") ;
    (*det_obj) (l->vue->objet) ;
    l->vue->objet = obj ;
}

/* Suppression d'une cellule */

static void oter (liste l, void (*det_obj) ())
```

```
{
    cellule s, q ;

    if (l->vue)
    {
    q = l->vue ;
    if (l->vue == l->prem)
    {
        l->vue = l->prem = (l->vue)->suiv ;    /* !!! */
        if (!l->vue) l->der = NULL ;
    }
    else
    {
        s = precedent (l) ;
        s->suiv = (l->vue)->suiv  ;          /* !!! */
        if (l->vue == l->der)
        l->der = l->vue = s ;
        else
        l->vue = l->vue->suiv ;
    }

    (*det_obj) (q->objet) ; free (q) ;
    }
    else ERR_vide ("oter") ;
}

/* Lecture d'un objet */

static void * lire (liste l)
{
    if (l->vue)
    return l->vue->objet ;
    else
    ERR_vide ("lire") ;
}

/* Sauvegarde de la position de la vue */

static void fixer (liste l)
{
    cellule n ;

    n = (cellule) malloc (sizeof (struct cell)) ;
    n->objet = l->vue ; n->suiv = l->s ; l->s = n ;
}

/* Retablissement de la vue sur la derniere sauvegarde */

static int retablir (liste l)
{
    cellule q ;

    if (l->s)
    {
    q = l->s ;
    l->s = l->s->suiv ;
    l->vue = q->objet ; free (q) ;
    return 1 ;
    }
    return 0 ;
}

/* Affichage d'une liste  */
```

```
static void afficher (liste l, void (*afic) (void *))
{
    cellule p = l->prem ;

    printf ("(") ;

    while (p)
    {
    (*afic) (p->objet) ;
    if (p->suiv) printf (", ") ;
    p = p->suiv ;
    }

    printf (")\n") ;
}

void * lier_liste ()
{
    Liste.creer        = &creer        ;
    Liste.copier       = 0             ;
    Liste.detruire     = &detruire     ;
    Liste.nulle        = &nulle        ;
    Liste.premier      = &premier      ;
    Liste.dernier      = &dernier      ;
    Liste.suivant      = &suivant      ;
    Liste.fin          = &fin          ;
    Liste.lire         = &lire         ;
    Liste.insereravant = &insereravant ;
    Liste.insererapres = &insererapres ;
    Liste.remplacer    = &remplacer    ;
    Liste.oter         = &oter         ;
    Liste.fixer        = &fixer        ;
    Liste.retablir     = &retablir     ;
    Liste.afficher     = &afficher     ;
}
```

A.2 Le fichier Liste.h

```
#define LISTE_H
#include <stdio.h>
#include <stdlib.h>

#define nom(a,b) a##b

typedef struct cell
{
    void        * objet ;
    struct cell * suiv  ;
} * cellule ;

typedef struct tete
{
    cellule prem, vue, der, s ;
} * liste ;

extern void * lier_liste () ;
struct fonctions
{
    liste  (*creer)      ()                          ;
```

```
    void   (*copier)     (void *)                   ;
    void   (*detruire)   (liste, void (*) (void*))  ;
    int    (*nulle)      (liste)                    ;
    void   (*premier)    (liste)                    ;
    void   (*dernier)    (liste)                    ;
    void   (*suivant)    (liste)                    ;
    int    (*fin)        (liste)                    ;
    void * (*lire)       (liste)                    ;
    void   (*insereravant) (liste, void *)          ;
    void   (*insererapres) (liste, void *)          ;
    void   (*remplacer)  (liste, void *, void (*) (void*)) ;
    void   (*oter)       (liste, void (*) ())       ;
    void   (*fixer)      (liste)                    ;
    int    (*retablir)   (liste)                    ;
    void   (*afficher)   (liste, void (*) ())       ;
} Liste ;
```

A.3 Le fichier Liste.tda

```
#ifndef LISTE_H
#include "Liste.h"
#endif


#define liste(type_objet)                                       \
typedef struct nom(type_objet, liste)                           \
{                                                               \
    void *     rep ;                                            \
    type_objet (*copier_objet)   (type_objet) ;                 \
    void       (*detruire_objet) (type_objet) ;                 \
    void       (*afficher_objet) (type_objet) ;                 \
} * nom(type_objet, liste) ;                                    \
nom(type_objet, liste) nom(type_objet, liste_creer)             \
    (type_objet (*copier)   (type_objet),                       \
     void       (*detruire) (type_objet),                       \
     void       (*editer)   (type_objet))                       \
{                                                               \
    nom(type_objet, liste) l ;                                  \
    lier_liste () ;                                             \
    l = (nom(type_objet, liste)) malloc                         \
     (sizeof (struct nom (type_objet, liste))) ;                \
    l->rep = (*Liste.creer) () ;                                \
    l->copier_objet = copier ;                                  \
    l->detruire_objet = detruire ;                              \
    l->afficher_objet = editer ;                                \
    return l ;                                                  \
}                                                               \
void nom (type_objet, liste_insererapres)                       \
    (nom (type_objet, liste) l, type_objet e)                   \
{                                                               \
    (*Liste.insererapres) (l->rep, (l->copier_objet) (e)) ;     \
}                                                               \
void nom (type_objet, liste_insereravant)                       \
    (nom(type_objet, liste) l, type_objet e)                    \
{                                                               \
    (*Liste.insereravant) (l->rep, (l->copier_objet) (e)) ;     \
}                                                               \
void nom (type_objet, liste_premier) (nom(type_objet, liste) l) \
{                                                               \
```

```
    (*Liste.premier) (l->rep) ;                                    \
}                                                                  \
void nom (type_objet, liste_dernier) (nom(type_objet, liste) l) \
{                                                                  \
    (*Liste.dernier) (l->rep) ;                                    \
}                                                                  \
void nom (type_objet, liste_suivant) (nom(type_objet, liste) l) \
{                                                                  \
    (*Liste.suivant) (l->rep) ;                                    \
}                                                                  \
int nom(type_objet, liste_nulle) (nom(type_objet, liste) l)     \
{                                                                  \
    return (*Liste.nulle) (l->rep) ;                               \
}                                                                  \
int nom(type_objet, liste_fin) (nom(type_objet, liste) l)       \
{                                                                  \
    return (*Liste.fin) (l->rep) ;                                 \
}                                                                  \
type_objet nom(type_objet, liste_lire)                          \
               (nom(type_objet, liste) l)                          \
{                                                                  \
    return                                                         \
     (l->copier_objet) ((type_objet) (*Liste.lire) (l->rep)) ;  \
}                                                                  \
void nom(type_objet, liste_afficher) (nom(type_objet, liste) l) \
{                                                                  \
    if (l->afficher_objet)                                         \
    (*Liste.afficher) (l->rep, l->afficher_objet) ;                \
    else                                                           \
    printf ("ERR : fonction d'affichage  nulle\n") ;               \
}
```

A.4 Application : le fichier usager.c

```
/* Ecp - 1ere Annee -
 * jjd                        Liste1.c
 */
#include "Liste.tda"
typedef int  *entier ;     /* Creation du nouveau type pointeur sur int */
liste(entier) ;

/* fonctions particulieres au nouveau type */

void entier_editer(entier i)
{ printf("%d", *i) ; }

entier entier_copier(entier i)
{
    entier      j ;
    j = (entier) malloc(sizeof(*i)) ;
    *j = *i ;
    return j ;
}

void main()
{
    entierliste   l ;
    entier        x ;
    int           i ;
```

```
/* Instanciation */
    l = entierliste_creer(entier_copier, 0, entier_editer) ;

    for (i = 0 ; i < 5 ; i++)
    {
        x = entier_copier((entier) & i)         ;
        entierliste_insererapres(l, (entier) x) ;
    }
    entierliste_afficher(l) ;
}
```

Le TDA pile

B.1 Le fichier Pile.tda

```
#ifndef LISTE_H
#include "Liste.h"
#endif


#define pile(type_objet)                                                \
typedef struct nom(type_objet, pile)                                    \
{                                                                       \
    void     * rep ;                                                    \
    type_objet (*copier_objet)   (type_objet) ;                         \
    void       (*detruire_objet) ()           ;                         \
    void       (*afficher_objet) (type_objet) ;                         \
} * nom(type_objet, pile) ;                                             \
nom(type_objet, pile) nom(type_objet, pile_creer)                       \
    (type_objet (*cop) (type_objet),                                    \
     void       (*det) (),                                              \
     void       (*aff) (type_objet))                                    \
{                                                                       \
    nom(type_objet, pile) p ;                                           \
    lier_liste () ;                                                     \
    p = (nom(type_objet, pile)) malloc                                  \
     (sizeof (struct nom (type_objet, pile))) ;                         \
    p->rep = (*Liste.creer) () ;                                        \
    p->afficher_objet = aff ;                                           \
    p->detruire_objet = det ;                                           \
    p->copier_objet = cop ;                                             \
    return p ;                                                          \
}                                                                       \
void nom(type_objet, pile_empiler)                                      \
    (nom(type_objet, pile) p, type_objet obj)                           \
{                                                                       \
    (*Liste.insereravant) (p->rep, (p->copier_objet) (obj)) ;           \
}                                                                       \
void nom(type_objet, pile_desempiler)                                   \
  (nom(type_objet, pile) p)                                             \
{                                                                       \
    (*Liste.oter) (p->rep, p->detruire_objet) ;                         \
}                                                                       \
void nom(type_objet, pile_afficher)                                     \
                  (nom(type_objet, pile) p)                             \
{                                                                       \
    if (p->afficher_objet)                                              \
    (*Liste.afficher) (p->rep, p->afficher_objet) ;                     \
    else                                                                \
    printf ("ERR : fonction d'affichage d'obj nulle\n") ;               \
}
/*                                                                      \
void nom(type_objet, pile_detruire) (nom(type_objet, pile) p)           \
{                                                                       \
    (*Liste.detruire) (p->rep, p->detruire_objet) ;                     \
}                                                                       \
int nom(type_objet, pile_nulle) (nom(type_objet, pile) p)               \
{                                                                       \
    return ((*Liste.nulle) (p->rep)) ;                                  \
}                                                                       \
```

```
nom (type_objet, pile) nom(type_objet, pile_copier)                    \
                               (nom(type_objet, pile) t)               \
{                                                                      \
    nom(type_objet, pile)  p ;                                         \
    p = nom(type_objet, pile_creer)                                    \
          (t->copier_objet, t->detruire_objet, t->afficher_objet) ;    \
    p->rep = (*Liste.copier) (t->rep, t->copier_objet) ;               \
    return p ;                                                         \
}                                                                      \
type_objet nom(type_objet, pile_lire)) (nom(type_objet, pile) p)       \
{                                                                      \
    return (p->copier_objet)                                           \
    ((type_objet) (*Liste.lire) (p->rep)) ;                            \
}
*/
```

B.2 Application : le fichier expression.c

```
#include "Pile.tda"

typedef char * lettre ;


void lettre_editer (lettre i)
{
    printf ("%c", *i) ;
}

lettre lettre_copier (lettre i)
{
    lettre j ;
    j = (lettre) malloc (sizeof (*i)) ;
    *j = *i ;
    return j ;
}

void lettre_detruire (lettre i)
{
    free (i) ;
}

pile (lettre) ;    /* utilise les  definitions ci-dessus */

void main ()
{
    lettrepile p ; int i ; lettre x ;

    p = lettrepile_creer(lettre_copier, lettre_detruire, lettre_editer) ;

    for (i = 65 ; i < 70 ; i++)
    {
    x = lettre_copier   ((lettre) &i) ;
        lettrepile_empiler  (p, x)        ;
        lettrepile_afficher (p)           ;
    }

    for (i = 0 ; i < 5 ; i++)
    {
        lettrepile_desempiler (p) ;
```

```
        lettrepile_afficher   (p) ;
    }
}
```

B.3 Application : le fichier inf2pre.c

```
#include <stdio.h>
#include <ctype.h>
#include "Pile.tda"

#define RIEN        -1
#define NB         256
#define OP         257
#define EOE        258


struct item
{
   int type ;
   union
   {
       char op  ;
       int  val ;
   } contenu ;

} ;

typedef struct item   ITEM ;
typedef struct item * Item ;


void Item_editer (Item i)
{
    if (OP == i->type ) printf ("%c", i->contenu) ;
    else printf ("%d", i->contenu) ;
}

Item Item_copier (Item i)
{
    Item j ;

    j = (Item) malloc (sizeof (*i)) ;
    *j = *i ;
    return j ;
}

void Item_detruire (Item i)
{
    free (i) ;
}

pile (Item) ;
Itempile p ;

int ValLex ;
int Symbole ;

Erreur (char * Message)
{
```

```
    fprintf (stderr, "%s\n", Message) ;
    exit (1) ;
}

int AnalLex ()
{
    int T ;

    while (1)
    {
    T = getchar() ;
    if (' ' == T || '\t' == T)
       ;     /* On saute les blancs */
    else if ('\n' == T || ';' == T)
    {
       return EOE ;
    }
    else if (isdigit(T))
    {
        ungetc (T, stdin) ;
        scanf  ("%d", &ValLex) ;
        return NB ;
    }
    else if (EOF == T)
        return EOF ;
    else
    {
        ValLex = RIEN ;
        return T ;
    }
    }
}

Analyse ()
{
    Symbole = AnalLex () ;
    while (EOF != Symbole)
    {
    Expr () ;
    if (EOE == Symbole)
    {
        Itempile_afficher(p) ;
        Accepter (Symbole) ;
    }
    }
}


Expr ()
{
     int T ;

     Terme () ;
     while (1)
     switch (Symbole)
     {
         case '+' : case '-' :
         T = Symbole ;
         Accepter (Symbole) ;
         Terme () ;
         Emettre (T, RIEN) ;
         continue ;
         default :
         return ;
```

```
        }
}

Terme ()
{
    int T ;

    Facteur () ;

    while (1)
    switch (Symbole)
    {
        case '*' : case '/' :
         T = Symbole ;
         Accepter (Symbole) ;
         Facteur () ;
         Emettre (T, RIEN) ;
         continue ;
         default :
         return ;
    }
}

Facteur ()
{
    switch (Symbole)
    {
    case '(' :
        Accepter ('(') ; Expr () ; Accepter (')') ; break ;
    case NB :
            Emettre (NB, ValLex) ; Accepter (NB) ; break ;
    case EOE :
               break ;
    default :
        Erreur ("Syntaxe") ;
    }
}

Accepter (int Lex)
{
     if (Symbole == Lex)
     Symbole = AnalLex () ;
     else Erreur ("Syntaxe") ;
}



Emettre (Lex, Val)
int Lex, Val ;
{
     ITEM x ; Item y ;

     switch (Lex)
     {
     case '+' : case '-' : case '*' : case '/' :
         /* printf ("%c\n", Lex) ; break ; */
             x.type = OP ;
         x.contenu = Lex ;
         y = Item_copier(&x) ;
         Itempile_empiler  (p, y) ;
         break ;

     case NB :
         /* printf ("%d\n", Val) ; break ; */
```

```
            x.type = NB ;
        x.contenu = Val ;
        y = Item_copier(&x) ;
        Itempile_empiler  (p, y) ;
        break ;

     default :
        printf ("Lexeme %d, ValLex %d\n", Lex, Val) ;
     }
}

main ()
{

    p = Itempile_creer(Item_copier, Item_detruire, Item_editer) ;

    Analyse() ;
    exit (0)  ;
}
```

B.4 Application : le fichier entier.c

```
#include "Liste.tda"

typedef int * entier ;


void entier_editer (entier i)
{
    printf ("%d", *i) ;
}

void entier_detruire (entier i)
{
    free (i) ;
}

entier entier_copier (entier i)
{
    entier j ;

    j = (entier) malloc (sizeof (*i)) ;
    *j = *i  ;

    return j ;
}

liste(entier) ;

void main ()
{
    entierliste l ; int i ; entier x ;

    l = entierliste_creer (entier_copier,
                           entier_detruire,
                           entier_editer) ;

    for (i = 10 ; i < 15 ; i++)
```

```
    {
        x = entier_copier   (&i)   ;
        entierliste_insereravant (l, x) ;
        entierliste_afficher     (l)    ;
    }

    for (i = 0 ; i < 5 ; i++)
    {
        x = entier_copier   (&i)   ;
        entierliste_insererapres (l, x) ;
        entierliste_afficher     (l)    ;
    }

    printf ("\nl = ") ;
    entierliste_afficher (l) ;

    printf ("Element courant : ") ;
    entier_editer (entierliste_lire(l)) ; printf ("\n") ;
    entierliste_suivant(l) ;
    printf ("Element suivant : ") ;
    entier_editer (entierliste_lire(l)) ; printf ("\n") ;
    entierliste_premier(l) ;
    printf ("Element premier : ") ;
    entier_editer (entierliste_lire(l)) ; printf ("\n") ;
    entierliste_suivant(l) ;
    printf ("Element deuxieme : ") ;
    entier_editer (entierliste_lire(l)) ; printf ("\n") ;
    entierliste_dernier(l) ;
    printf ("Element dernier : ") ;
    entier_editer (entierliste_lire(l)) ; printf ("\n") ;
}
```

Le TDA file

C.1 Le fichier File.tda

```
#ifndef LISTE_H
#include "Liste.h"
#endif
#define file(type_objet)                          \
typedef struct nom(type_objet, file)              \
{                                                 \
     void     * rep ;                             \
     type_objet (*copier_objet)   (type_objet) ;\
     void       (*detruire_objet) ()           ;\
     void       (*afficher_objet) (type_objet) ;\
} * nom(type_objet, file) ;                       \
nom(type_objet, file) nom(type_objet,file_creer)\
    (type_objet (*cop) (type_objet),              \
     void       (*det) (),                        \
     void       (*aff) (type_objet))              \
{                                                 \
    nom(type_objet, file) f ;                     \
    lier_liste () ;                               \
    f = (nom(type_objet, file)) malloc            \
      (sizeof (struct nom (type_objet, file))) ;\
    f->rep = (*Liste.creer) () ;                  \
    f->afficher_objet = aff ;                     \
    f->detruire_objet = det ;                     \
    f->copier_objet = cop ;                       \
    return f ;                                    \
}                                                 \
void nom(type_objet, file_afficher)               \
                  (nom(type_objet, file) p)       \
{                                                 \
    if (p->afficher_objet) (*Liste.afficher)      \
     (p->rep, p->afficher_objet) ;                \
    else                                          \
     printf                                       \
 ("ERR : fonction d'affichage d'obj nulle\n") ; \
}                                                 \
void nom(type_objet, file_enfiler)                \
    (nom(type_objet, file) p, type_objet obj)     \
{                                                 \
    (*Liste.dernier) (p->rep) ;                   \
    (*Liste.insererapres)                         \
            (p->rep, (p->copier_objet) (obj)) ; \
}                                                 \
void nom(type_objet, file_desenfiler)             \
  (nom(type_objet, file) p)                       \
{                                                 \
    (*Liste.premier) (p->rep) ;                   \
    (*Liste.oter) (p->rep, p->detruire_objet) ; \
}                                                 \
int nom(type_objet, file_nulle)                   \
                  (nom(type_objet, file) p)       \
{                                                 \
    return ((*Liste.nulle) (p->rep)) ;            \
}
```

C.2 Application : Le fichier attente.c

```
#include "File.tda"

typedef char * lettre ;


void lettre_editer (lettre i)
{
    printf ("%c", *i) ;
}

lettre lettre_copier (lettre i)
{
    lettre j ;
    j = (lettre) malloc (sizeof (*i)) ;
    *j = *i ;
    return j ;
}

void lettre_detruire (lettre i)
{
    free (i) ;
}

file (lettre) ;    /* utilise les  definitions ci-dessus */

void main ()
{
    lettrefile p ; int i ; lettre x ;

    p = lettrefile_creer(lettre_copier, lettre_detruire, lettre_editer) ;

    for (i = 65 ; i < 70 ; i++)
    {
        x = lettre_copier   ((lettre) &i) ;
        lettrefile_enfiler  (p, x)        ;
        lettrefile_afficher (p)           ;
    }

    for (i = 0 ; i < 5 ; i++)
    {
        lettrefile_desenfiler (p) ;
        lettrefile_afficher   (p) ;
    }
}
```

C.3 Application : Le fichier inf2post.c

```
#include <stdio.h>
#include <ctype.h>
#include "File.tda"

#define RIEN        -1
#define NB         256
#define OP         257
#define EOE          258
```

```
struct item
{
   int type ;
   union
   {
       char op  ;
       int  val ;
   } contenu ;

} ;

typedef struct item   ITEM ;
typedef struct item * Item ;


void Item_editer (Item i)
{
    if (OP == i->type ) printf ("%c", i->contenu) ;
    else printf ("%d", i->contenu) ;
}

Item Item_copier (Item i)
{
    Item j ;

    j = (Item) malloc (sizeof (*i)) ;
    *j = *i ;
    return j ;
}

void Item_detruire (Item i)
{
    free (i) ;
}

file (Item) ;
Itemfile p ;

int ValLex ;
int Symbole ;

Erreur (char * Message)
{
    fprintf (stderr, "%s\n", Message) ;
    exit (1) ;
}

int AnalLex ()
{
    int T ;

    while (1)
    {
        T = getchar() ;
        if (' ' == T || '\t' == T)
           ;     /* On saute les blancs */
        else if ('\n' == T || ';' == T)
        {
           return EOE ;
        }
        else if (isdigit(T))
        {
            ungetc (T, stdin) ;
```

```
            scanf  ("%d", &ValLex) ;
            return NB ;
        }
        else if (EOF == T)
            return EOF ;
        else
        {
            ValLex = RIEN ;
            return T ;
        }
    }
}

Analyse ()
{
    Symbole = AnalLex () ;
    while (EOF != Symbole)
    {
        Expr () ;
        if (EOE == Symbole)
        {
            Itemfile_afficher(p) ;
            Accepter (Symbole) ;
        }
    }
}


Expr ()
{
     int T ;

     Terme () ;
     while (1)
     switch (Symbole)
     {
         case '+' : case '-' :
         T = Symbole ;
         Accepter (Symbole) ;
         Terme () ;
         Emettre (T, RIEN) ;
         continue ;
         default :
         return ;
     }
}

Terme ()
{
    int T ;

    Facteur () ;

    while (1)
    switch (Symbole)
    {
        case '*' : case '/' :
         T = Symbole ;
         Accepter (Symbole) ;
         Facteur () ;
         Emettre (T, RIEN) ;
         continue ;
         default :
         return ;
```

```
    }
}

Facteur ()
{
    switch (Symbole)
    {
    case '(' :
        Accepter ('(') ; Expr () ; Accepter (')') ; break ;
    case NB :
            Emettre (NB, ValLex) ; Accepter (NB) ; break ;
    case EOE :
               break ;
    default :
        Erreur ("Syntaxe") ;
    }
}

Accepter (int Lex)
{
     if (Symbole == Lex)
     Symbole = AnalLex () ;
     else Erreur ("Syntaxe") ;
}



Emettre (Lex, Val)
int Lex, Val ;
{
     ITEM x ; Item y ;

     switch (Lex)
     {
     case '+' : case '-' : case '*' : case '/' :
         /* printf ("%c\n", Lex) ; break ; */
             x.type = OP ;
         x.contenu = Lex ;
         y = Item_copier(&x) ;
         Itemfile_enfiler  (p, y) ;
         break ;

     case NB :
         /* printf ("%d\n", Val) ; break ; */
             x.type = NB ;
         x.contenu = Val ;
         y = Item_copier(&x) ;
         Itemfile_enfiler  (p, y) ;
         break ;

     default :
         printf ("Lexeme %d, ValLex %d\n", Lex, Val) ;
     }
}

main ()
{

    p = Itemfile_creer(Item_copier, Item_detruire, Item_editer) ;

    Analyse() ;
    exit (0)  ;
}
```

Le TDA arbre

D.1 Le fichier Arbrebin.c

```
static char * nom_module = "arbre binaire" ;

typedef struct noeud
{
    void         * objet                   ;
    struct noeud * gauche, * droit, * pere ;
} * noeud ;

struct csvg
{
    noeud         svue  ;
    struct csvg * sprec ;
} ;

typedef struct tete
{
    noeud         racine, vue ;
    struct csvg * svg         ;
} * arbrebin ;

typedef enum {racine, pere, gauche, droite} direction ;

struct
{
    arbrebin (*creer)     ()                                    ;
    void     (*detruire)  (arbrebin, void (*) (void *))         ;
    int      (*existe)    (arbrebin, direction)                 ;
    void *   (*lire)      (arbrebin)                            ;
    int      (*aller)     (arbrebin, direction)                 ;
    void     (*ajouter)   (arbrebin, direction, void *)         ;
    int      (*oter)      (arbrebin)                            ;
    void     (*remplacer) (arbrebin, void *, void (*) (void *)) ;
    void     (*fixer)     (arbrebin)                            ;
    int      (*retablir)  (arbrebin)                            ;
    void     (*afficher)  (arbrebin, void (*) (void *))         ;
    void     (*deplacer)  (arbrebin, direction)                 ;
    void     (*greffer)   (arbrebin, arbrebin, direction)       ;
    arbrebin (*tailler)   (arbrebin, direction)                 ;
} Arbrebin ;

static arbrebin creer ()
{
    arbrebin r ;

    r = (arbrebin) malloc (sizeof (struct tete)) ;
    r->racine = r->vue : r->svg = NULL           ;

    return r ;
}

static void detruire (arbrebin r, void (*det_obj) (void *))
{
    noeud x ;
    r->vue = r->racine ;
```

```
    while (r->vue)
    {
        while (r->vue->gauche)
            r->vue = r->vue->gauche ;
        while (!r->vue->droite)
        {
            x = r->vue ;
            if (r->pere)
            r->vue = r->pere ;
            else
            {
                (*det_obj) (x->objet)     ;
                free (x)                  ;
                r->vue = r->racine = NULL ;
                return                    ;
            }
            (*det_obj) x->objet) ;
            free (x)             ;
        }
        x = r->vue->droite    ;
        r->vue->droite = NULL ;
        r->vue = x            ;
    }
}

static int existe (arbrebin r, direction d)
{
    switch (d)
    {
    case racine : return (r->racine ? 1           : 0) ;
    case gauche : return (r->vue ? r->vue->gauche : 0) ;
    case droite : return (r->vue ? r->vue->droite : 0) ;
    case pere   : return (r->vue ? r->vue->pere   : 0) ;
    }
}

static void * lire (arbrebin r)
{
    if (!r->vue) { ERR_objet ("lire") ; return NULL ; }
    return r->vue->objet ;
}

static int aller (arbrebin r, direction d)
{
    switch (d)
    {
    case recine : r->vue     = r->racine ; return 1 ;
    case gauche : if (!r->vue) {ERR_vue ("aller") ; return 0 ;
              if (r->vue->gauche)
              {   r->vue = r->vue->gauche ; return 1 }
              return 0 ;
    case droite : if (!r->vue) {ERR_vue ("aller") ; return 0 ;
              if (r->vue->droite)
              {   r->vue = r->vue->droite ; return 1 }
              return 0 ;
    case pere   : if (!r->vue) {ERR_vue ("aller") ; return 0 ;
              if (r->vue->pere)
              {   r->vue = r->vue->pere ; return 1 }
              return 0 ;
    }
}

static void ajouter (arbrebin r, direction d, void * obj)
{
```

```
    noeud n ;

    n = (noeud) malloc (sizeof (struct noeud)) ;
    n->gauche = n->droite = NULL ;
    n->objet  = obj ;

    switch (d)
    {
    case racine : r->vue    = n         ;
              n->gauche = r->racine ;
              if (r->racine) r->racine->pere = n ;
              r->racine = n ;
              break ;
    case gauche : n->gauche      = r->vue->gauche ;
              r->vue->gauche = n              ;
              n->pere        = r->vue         ;
              if (n->gauche) n->gauche->pere = n ;
              r->vue         = r->vue->gauche ;
              break ;
    case droite : n->droite      = r->vue->droite ;
              r->vue->droite = n              ;
              n->pere        = r->vue         ;
              if (n->droite) n->droite->pere = n ;
              r->vue         = r->vue->droite ;
              break ;
       case pere    : if (!r->vue->pere) r->racine = n ;
              n->pare        = r->vue->pere   ;
              r->vue->pere   = n              ;
              n->gauche      = r->vue         ;
              if (n->pere)
              if (n->pere->gauche == r->vue)
                  n->pere->gauche = n ;
              else
                  n->pere->droite = n ;
              r->vue         = n              ;
    }
}

/* static int oter (arbrebin, void (*) (void *)) */
static int oter (arbrebin r)
{
    noeud s, x ;

    if (!r->vue) { ERR_vue ("Oter") ; return 0 ; }

    if ((r->vue->gauche) && (r->vue->droite)) return 0 ;
    s = r->vue->pere ;
    if (s)
    if (s->gauche == r->vue)
        x = s->gauche =
        ! r->vue->gauche ? r->vue->droite : r->vue->gauche ;
    else
        x = s->droite =
        ! r->vue->gauche ? r->vue->droite : r->vue->gauche ;
    else
    x = s->racine =
    ! r->vue->gauche ? r->vue->droite : r->vue->gauche ;

    if (x) x->pere = s ;
    oterfixer (r) ;
    free (r->vue) ;
    r->vue = x ? x : s ;
    return 1 ;
}
```

```
static void remplacer (arbrebin r, void *obj, void (*det_obj) (void *))
{
    if (!r->vue)
    { ERR_noeud ("Remplacer" ; return ; }
    (*det_obj) (r->vue->objet) ;
    r->vue->objet = obj ;
}

static void fixer (arbrebin r)
{
    struct csvg * q ;
    q = (struct csvg *) malloc (sizeof (struct csvg)) ;
    q->svue = r->vue ; q->sprec = r->svg ; r->svg = q ;
}

static int retablir (arbrebin r)
{
    struct csvg * q ;
    if (r->svg)
    {
        q = r->svg ; r->svg = r->svg->sprec ;
        free (q) ;
        return 1 ;
    }
    return 0 ;
}

static void afficher (arbrebin r, void (*affic) (void *))
{
    if (!r->racine)
        printf ("()") ;
    else _afficher (r->racine, affic) ;
}

static void _afficher (noeud n, void (*afficher) (void *))
{
    if (!(n->gauche) && !(n->droite))
    (*affic) (n->objet) ;
    else
    {
        printf ("(") ;
        (*affic) (n->n->objet) ;

        if (n->gauche)
            _afficher (n->gauche, affic) ;
        else printf ("-") ;

        if (n->droite)
            _afficher (n->droite, affic) ;

        printf (")") ;
    }
}

static void deplacer (arbrebin r, direction d)
{
    noeud n ;

    if (!verifvue(r)) { ERR_vue() ; return ; }

    n = (noeud) malloc (sizeof (struct noeud) ; n->objet = NULL ;

    switch (d)
```

```
    {
    case racine : n->gauche = r->svg->svue ;
              n->droite = r->vue       ;
              n->pere   = NULL         ;
              r->racine = r->vue = n   ;
              break ;
    case pere   : n->gauche = r->svg->svue ;
              n->droite = r->vue       ;
              n->pere   = r->vue->pere ;
              r->vue->pere = n ;
              r->vue       = n ;
              break ;
    case gauche : if (r->vue->gauche)
              {
              n->gauche = r->svg->svue   ;
              n->droite = r->vue->gauche ;
              n->pere   = r->vue         ;
              r->vue->gauche->pere = n   ;
              r->vue->gauche       = n   ;
              r->vue               = n   ;
              }
              else
              {
              n              = r->vue         ;
              r->vue->gauche = r->svg->vue    ;
              r->vue         = r->vue->gauche ;
              }
    case droite : if (r->vue->droite)
              {
              n->gauche = r->svg->svue   ;
              n->droite = r->vue->gauche ;
              n->pere   = r->vue         ;
              r->vue->droite->pere = n   ;
              r->vue->droite       = n   ;
              r->vue               = n   ;
              }
              else
              {
              n              = r->vue         ;
              r->vue->droite = r->svg->vue    ;
              r->vue         = r->vue->droite ;
              }
    }
    if (r->svg->svue->pere->gauche == r->svg->svue)
         r->svg->svue->pere->gauche = NULL ;
    else r->svg->svue->pere->droite = NULL ;

    r->svg->svue->pere = n ;
    r->svg = r->svg->sprec ;
}

static void greffer (arbrebin r, arbrebin s, direction d)
{
    noeud n ;
    n = (noeud) malloc (sizeof (struct noeud) ; n->objet = NULL ;

    switch (d)
    {
    case racine : n->gauche = r->racine ;
              n->droite = s->vue    ;
              n->pere   = NULL      ;
              n->gauche->pere = n->droite->pere = n ;
              r->vue = r->racine = n ;
              break ;
```

```
    case pere   : if (r->vue->pere)
                {
                n->gauche       = r->vue              ;
                n->droite       = s->vue              ;
                n->pere         = r->vue->pere        ;
                n->gauche->pere = n->droite->pere = n ;
                r->vue          = n                   ;
                }
                else ERR_ajout ("greffer") ;
                break ;
    case gauche : if (r->vue->gauche)
                {
                n->gauche = r->vue->gauche ;
                n->droite = s->vue         ;
                n->pere   = r->vue         ;
                n->droite->pere = n->gauche->pere = n ;
                r->vue->gauche  = n ;
                r->vue          = n ;
                }
                else
                {
                r->vue->gauche = s->vue         ;
                s->vue->pere   = r->vue         ;
                r->vue         = r->vue->gauche ;
                }
                break ;
    case droite : if (r->vue->droite)
                {
                n->gauche = r->vue->droite ;
                n->droite = s->vue         ;
                n->pere   = r->vue         ;
                n->droite->pere = n->gauche->pere = n ;
                r->vue->droite  = n ;
                r->vue          = n ;
                }
                else
                {
                r->vue->droite = s->vue         ;
                s->vue->pere   = r->vue         ;
                r->vue         = r->vue->droite ;
                }
    }
}

static arbrebin tailler (arbrebin r, direction d)
{
    arbrebin s ; noeud n ;
    s = creer () ;

    switch (d)
    {
    case racine : s->racine = s->vue = r->racine ;
                r->racine = r->vue = NULL      ;
                return s ;
    case droite : if (!r->vue) ERR_noeud ("tailler") ;
                if (r->vue->droite)
                {
                s->racine = s->vue   = r->vue->droite ;
                r->vue->droite->pere = NULL           ;
                r->vue->droite       = NULL           ;
                }
                return s ;
    case gauche : if (!r->vue) ERR_noeud ("tailler") ;
                if (r->vue->gauche)
```

```
            {
            s->racine = s->vue   = r->vue->gauche ;
            r->vue->gauche->pere = NULL           ;
            r->vue->gauche       = NULL           ;
            }
            return s ;
    case pere   : if (!r->vue) ERR_noeud ("tailler") ;
            if (r->vue->pere)
            {
                s->racine = s->vue   = r->vue->pere ;
                    if (r->vue->pere == racine)
                    ERR_ajout ("tailler") ;
                if (s->vue->pere->gauche == s->vue)
                    s->vue->pere->gauche = NULL ;
                else
                    s->vue->pere->droite = NULL ;
                r->vue = s->vue->pere ;
            }
            return s ;
    default : if (!r->vue) ERR_noeud ("tailler") ;
                s->racine = s->vue   = r->vue ;
            if (r->vue == racine) ERR_ajout ("tailler") ;
            if (r->vue->pere->gauche == r->vue)
                r->vue->pere->gauche = NULL ;
            else
                r->vue->pere->droite = NULL ;
            r->vue       = s->vue->pere ;
            s->vue->pere = NULL         ;
    }
}

static void ERR_vue ()
{
    printf ("ERREUR dans module %s : vue \n", nom_module) ;
}

static void ERR_noeud (char * mess)
{
    printf ("ERREUR dans module %s : fonction %s\n", nom_module, mess) ;
}

lier arbrebin ()
{
    arbrebin.creer     = &creer     ;
    arbrebin.detruire  = &detruire  ;
    arbrebin.existe    = &existe    ;
    arbrebin.aller     = &aller     ;
    arbrebin.lire      = &lire      ;
    arbrebin.ajouter   = &ajouter   ;
    arbrebin.oter      = &oter      ;
    arbrebin.remplacer = &remplacer ;
    arbrebin.fixer     = &fixer     ;
    arbrebin.retablir  = &retablir  ;
    arbrebin.deplacer  = &deplacer  ;
    arbrebin.greffer   = &greffer   ;
    arbrebin.tailler   = &tailler   ;
    arbrebin.afficher  = &afficher  ;
}
```

D.2 Le fichier Arbrebin.h

```
#define ARBREBIN_H
typedef enum {racine, pere, gauche, droite} direction ;
typedef void * arbrebin ;
extern struct
{
    arbrebin (*creer)     ()                                    ;
    void     (*detruire)  (arbrebin, void (*) (void *))         ;
    int      (*existe)    (arbrebin, direction)                 ;
    void *   (*lire)      (arbrebin)                            ;
    int      (*aller)     (arbrebin, direction)                 ;
    void     (*ajouter)   (arbrebin, direction, void *)         ;
    int      (*oter)      (arbrebin, void (*) (void *))         ;
    void     (*remplacer) (arbrebin, void *, void (*) (void *)) ;
    void     (*fixer)     (arbrebin, direction)                 ;
    int      (*retablir)  (arbrebin)                            ;
    void     (*afficher)  (arbrebin, void (*) (void *))         ;
    void     (*deplacer)  (arbrebin, direction)                 ;
    void     greffer      (arbrebin, arbrebin, direction)       ;
    arbrebin tailler      (arbrebin, direction)                 ;
} Arbrebin ;
extern lier_arbrebin () ;
```

D.3 Le fichier Arbrebin.tda

```
#ifndef ARBREBIN_H
#include "Arbrebin.h"
#endif

#define arbrebin(type_objet)                                        \
typedef struct nom (type_objet, arbrebin)                           \
{                                                                   \
    arbrebin   rep                          ;                       \
    struct nom (type_objet, arbrebin) *adr    ;                     \
    type_objet (*copier_objet) (type_objet)   ;                     \
    void       (*detruire_objet) (type_objet) ;                     \
    void       (*afficher_objet) (type_objet) ;                     \
} * nom (type_objet, arbrebin) ;                                    \
nom (type_objet, arbrebin) nom (type_objet, arbrebin_creer)         \
    (type_objet (*cop) (type_objet),                                \
     void       (*det) (type_objet),                                \
     void       (*aff) (type_objet))                                \
{                                                                   \
    nom (type_objet, arbrebin)  s ;                                 \
    s = nom(type_objet, arbrebin) malloc (sizeof (                  \
                struct (nom(type_objet, arbrebin))) ;               \
    s->adr            = s   ;                                       \
    s->copier_objet   = cop ;                                       \
    s->detruire_objet = det ;                                       \
    s->afficher_objet = aff ;                                       \
                                                                    \
    lier_arbre () ;                                                 \
    s->rep = (*Arbrebin.creer) () ;                                 \
    return s ;                                                      \
}
```

E.1 Le fichier Makefile

```
all : attente expression inf2pre inf2post  entier

attente :  Liste.o Liste.h File.tda attente.c
    cc -g -o attente attente.c Liste.o

expression : Liste.o Liste.h Pile.tda expression.c
    cc -g -o expression expression.c Liste.o

inf2pre : Liste.o Liste.h Pile.tda inf2pre.c
    cc -g -o inf2pre inf2pre.c Liste.o

inf2post : Liste.o Liste.h File.tda inf2post.c
    cc -g -o inf2post inf2post.c Liste.o

usager : Liste.o Liste.h Liste.tda usager.c
    cc -g -o usager usager.c Liste.o

entier : Liste.o Liste.h Liste.tda entier.c
    cc -g -o entier entier.c Liste.o

Liste.o : Liste.h Liste.c
    cc -g -c Liste.c

clean :
    rm -f a.out Liste Liste.o entier inf2post
```

Les sources des exemples présentés dans ce support sont disponibles à :
`http://www.fnet.fr/dhenin`

Proverbes de programmation [1]

Ne violez pas les règles avant de les apprendre.

Étude du programme

pp **??**, **??**, **??**, 11

- Un problème bien posé, est à moitié résolu
  Définissez-le aussi complètement que possible ; nous sommes habitués à résoudre des problèmes, peu (ou pas) à les poser
- Sachez ce que vous allez faire avant de le faire
- Utilisez l'étude descendante
- Méfiez-vous des autres études

Écriture du programme

- Construisez votre programme en unités logiques
- Utilisez des procédures
- Évitez des branchements inutiles
- Évitez les effets de bord
- Soignez la syntaxe tout de suite
- Choisissez bien vos identificateurs
- Utilisez proprement les variables intermédiaires
- Ne touchez pas aux paramètres d'une boucle
- Ne recalculez pas de constante dans une boucle
- Évitez les particularités d'une implantation
- Évitez les astuces
- Prévoyez des facilités de mise au point
- Ne supposez jamais que l'ordinateur suppose quelque chose
- Employez des commentaires
- Soignez la présentation
- Fournissez une bonne documentation

Exécution du programme

- Testez le programme à la main avant de l'exécuter
- Ne vous occupez pas d'une belle présentation des résultats avant que le programme ne soit correct
- Quand le programme est correct, soignez la présentation des résultats

De toutes façons

- Relisez le manuel
- Considérez un autre langage
- N'ayez pas peur de tout recommencer
- Ne compliquez pas inutilement les choses
  S'il y a une marche à suivre évidente
  utilisez-la
- L'évidence se nourrit de connaissances
- Élargissez le champ de l'évidence par la connaissance de programmes courts
- Précisez la forme des relations entre les variables
  Fixez les paramètres en examinant "à la main" le cas général et **les cas limites**
- Écrivez de nombreux programmes
  travaillez-les soigneusement
  vous enrichirez votre expérience
  vous développerez votre flair
- Si vous ne pouvez préciser les détails autrement que par des essais empiriques
  N'insistez pas
  il y a d'autres façons de faire.
- Traitez d'abord les cas les plus simples
- Sériez les questions
  Une seule question à la fois.
- Essayer un programme peut servir
  à montrer qu'il contient des erreurs
  jamais qu'il est juste
- Raisonnez pour que votre programme soit juste **par construction**
- Il faut se méfier **comme si l'erreur** était inévitable
- Pour comprendre un programme
  explicitez les situations qu'il engendre
- Pour créer le programme,
  il faut partir des situations
- Ne vous demandez pas
  que vais-je faire ?
  demandez-vous plutôt
  où en suis-je ?
- Pour construire une boucle
  proposez d'abord une situation générale
  Assurez vous que chaque pas
  rapproche de la solution
- Pour obtenir une situation générale
  Supposez qu'on **a fait une partie du travail**
- Déterminez dans quelles conditions
  le travail est fini
- progressez vers la solution
  et rétablissez la solution générale
- trouvez des valeurs initiales
  satisfaisant la situation générale

1. B. Mammeri, *Programmation*, École centrale de Paris, p. 6 et 7.

Table ASCII

Code ASCII

B i t S Hex					b_7	0				1			
					b_6	0				1			
					b_5	0		1		0		1	
					b_4	0	1	0	1	0	1	0	1
						0	1	2	3	4	5	6	7
						control		graphic input					
b_3	b_2	b_1	b_0	Hex				hight x & y		low x		low y	
0	0	0	0	0		0 0 NUL	20 16 DLE	40 32 SP	60 48 0	100 64 @	120 80 P	140 96 ‘	160 112 p
			1	1		1 1 SOH	21 17 DC1	41 33 !	61 49 1	101 65 A	121 81 Q	141 97 a	161 113 q
		1	0	2		2 2 STX	22 18 DC2	42 34 ”	62 50 2	102 66 B	122 82 R	142 98 b	162 114 r
			1	3		3 3 ETX	23 19 DC3	43 35 #	63 51 3	103 67 C	123 83 S	143 99 c	163 115 s
	1	0	0	4		4 4 EOT	24 20 DC4	44 36 $	64 52 4	104 68 D	124 84 T	144 100 d	164 116 t
			1	5		5 5 ENQ	25 21 NAK	45 37 %	65 53 5	105 69 E	125 85 U	145 101 e	165 117 u
		1	0	6		6 6 ACK	26 22 SYN	46 38 &	66 54 6	106 70 F	126 86 V	146 102 f	166 118 v
			1	7		7 7 BEL	27 23 ETB	47 39 ’	67 55 7	107 71 G	127 87 W	147 103 g	167 119 w
1	0	0	0	8		10 8 BS	30 24 CAN	50 40 (	70 56 8	110 72 H	130 88 X	150 104 h	170 120 x
			1	9		11 9 HT	31 25 EM	51 41)	71 57 9	111 73 I	131 89 Y	151 105 i	171 121 y
		1	0	A		12 10 LF	32 26 SUB	52 42 *	72 58 :	112 74 J	132 90 Z	152 106 j	172 122 z
			1	B		13 11 VT	33 27 ESC	53 43 +	73 59 ;	113 75 K	133 91 [	153 107 k	173 123 {
	1	0	0	C		14 12 FF	34 28 FS	54 44 ,	74 60 <	114 76 L	134 92 \	154 108 l	174 124 |
			1	D		15 13 CR	35 29 GS	55 45 -	75 61 =	115 77 M	135 93]	155 109 m	175 125 }
		1	0	E		16 14 SO	36 30 RS	56 46 .	76 62 >	116 78 N	136 94 ∧	156 110 n	176 126 ∼
			1	F		17 15 SI	37 31 US	57 47 /	77 63 ?	117 79 O	137 95 _	157 111 o	177 127 DEL

La documentation des ordinateurs ainsi que les ouvrages d'informatique fournissent la plupart du temps une table ASCII.

Cette table ci-contre est destinée à rappeler les principes et les repères qu'il est bon d'avoir en tête.

Elle ne présente qu'une demi-table, puisque l'autre partie est spécifique à chaque machine et à chaque périphérique ; la demi-table non présentée contient les codes des lettres accentuées et les caractères semi-graphiques.

Chaque caractère est représenté par un octet. Dans la table les 8 bits qui composent le caractère sont numérotés de 0 à 7, le bit 0 ayant le poids le plus faible.

La valeur en *binaire* peut être construite en lisant les valeurs des bits b_7 à b_0, la valeur en *octal* est inscrite en haut à gauche de chaque case, à droite est indiquée la valeur *décimale*. Enfin, la valeur *hexadécimale* est obtenue par juxtaposition du chiffre de la ligne horizontale (Hex) et du chiffre de la colonne (Hex) ; ainsi le caractère «A» vaut 41_{16}.

Les 32 premiers caractères sont des caractères de contrôle.

NUL	
SOH	CTRL A
STX	CTRL B
...	...
SUB	CTRL Z

Les chiffres commencent au caractère 48_{10}.

Le caractère 65_{10} est « A », chaque lettre minuscule (pax ex «a», 97_{10}) vaut le caractère majuscule + 32 («A», 65_{10}).

Table des figures